Towards instantaneous performance analysis using coarse-grain sampled and instrumented data

Author:
Harald *Servat Gelabert*

Advisor:
Prof. Jesús *Labarta Mancho*

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor per la Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors

Barcelona, 2015.

To *Txell*.

A la memòria de la *iaia*. Sempre seràs als nostres records.

Que el teu somriure i els teus panellets
alegrin el cor als teus nous companys de viatge.

Abstract

Computation is currently an integral part of scientific experimentation. There are many disciplines in which computers help to solve a myriad of scientific problems using a wide spectrum of algorithms implemented into programs. Some of these problems are so complex that a single computer would take too long that they become impractical to tackle. High-Performance Computing (HPC) systems (or informally, *supercomputers*) appeared to solve these problems and let the user do more science by offering more computational power. Current HPC systems consist of nodes containing multiple processors that communicate to each other using a fast network, resulting in a tightly coupled systems. If applications are to take advantage of these replicated resources they need to communicate and collaborate with one another to reach the final solution. To this end, applications need to divide their work into smaller work units and split them among the processors, so each processor works in parallel.

Nowadays, supercomputers deliver an enormous amount of computational power; however, it is well-known that applications only reach a fraction of it. There are several factors that limit the achieved application performance. One of the most important factors is the single processor performance (*i.e.* how fast a processor executes a work unit) because it ultimately dictates the overall achieved performance. Performance analysis tools are pieces of software that help to locate performance inefficiencies and identify their nature within applications to improve eventually the application performance. Performance tools rely on two collection techniques to invoke their performance monitors: instrumentation and sampling. The instrumentation mechanism is the ability inject performance monitors into concrete application locations whereas sampling it the capacity to invoke periodically monitors according to external events. Each technique has its advantages. The measurements obtained through instrumentation are directly associated with the application structure while sampling allows a simple way to determine the volume of the measurements captured. In any case, however, the granularity of the measurements that provides valuable insight cannot be easily determined *a priori*. Should analysts study the performance of an application for the first time, they may consider using a performance tool and instrument every routine or use high-frequency sampling rates in order to provide the most detailed results. More often than not, these approaches lead to large overheads that impact on the application performance and so altering the measurements gathered and therefore mislead the analyst.

This thesis describes the folding mechanism that overcomes the overhead by taking advantage of the repetitiveness of many applications. This mechanism smartly combines punctual instrumented and inexpensive coarse-grain sampled information and then generates rich reports that show the instantaneous performance evolution within instrumented regions of code. In order to produce these reports, the folding processes performance metrics from different types of sources: performance and energy counters, source code and memory references. The folding processes these metrics according to their nature. While performance and energy counters represent continuous metrics, the source code and memory references refer to discrete values that point out locations within the application code or the application address space, respectively. This thesis evaluates and validates two fitting algorithms used in different areas to report continuous

metrics: a Gaussian interpolation process known as Kriging and piece-wise linear regressions. Also, when reporting performance counters, it is crucial to correlate the value from different events at the same time to unveil the nature of the bottlenecks. The folding takes advantage of of analytic performance models derived from performance counters to focus on a small set of performance metrics. This fact avoids the analyst to dig into all the available performance counters and search for multiple reports to look for correlations. The folding also presents the correlation with the source code. To this end, the work described here also proposes two alternatives: using the outcome of the piece-wise linear regressions and a mechanism inspired by Multi-Sequence Alignment techniques. Finally, this thesis explores the applicability of the folding mechanism to captured memory references to give insight regarding the accesses to the application data object and report their temporal accesses.

This thesis proposes an analysis methodology for the first-time seen parallel applications that focus on describing the most time-consuming computing regions. This methodology is implemented on top of a framework that is based on a previously existing clustering tool and the folding mechanism. The clustering tool explores an application trace-file exploring the structural behavior of the application based on the performance data captured by instrumenting parallel programming run-times (such as MPI and OpenMP). The folding mechanism then uses the information on the identified structures to report their internal evolution and correlate with the source code, making the folding the perfect companion for the clustering tool. As a result, the framework finely accurately depicts the application performance using coarse-grain sampling and minimal instrumentation with the consequent savings in terms of overhead.

To demonstrate the usefulness of the methodology and the framework and therefore the value of the folding mechanism, this thesis includes the discussion of multiple first-time seen parallel in-production applications executed in several supercomputers. The discussions include a high level of detail regarding the application performance bottlenecks and correlate with tiny pieces of code that are responsible for most the execution time. Although many analyzed applications have been compiled using aggressive compiler optimization flags, the insight obtained from the folding mechanism has turned into small code transformations based on widely-known optimization techniques that have improved the performance in some applications. In addition, this work also takes advantage of the power monitoring capabilities of recent processors and discusses the simultaneous performance and energy behavior on a selection of benchmarks and in-production applications.

Contents

Tables

Figures

Listings

Acronyms

ALU Arithmetic-Logic Unit.
AOS Array of Structures.
ATX Advanced Technology Extended.

CPI Cycles Per Instruction.

DPI Duration Per Instruction.
DRAM Dynamic Random-Access Memory.
DVFS Dynamic Voltage and Frequency Scaling.

FLOPS Floating-point Operations Per Second.
FPU Floating-Point Unit.

GPGPU General Purpose Graphics Processing Unit.
GPU Graphics Processing Unit.

HPC High-Performance Computing.

IBS Instruction Based Sampling.
ILP Instruction-Level Parallelism.
IPC Instructions Per Cycle.
ISA Instruction Set Architecture.

L1D Level 1 Data-cache.
L2D Level 2 Data-cache.
LAPACK Linear Algebra Package.
LLC Last-level cache.
LSU Load Store Unit.

MIMD Multiple Instruction Multiple Data.
MIPJ Millions of Instructions per Joule.
MIPS Millions of Instructions per Second.
MISD Multiple Instruction Single Data.
MPI Message Passing Interface.
MSA Multi-Sequence Alignment.

NVRAM Non-volatile Random-Access Memory.

PC Program Counter.
PCU Package Control Unit.
PEBS Precise Event Based Sampling.

PMU Performance Monitoring Unit.

RAM Random-Access Memory.
RAPL Running Average Power Limit.

SIMD Single Instruction Multiple Data.
SISD Single Instruction Single Data.
SOA Structure of Arrays.
SPMD Single Program Multiple Data.

TDP Thermal Design Power.
TLB Translation lookaside Buffer.

VLIW Very Long Instruction Word.

1 Introduction

```
The cathedral was on fire.
It was done now.  There was no turning back.
```

— Ken Follet, THE PILLARS OF THE EARTH

Computers have become essential to almost every discipline they are used in. Disciplines such as chemistry, physics, engineering, earth sciences, life sciences and medicine, to name a few, bear strong testimony to that. Today's computers are being used to assist in generating accurate models for compound structures [200], studying the behavior of galaxies [11], improving the efficiency of wind turbines [7], forecasting tomorrow's weather [148], designing new drugs for therapy [171], and processing diagnostic images [115], respectively. These all involve problems that are tackled through a wide spectrum of methods implemented in computer programs. However, even with the computational capacity of a computer, some of these problems are too large to execute using the resources of a single computer; even, where resources are not a limitation, the problem may require too much time to be practical to tackle. Computers have evolved to cooperate in solving a problem that a single computer cannot undertake and to shorten the time-to-solution allowing the user to do more science instead of waiting for results. Although the improvements implemented into computers to make them faster, it is necessary to evaluate whether the applications take proper advantage of the computing characteristics. This thesis helps in this evaluation by introducing a mechanism named *folding* that exposes the instantaneous performance evolution of the existing repetitive behavior due to the nature of many methods used in the application without incurring in performance expenses. The work presented in this thesis not only covers evaluation through exposing many performance measurements, but also pointing to regions of code that is responsible for such performance and exploring the memory references to the application address space. This thesis also proposes a methodology and a framework for implementing it, that provide the analyst with very detailed reports that indicate the nature of the application performance inefficiencies and where their location within the application source code.

1.1 Introduction and historical background of supercomputing

High-Performance Computing (hereafter, HPC) is a discipline within Computer Sciences that involves studying, developing and manufacturing computer-based systems (named HPC systems

or more informally, *supercomputers*) with such computational power that they are able to solve the most computationally bounded and memory intensive applications within a practical period time. HPC systems have evolved significantly since their inception. Early HPC systems were based on serial vector processors specifically designed for performance (such as Cray I, marketed by Cray Research and installed at Los Alamos National Laboratory in 1,976). Vector processors are processors that operate vectors (or arrays) of data in addition to scalar operands as opposed to scalar processors that only operate on scalar operands. Their main benefits include reducing the number of control instructions executed, reducing the number of address translations and operating in parallel the values of the vectors. The main drawbacks, however, are their programmability and high memory bandwidth requirements. Vector-based processor systems were overtaken in performance by clusters of computers mainly when the cost of commodity components decreased and allowed the building of HPC systems on top of simpler processors and when the performance gap between the processor and the memory increased [232, 142]. A cluster of computers consists of a set of computers interconnected together using a fast network so that they can be viewed as a single system simplifying its programmability and cutting down the developer's efforts to get more computing power. In these systems, calculations are carried out simultaneously after dividing large problems into smaller ones that are solved independently in parallel and reducing the required time to solve the problem. Originally, these clusters were built using proprietary components but were later able to be built from commodity components (such as ASCI Red, built by Intel Corporation and installed at Sandia National Laboratories in 1,997). Nowadays, there is an increased transistor density that allows having multiple central processing units (cores) on each cluster node processor and while making the hardware accelerators more accessible. These devices are computational units separated from the processors that perform certain operations much faster than the processor itself. Currently, the most widespread accelerators are the general purpose graphical processing units (GPGPU, in short). Accelerators are orders of magnitude faster than regular processors, which makes them very attractive in HPC environments, even though they are mostly effective for stream-based processing problems. At the time of writing this thesis, the highest exponent of such developments is the Tianhe-2 supercomputer, developed by China's National University of Defense Technology and installed at National Super Computer Center in Guangzhou in 2,013. This supercomputer sums up more than 3 million cores and 1 Exabyte of memory distributed in 16,000 computing nodes where each computing node contains three accelerator chips.

The increasing computing capabilities of HPC systems rarely come without a price, however. As HPC systems evolve generation after generation, so does their complexity, making it so harder for a developer and/or user to benefit from the maximum system performance. For instance, the ubiquitous Top500 list [220] keeps a record of the most powerful HPC systems twice a year and uses the Linpack benchmark [50] to determine the system performance by solving a dense system of linear equations. Benchmarks are synthetic applications that approximate the behavior of a specific workload, so the result of a benchmark is biased towards stressing some - but not all - of the system's components, so the result of sole benchmark may not represent the performance of the system on the actual workloads. The Linpack benchmark, for instance, simply evaluates the performance focusing on arithmetic instructions but ignoring the remaining instructions (including data movement, value testing and branching, among others). The benchmark returns the number of arithmetic operations per unit of time (commonly known as floating-point operations per second and abbreviated as Flop/s) as well as the system solution. Table 1.1 tabulates several performance milestones of the Top500 list, including their year of installation, theoretical maximum performance (R_{peak}), achieved performance (R_{max}) and achieved efficiency (R_{max}/R_{peak}) in order to show the performance evolution of several HPC systems. While the benchmark is specifically tuned for the system where it runs, it may be observed that the benchmark cannot take relevant advantage of all the computational power of the systems, except for the Earth Simulator and the K computer[1]. Quite the opposite, i.e. some systems seem to have been developed to achieve the best results on workloads similar to these benchmarks, overlooking the remaining type of workloads.

[1] Chapter 2, Section 2.3.4 gives some insights on why these systems outperform the rest with respect to efficiency.

Table 1.1
Performance milestones of the Top500 list sorted by installation year. R_{max} and R_{peak} refer to the maximum Linpack performance and the peak performance of the system, respectively.

System name	Year	Clock rate (MHz)	R_{max} (Flop/s)	R_{peak} (Flop/s)	$\frac{R_{max}}{R_{peak}}$
Cray Y-MP	1,988	167	2.1×10^{9}	2.6×10^{9}	0.80
Numerical Wind Tunnel	1,993	105	12×10^{9}	23×10^{9}	0.52
ASCI Red[2]	1,997	333	1.1×10^{12}	1.3×10^{12}	0.81
Earth Simulator	2,002	500	35.8×10^{12}	40.9×10^{12}	0.87
BlueGene/L	2,005	700	280×10^{12}	367×10^{12}	0.76
IBM RoadRunner[3]	2,008	1,800[4]	1.0×10^{15}	1.3×10^{15}	0.74
K computer	2,011	2,000	10.5×10^{15}	11.2×10^{15}	0.89
Tianhe-2	2,013	2,200[4]	33.8×10^{15}	54.9×10^{15}	0.61

The next milestone in HPC is the ExaFlop system, a machine that executes at least $10^{18} Flop/s$ sustainedly and performs $30x$ compared to the fastest machine in the current Top500 list. The projection of the performance of the first machine on the Top500 list along time and some estimates [49] predict that the first ExaFlop system is due between 2,019 and 2,020. In addition, the processor frequency has stalled [181, 49, 57, 155, 177] mainly due to energy constraints. Since the processor is the main responsible for the node-level performance, it becomes clear that the stall on the processor frequency restricts the performance gains of current HPC systems and limits further increase to an improved use of the available resources.

1.2 Performance analysis

In every new system generation, users expect systems to execute their applications faster; however, it is becoming natural with each newer generation application for developers to use newer development approaches and require deeper system knowledge to achieve the best performance. Currently, there are many topics involved on achieving a competent efficiency such as data dependencies, memory hierarchy, instruction-level and task parallelism, symmetric multiprocessing, cluster computing, programming models, compiler suite and flags, network topology, bandwidth and latency, to name a few. It is therefore legitimate for users to ask themselves whether their applications are taking the proper benefit of the resources and they do not rely only on the behavior of a single benchmark. Similarly, computer architects design and plan future processors so as to improve the performance and they need feedback to improve future processor generations. Also, it is likely that system administrators wish to know the conditions under which their user's applications are optimally executed to achieve the best use of the resources.

While this thesis focuses on a particular topic of the performance analysis area of study that aims at improving the performance of an application, it is noticeable that performance analysis can be divided in the following parts:

performance measurements covers capturing information (and its posterior analysis) to unveil whether the application runs optimally on a system, or if the application presents any bottlenecks.

performance simulation (or prediction) assists in predicting the application performance on systems that do not currently exist, or exploring alternative optimization techniques.

performance models describe the performance behavior of an application or a system by using abstract concepts.

[2] First system to reach the TeraFlop/s ($10^{12} Flop/s$) rate.
[3] First system to reach the PetaFlop/s ($10^{15} Flop/s$) rate.
[4] This is an heterogeneous system. The clock rate reported refers to the main computing nodes, not to the accelerators.

Performance measurement tools (or performance tools) come into play to answer the user questions relating to the behavior of an application in a specific machine. Performance tools are pieces of software that deliver comprehensive details of the application behavior, both quantitatively and qualitatively. These tools expose metrics that describe how an application behaves by providing metrics such as execution time, number of function calls, the number of bytes transferred, the number of instructions executed, the number of elapsed cycles and determining whether the work is balanced. This information not only helps the user to understand how the program behaves, it also provides details on how it interacts with different system aspects pointing out what bottlenecks (if any) there are and why and where. While knowing these details is important, the most beneficial outcome for the user is an increased possibility of taking greater advantage of the system resources. Ultimately, this knowledge will lead to modifications based on the identified diagnostics and may increase the application performance, which directly translates into a reduction for the time-to-solution and enables the software to solve a larger problem size.

Since the inception of the performance measurement tools, there have been several alternatives for providing feedback to the analyst. For instance, tools used several decades ago [230, 174, 27, 28, 201, 170] already required choosing the best approach to provide valuable insight to the user. The alternatives these tools had are similar to currently available options, no matter the technology has progressed. Still nowadays, there are several ways to classify these tools, including: how measurements are stored and presented to the user and how measurements are captured within the application. With respect to the first group, there are some tools that collect a time-stamped sequence of measurements (trace-files), enabling the analyst to understand the temporal evolution of the application and observe variations in time during the process execution. However, since trace-files may turn into a huge amount of data for long runs or runs involving a large amount of processes, other tools summarize all the performance data and simply present first order statistics (such as average, mean, count, max and min) correlated with application routines. Each approach has its own advantages and drawbacks and while the scope of this thesis does not cover the matter of which approach is better, it is worth mentioning that it is framed in performance tools that use time-stamped trace-files.

Regarding the collection of performance measurements, data collection uses monitors invoked by either sampling or code instrumentation mechanisms. On the one hand, sampling periodically executes monitors injected into the program by interrupting it using timer interrupts and/or operating system signals. When using sampling, the cost associated with each sample interval is correlated with the routine that was being interrupted, therefore the results obtained using this mechanism are statistical approximations. Such statistical inference on the behavior requires the application to run during long enough for the results to approximate the actual distribution, though highly volatile metrics may not be captured. On the other hand, instrumentation refers to injecting monitors into the target application at certain points. With instrumentation, metrics are accurately correlated to the program source code because monitors are associated with components with syntactical structure within a program (such as routines, loops or sentences). However, as monitors are injected at specific code locations, their invocations completely rely on the application control flow, which means that the granularity and the volume of the performance metrics gathered directly depends on the activity of application.

It is beyond dispute that the more details a performance tool provides, the deeper the understanding of the application that the tool may offer, but at the cost of dealing (both the user and the tool) with large amounts of data. However, a tool that uniquely relies on instrumentation cannot provide additional metrics between two instrumentation points. The same occurs with sampling-based performance tools, it is impossible to finely depict what occurs between two samples. To increase the granularity of the data, these tools have two alternatives with respect to measurement collection: inserting additional instrumentation monitors or using higher sampling frequencies. For instrumentation based tools, this may be particularly difficult because it requires *a priori* knowledge of the application control flow in order to place the instrumentation monitors into the source code adequately. Focusing on the overhead, it does not matter which alternative is adopted, the application suffers from the *observer effect* (*i.e.* the performance is altered) because monitors occur in first person and so need to interrupt the application to collect measurements. An indiscriminate increase of the number of monitors executed ends up altering the application

performance and therefore the measurements, so these measurements may become misleading or even useless for an ulterior analysis. The main objective of this thesis aims is to combine instrumented and sampled data to depict performance phases that are finer than the sampling frequency between instrumentation points without increasing the overall overhead.

1.3 Contributions

While increasing the number of monitors injected into the application may shift the results into non-representative performance data, so rendering these data useless for an analysis, it is still possible to combine both instrumentation and sampling mechanisms. The research described in this thesis aims at combining performance metrics that come from these two mechanisms and its main contribution is a mechanism named *folding*. The folding process is capable of providing instantaneous node-level performance evolution of regions to unveil whether the processor resources are being used efficiently, determine the nature of any bottleneck and identify the source code regions that are responsible for such performance. This thesis demonstrates that applying the folding to regions that are executed multiple times allows a detailing of the performance evolution within these regions of code even if the sampling period is coarser than the region period.

While the folding itself is the major contribution, this thesis also offers additional research that takes advantage of this mechanism. The folding mechanism is applied to measurements collected through a performance monitoring tool and includes data such as performance and energy counters as well as, source code and memory references. Due to the different nature of the performance counters and the source code references, this thesis evaluates different approaches to process both performance data types. It also explores the applicability of the folding mechanism to memory references to depict the temporal evolution of the accesses to the application data objects.

This thesis also introduces a methodology to evaluate parallel applications. Such a methodology has been implemented in a framework that relies on a previously existing tool. The framework generates the detailed evolution of the application bottlenecks, their nature and their associated core on the most time-computing regions within a parallel application. These reports are generated by applying minimal instrumentation to the parallel application and sampling at very coarse-grain frequencies, so it does not penalize the application execution. In addition, these reports are obtained through a single execution on parallel application binaries even if the analyst does not have any former knowledge from the application, so saving computing resources.

The thesis goes beyond the performance analysis and shows that the folding mechanism can be applied to other studies such as an energy-consumption analysis. Last, but not least, this thesis details multiple studies on first-time seen applications to demonstrate the process' usefulness. In some cases, these studies include applying small modifications on the application source code which end up improving the application performance according to the findings of the folding results.

1.3.1 Folding mechanism

Many applications executed in HPC systems present a strong repetitive behavior because the nature of the problem they are meant to solve using numerical analysis theory. Consequently, these applications are structured as an iterative sequence of routines and loops which means they periodically expose the same performance from one iteration to the next and each iteration exposes different performance phases as the application advances from one loop to the next within the same iteration. Because of these characteristics, the application performance at a particular time since the start of a repetitive region will be the same independently from the region that is being executed, which means that the performance of every region can be considered an ergodic system.

The folding exploits this latter quality to show detailed performance progression within regions of code by combining metrics that come from instrumentation and sampling mechanisms.

Within the folding, the instrumentation and sampling collected metrics play different roles. While the metrics captured in the instrumentation points are used to delimit repetitive regions (such as routines and loop bodies), the metrics gathered from sampling points are used to report progress within the regions. The process receives this name since it intuitively takes each region independently and folds (or superimposes) them to generate a synthetic region and the more instances folded, the more details within the synthetic region. The benefits of the folding are twofold. First, it exposes precise performance characterization of regions even using coarse grain sampling, so allowing a precise performance characterization while keeping the overhead that the application suffers to the minimum. Second, when analyzing applications for the first-time, the process might provide enough insight even if the captured metrics do not characterize the regions of interest sufficiently, which turns into a reduced number of runs and, consequently, saves computing resources.

As stated, the folding mechanism requires some instrumentation markers to delimit the region of interest. Needless to say, applications are typically large in terms of lines of code and delimiting the application source code to apply requires a certain knowledge of the application that the analyst may not have without incurring on an additional execution or techniques such as source code analysis. To circumvent this issue, this thesis introduces a methodology to study first-time seen parallel applications. This methodology is supported by a framework based on a tool that categorizes the application computing regions through performance metrics [85] and also by the folding process to described the node-level performance of these computing regions. Since both tools rely on minimal instrumentation on parallel run-time calls and coarse-grain sampling, there is minimal intrusion during the application execution.

The thesis also covers different implementation alternatives, discusses how the results change depending on the alternative used and, more importantly, examines further analysis benefits from the alternatives considered. For instance, the folding requires the interpolation of intermediate results and this thesis evaluates the use of two contouring algorithms. The following publications describe and evaluate the first approach for the folding mechanism when adopting a contouring algorithm used in geo-statistical studies [221]. They also demonstrate the usefulness of the framework by discussing and analyzing multiple parallel applications:

- [188] Harald Servat, Germán Llort, Judit Giménez, Jesús Labarta: **Detailed Performance Analysis Using Coarse Grain Sampling**. In proceedings of the *Workshop on Productivity and Performance (PROPER)* in conjunction with Euro-Par 2,009: 185-198.
- [193] Harald Servat, Germán Llort, Judit Giménez, Kevin A. Huck, Jesús Labarta: **Unveiling Internal Evolution of Parallel Application Computation Phases**. In proceedings of *International Conference on Parallel Processing (ICPP)* 2,011: 155-164.
- [189] Harald Servat, Germán Llort, Judit Giménez, Kevin A. Huck, Jesús Labarta: **Folding: Detailed Analysis with Coarse Sampling**. In proceedings of the *Parallel Tools Workshop* 2,011: 105-118.

1.3.2 Applicability of performance models

To report the node-level performance, the folding results include detailed evolution of hardware performance counters for the repetitive regions. The increasing complexity of the microprocessor has resulted in an increase in the number of performance counters and also in semantics strongly linked to the microprocessor architecture. Yet users can apply the folding mechanism to a number of these counters when analyzing an application, the resulting number of plots and the data they present may be intimidating. Several performance models have been described outside this thesis in order to overcome such complexity and simplify the use of the performance counters. For instance, the IBM CPIstack model for the IBM®Power5® [202] and the IBM®Power7® [62] processors and a model described for the Intel®Itanium2®processor [108] help to identify the source of the stalled cycles within the processor. To ease the comprehension of the application performance, while still providing detailed evolution, the folding results are combined with these performance models to generate a summarized evolution of the performance metrics.

This work resulted in the following publication:

- [190] Harald Servat, Germán Llort, Kevin A. Huck, Judit Giménez, Jesús Labarta: **Framework for a productive performance optimization**. In *Parallel Computing journal*, 39(8): 336-353 (2,013).

1.3.3 Source code and performance correlation

The correlation between performance inefficiencies and their associated source code has become a cornerstone to understanding why the efficiency of an application falls behind the computer's peak performance and to ultimately enabling optimizations on the application code. To this end, performance analysis tools rely on collecting the processor call-stack and then combine this information with performance measurements to enable the analyst understand the application behavior. This work explores the capabilities that the folding mechanism offers for correlating performance and source code, to allow the analyst to easily understand the application's bottlenecks. Although source code references do not benefit from fitting models as performance counters, this work explores two approaches to establish an approximate correlation between performance and source code.

The first approach benefits from detecting phases derived from the performance results and associating each phase with a routine or loop. This approach complements the initial implementation of the folding mechanism and uses piece-wise linear regressions, which have been used in bio-medicine [158] and financial studies [153]. The main benefit of piece-wise linear regressions applied to the folding results is that they help to determine where the performance changes between two consecutive performance phases. These phase breaks enable correlations between the source code and each phase and allow the analyst to focus on very small regions of source code when studying each of the phases, ideally at loop level.

The second approach develops a mechanism inspired on Multiple Sequence Alignment (MSA) algorithms by treating samples as molecules. The goal of this approach is to reduce the cost of capturing the processor call-stack information no matter its depth and to make it uniform. This way the folded data are treated like independent sequences of biological molecules that are aligned using MSA algorithms and then the most representative routines are provided to the analyst.

This work resulted in the following publications:

- [191] Harald Servat, Germán Llort, Juan González, Judit Giménez, Jesús Labarta: **Identifying code phases using piece-wise linear regressions**. In proceedings of the *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS)* 2,014: 941-951.

- [186] Harald Servat, Germán Llort, Juan González, Judit Giménez, Jesús Labarta: **Bio-inspired call-stack reconstruction for performance analysis**. Technical report UPC-DAC-RR-2014-20.

1.3.4 Energy consumption analyses

This thesis shows that the folding is not limited to performance measurements, but can manipulate other metrics gathered by the instrumentation and sampling monitors. For instance, the processor is responsible not only for the node-level performance but also for most of the energy consumed by a system, which means that it is important to have tools to analyze both performance and energy efficiency. The folding allows correlating the evolution of drained power, in addition to performance metrics.

This work resulted in the following publication:

- [187] Harald Servat, Germán Llort, Judit Giménez, Jesús Labarta: **Detailed and simultaneous power and performance analysis**. In *Concurrency and Computation: Practice and Experience* journal, (2,013).

1.3.5 Detection of memory access patterns and their time evolution

The memory hierarchy is becoming more and more sophisticated as processors evolve generation after generation. Its advances respond not only to address the speed divergence between the processor and the memory outside the chip, but also to reduce the energy dissipated by the data movement. Processor manufacturers have typically organized the memory hierarchy in different *strata* to exploit the temporal and spatial localities of reference. The memory hierarchy ranges from the extremely fast but tiny and power-hungry registers to the slow but huge and less energy-consuming DRAM, including multiple cache levels. Still, some processor researchers and manufacturers are looking for opportunities to extend the memory hierarchy to improve the application execution in terms of performance and energy. Their research considers additional integration directions so that the memory hierarchy adds layers as scratchpad memories, stacked 3D DRAM [134] and even non-volatile RAM [228].

This work explores the incorporation of the application address space perspective into the folding mechanism to unveil the access patterns and the locality of reference to the application data structures. Such an extension relies on the address sampling mechanisms offered by the PMU extension known as PEBS [37] or IBS [55] from Intel and AMD, respectively. The result of this combination provides complete support to gain insight of the application behavior, including the application syntactical level, its data structure organization and its memory hierarchy use and achieved performance.

This work resulted in the following publication:

- [192] Harald Servat, Germán Llort, Juan González, Judit Giménez, Jesús Labarta: **Low-overhead detection of memory access patterns and their time evolution**. Accepted for publication in the 2,015 edition of Euro-Par conference. Technical report UPC-DAC-RR-2015-01.

1.4 Thesis organization

The rest of this thesis is organized as follows: Chapter 2 briefly introduces some generalities in parallel programming that are used in the rest of thesis. Chapter 3 then details some aspects used across many performance tools and serves to describe other existing performance tools, comparing them with the work described in this thesis. Chapter 4 introduces and discusses the main contribution of this thesis: the folding process. Since this process applies to several types of measurements such as performance counters, power readings and source code and memory references, the discussion includes several implementations to report this information back to the analyst. The Chapter also introduces a methodology and a framework that can help an analyst to evaluate the performance of a parallel application without requiring former knowledge of the studied application in a single execution. Chapter 5 demonstrates the usefulness of the framework described by evaluating a number of first-time seen, in-production parallel applications. Chapter 6 draws some of the conclusions drawn from this thesis as well as further research directions. Finally, there are a number of appendices that include the folding manual and a basic description of the folding implementation.

2
Parallel programming

```
07.00 Decido salir en busca de Gurb.
```

— Eduardo Mendoza, SIN NOTÍCIAS DE GURB

There are a variety of approximations in HPC to make a parallel version of an application that takes the most benefit of supercomputers. This chapter revises Amdahl's law and describes computer systems according to their design points and the relations among their parts. The chapter follows introducing the most widespread mechanisms to parallelize applications in the HPC environment, from the cluster to the instruction level, with an assorted selection of examples. Although the examples are intentionally keep as simple as possible, they prepare the reader for the forthcoming chapters these topics are widely used.

2.1 Amdahl's law

Current HPC systems offer a vast amount of computational power because they are built on top of replicated components which are interconnected through a fast network, resulting in a tightly coupled systems that enable parallel executions at different levels of the system. A *naive* view of this replication depicts an HPC system as a collection of multiple computers, where each computer contains many processors, each processor accommodate several central processing units and each central processing unit works on different data sets at a time. So that applications take advantage of these resources, applications need to be *parallelized*, which means that the work of the application must be split into portions that can be simultaneously executed by different levels of the system replicas. Ideally, a user expects that an application executed on N processors lasts $1/N$ of the duration of the serial execution ($T(1)$); or in other words, executing a parallel application on N processors becomes N times faster so resulting in a reduced execution time ($T(N)$) inversely proportional to the number of processes. However, this is not necessarily true because the speedup (i.e. the ratio between the execution times in serial and parallel) achieved by executing a parallel application on N processors depends on many factors, mainly the application and the system. In exceptional cases such as those that memory hierarchy plays a significant role on the computation speed, the increase of the number of processing units increase the amount of cache capacity, the speedup may exceed the number of processors. Still, more often than not, the speedup is lower than the number of processors. Amdahl's law [5] states that

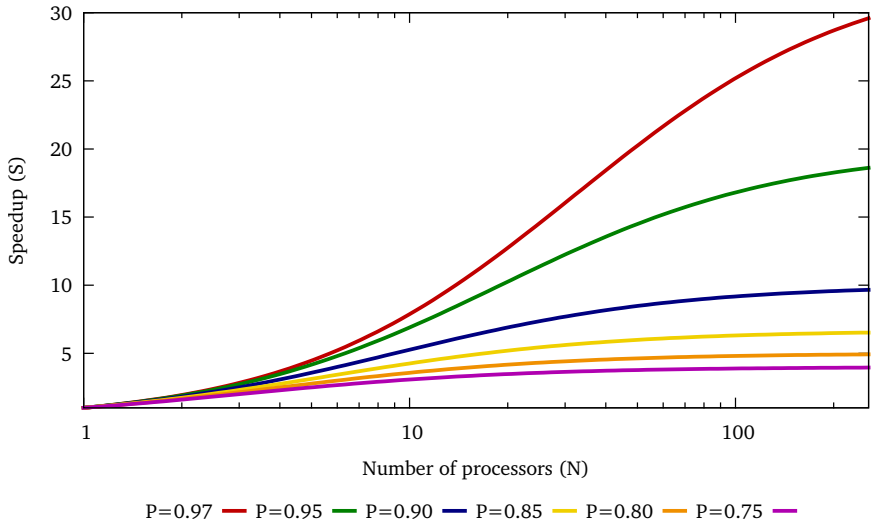


Figure 2.1
Graphical representation of Amdahl's law at different parallelization factors using up to 256 processors.

the theoretical maximum speedup (S) achieved on N processors depends on the portion of the algorithm that is executed in parallel ($P \in [0,1]$) and it is defined by:

$$S(N) = \frac{T(1)}{T(N)} = \frac{T(1)}{T(1)\left((1-P) + \frac{1}{N}P\right)} = \frac{1}{(1-P) + \frac{1}{N}P} \tag{2.1}$$

This equation points out that the serial part of the algorithm $(1-P)$ limits the overall execution time, therefore, it must be minimized to reach the maximum speedup. This minimization is what the parallel programming pursuits: attempting to reduce the serial portion so that the application execution time gets as small as possible by increasing the use of the available resources. Figure 2.1 shows the maximum expected speedup for a variety of parallelization levels (P ranging from 0.75 to 0.97) of an application when using multiple processors according to Equation 2.1. If only 3% of the total execution time is executed in serial, the achieved speedup is roughly 25 when using 100 processes. Therefore, if developers focus on making their applications scalable and take advantage of current and forthcoming supercomputers, they need to maximize their in-parallel execution time.

2.2 Parallel computer architecture taxonomy

There is a breadth of design alternatives for parallel computers. Flynn proposed a simple model to classify them based on the available number of concurrent instructions and data streams in the architecture [69]. The four resulting categories of such classifications are shown in Table 2.1 and, while there are some hybrids of these categories, they are summarized like:

SISD Represents a sequential computer without any attempt to exploit parallelism at neither instruction nor data level. There is a single control unit that fetches instructions one at a time and where each instruction only operates to scalar values.

MISD A very uncommon architecture where many instructions work on the same data. While some people object, systems that may belong to this group are fault tolerant systems that include multiprocessors running at lockstep to compare the results every cycle.

Table 2.1
Flynn's taxonomy.

	Single instruction	**Multiple instructions**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

SIMD A computer architecture that operate on vectors (or arrays) of values using one single instruction.

MIMD Multiple autonomous processors where each simultaneously execute different instructions on different data.

During the last years, there has been a rise of the MIMD architectures, mainly because they are built with cost/performance ratio advantages and offer an unprecedented flexibility. Multiprocessors are built today using the same processors found in workstations, which reduces design and implementation costs. MIMD architectures operate either as a single-application machine, or as a multiprogramming machine executing many independent tasks, or even as a combination of both, although these systems require appropriate hardware and software support.

A further categorization of parallel applications that run on MIMD architectures is based upon the technique employed to achieve parallelism. In Single Program Multiple Data (SPMD) applications, their processes run the same program operating on different data and in contrast, Multiple Program Multiple Data (MPMD) applications, their processes execute different programs on different data. The SPMD model is particularly appropriate for regular problems and exposes predictable communication patterns between computation phases. On the other side, MPMD model relies on manager process(es) to control a number of worker processes, which is typically implemented either on different binaries or in a single binary using different control flows. The model election must be selected according to the problem to solve and ultimately by the developer, but the fact that workers in MPMD applications have to communicate with the managers may limit the scalability due to a communication bottleneck.

There is an additional division of MIMD architectures that depends on the memory organization and interconnect strategy. The first group, called shared-memory architectures, interconnects memory and a modest number of processors with a bus. This multiprocessor offers a single memory address space that is used by all the processors and consequently, processors communicate through regular load and store instructions. Currently, there has been a raise in the multi-core processors due to a reduction in transistor size and it is very common to see multi-processor systems where each processor is multi-core. A multi-core processor is a single chip processor that contains two or more independent central processing units which may share some resources (such as the memory bus, the memory hierarchy and the I/O buses), so effectively, a multi-core processor becomes a shared multi-processor itself within a chip. The second group of MIMD architectures, named distributed-memory architectures, represents systems where processors have their own private memory and communicate with other processors using special purpose communication instructions that are nowadays leveraged to network devices. While the connectivity to the network depends on the network topology, these architectures allow connecting a larger amount of processors than shared-memory architectures. Nowadays, HPC systems combine these two types of architectures so that each computer from the system includes multiple processors and each computer is connected to the rest of the system using fast networks. For instance, the Marenostrum III supercomputer [138], which is installed at Barcelona Supercomputing Center, is currently a system consisting of 3,056 compute nodes interconnected through an Infiniband network and where each compute node contains 2 octo-core ($8x$) processors.

2.3 Parallelization alternatives

There exist various parallel programming models, languages, libraries and application programming interfaces (API) to write applications that benefit from the resources of HPC systems depending on the assumptions they make about the memory architecture. This section is divided

into several parts that describe the current techniques in parallel programming and each part describes how to take profit of the parallelism provided by the resource replication at different levels of the system, from the system to the processor level. First, there is an introduction to message-passing paradigms, which are typically used to communicate processes executing in different nodes of the system. The following section provides some basic principles of shared-memory paradigms, which are appropriate to assist in parallelizing applications within a node. The section continues by providing introductory foundations of parallelism within a single processor using current approaches to parallelizing through instruction-level parallelism (ILP) as well as SIMD instructions. Finally, the section finishes by providing a brief discussion on the use of accelerators to execute applications in parallel. So as to achieve the desired performance, it is necessary to optimize the resource use at each level of the system, which may imply combining several parallelization schemes to produce hybrid applications.

2.3.1 Message-passing paradigm

Starting a top-down perspective of HPC systems, these are a collection of individual nodes that are interconnected by high-speed networks exposing a distributed-memory architecture. So that applications benefit from this parallelism, they require some inter-process communication to coordinate and exchange data. Applications that make use of these inter-process communication to solve a common problem are known to use the message-passing paradigm. Although message-passing applications do not necessarily require to be written for this paradigm from the scratch, due to the requirements for the organization, coordination and computation in a distributed environment, the process of porting to this environment requires an amount of time.

There are several message-passing standards such as Remote Procedure Call (RPC) [206], Common Object Request Broker Architecture (CORBA) [163], Java Remote Method Invocation (Java RMI) [38] and Desktop Bus (D-Bus) [178]; but the most extended standard in the HPC community is the Message Passing Interface (MPI) [51, 146]. When running an MPI application, the MPI launcher spawns the binary (or binaries if the application is MPMD) onto the resources creating multiple processes. These processes run independently except when they need to initialize and finalize as well as to communicate to the rest of the processes spawned.

To ease the development of MPI applications, the standard provides a rich range of abilities that include: communicators, point-to-point operations, collective operations and derived data types. Communicators are logical objects that group processes within the execution of the MPI application and they are mainly used to allow communicating between the grouped processes. By default, MPI applications have two communicators, one that groups all the processes involved in the computation and another that contains each process itself, but MPI allows creating new communicators based on groups of processes.

Point-to-point operations involve communication between two processes from the application that are connected through a communicator. The most basic point-to-point operations are `MPI_Send` and `MPI_Recv` (shown in Listing 2.1), which send and receive a message to/from a particular process, respectively. A message in this context is formed by data of a particular data type and identified by a tag as shown in their declarations for the C language.

Listing 2.1
Basic MPI point-to-point routines.

```
int MPI_Send (void *buf, int count, MPI_Datatype datatype,
  int destination, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Collective operations distribute data among all processes belonging to a communicator and they are meant to save the developer the effort of having to invoke many times the point-to-point

operations to distribute messages to multiple partners with the consequent expense saving. While the standard describes a wide variety of collective operations, the operations `MPI_Bcast`, `MPI_Reduce`, `MPI_Scatter`, `MPI_Gather` and `MPI_Alltoall` can be highlighted to illustrate the functionality of the collective operations and are shown in Listing 2.2. The first and third operations are aimed at distributing messages from one process to all the processes that are involved in a communicator. Although on `MPI_Bcast` each receiver gets a copy of the whole message, on `MPI_Scatter` the message is divided among the receivers. Operations `MPI_Reduce` and `MPI_Gather` are intended to collect messages from many processes by one process. In the particular case of `MPI_Reduce` the values received are processed by a given function (`sum`, `max`, `min`, *e*tc...), whereas `MPI_Gather` simply concatenates all the messages received onto a larger message. Finally, it is worth mentioning the `MPI_Alltoall` operation, which allows data to be sent and received from and to all processes in the communicator.

Listing 2.2
Basic MPI collective routines.

```
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm);

int MPI_Reduce (void *sendbuf, void *recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

int MPI_Scatter (void *sendbuf, int sendcnt,
  MPI_Datatype sendtype, void *recvbuf, int recvcnt,
  MPI_Datatype recvtype, int root, MPI_Comm comm);

int MPI_Gather (void *sendbuf, int sendcnt,
  MPI_Datatype sendtype, void *recvbuf, int recvcnt,
  MPI_Datatype recvtype, int root, MPI_Comm comm);

int MPI_Alltoall (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf, int recvcount,
  MPI_Datatype recvtype, MPI_Comm comm);
```

Finally, it may well be that the application data are not homogeneous or contiguous in memory and therefore do not fit in predefined MPI types (such as integer, char, boolean and floating-point numbers). Under these circumstances, the application needs to convert its own data types into MPI data types, execute the MPI operation and then convert the received data from the MPI data types into its data types again, with the consequent overhead. MPI allows defining new data types to amortize the overhead of translating application data structures into MPI types as well as it would be more productive for the application developer which avoids programming such transformations in every place where the application needs them.

To exemplify the use of the MPI standard, consider first the code shown as Listing 2.3 that approximates the value of π through a quadrature method. In this code, each iteration of the loop contributes in the result independently and its parallelization is pretty simple. Table 2.2 shows two parallel versions derived from the serial code using MPI routines, the one the left uses MPI point-to-point operations (`MPI_Send` and `MPI_Recv`) and the one on the right uses MPI collective operations (`MPI_Bcast` and `MPI_Reduce`). While both versions approximate the same result, it is noticeable that the version that uses collective operations is significantly shorter. On these versions, each MPI process calculates a partial value of `area` in the variable `tmp2` and then, each process transmits the value of `tmp2` to process zero, where it is accumulated.

The previously stated MPI routines block the execution of the process until the communication has finished. The consequences of this blocking behavior includes situations where two or more processes are waiting for others to finish (a situation commonly known as *deadlock*) as well as that the application cannot computationally progress as long as it waits for a message to

Listing 2.3
Serial version to approximate the value of π.

```
h = 1.0 / n;
for (i = 1; i <= n; i++)
{
  x = h * (i - 0.5);
  area += (4.0 / (1.0 + x*x));
}
pi = h * area;
```

arrive. To circumvent these issues, the MPI standard introduced non-blocking point-to-point operations in MPI version 1 and non-blocking collective operations in MPI version 3. Although avoiding deadlocks is necessary to allow a correct execution of the application, with respect to performance, the non-blocking communications allow developers to overlap communications with computation so increasing the parallelism achieved.

2.3.2 Shared-memory paradigm

Although message-paradigm applications take advantage of independent processes possibly executing in separate nodes, the shared-memory paradigm allows multiple processes from the same node to access and exchange data by accessing a unique address space. This way, communication between processes is very fast and transparent to the user, as opposed to the message-passing paradigm. These shared-memory programming paradigms naturally match on top of the shared-memory architecture, so that each processor connected to the memory bus can easily access to the whole address space without requiring special instructions. The most extended parallel programming model using this paradigm in HPC systems is OpenMP [42, 166], but there are others such as pthreads [127], OmpSs [56], Cilk and Cilk Plus [18, 126] and Intel Threading Building Blocks [180], to name a few. The reader can even find additional languages such as Unified Parallel C (UPC) [61] and Co-Array Fortran [161] where the shared address space is partitioned and a portion of it is local to each thread or process, no matter the system is shared or distributed memory.

Since OpenMP is the most shared-memory paradigm in HPC, this section gives some insight about it. OpenMP is a portable standard for shared memory multiprocessing that mainly uses program annotations (`#pragma` constructs in C/C++ and special directives in Fortran) to indicate which regions of code are meant to be executed in parallel. Compared to MPI, OpenMP offers a simpler parallel scheme and also an incremental approach; however, since shared-memory architectures do not scale to large number of processors, the maximum expected scalability of OpenMP applications, compared to MPI applications, is far more reduced. The fact that the whole application shares a unique address space and that it is turned into parallel by adding some program annotations simplifies the application development and, significantly important, and increases productivity when parallelizing an existing serial code. In addition, the developer easily decides when to stop adding OpenMP directives, for instance when a particular target speed-up has been reached, as the application will normally work since non-parallel regions execute serially. Compared to MPI, the message-passing applications need orchestration during the whole execution, so the developer must ensure a correct parallel execution during the process lifetime, which requires additional modifications to the serial code.

The execution model of OpenMP applications is based on threads (also known as light-weight processes) that are single independent sequential flows of control within a program that have access to all the process data but also have a private data area. In this model, the application follows a fork-and-join behavior as illustrated in Figure 2.2. The application starts as a single process with one thread of execution (named master) and once it reaches a parallel construct (`#pragma omp parallel`, for instance), a team of threads is created within the process to execute the code within the parallel construct. Then, each thread of the team carries out its assigned work and once it finishes, the thread waits for the remaining threads of the team to finish, to

Table 2.2
Comparison of two versions of the same application that approximate the π value. On the left, the application uses MPI point-to-point operations, and on the right, the application uses MPI collective operations.

```
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
if (rank == 0)
    for (i = 1; i < size; i++)
        MPI_Send (&n, 1, MPI_INT, i, 1000, MPI_COMM_WORLD);
else
    MPI_Recv (&n, 1, MPI_INT, 0, 1000, MPI_COMM_WORLD, &s);
lb = 1 + rank*n/size;
ub = MIN(rank+1)*n/size, n);
h = 1.0 / n;
tmp2 = 0;
for (i = lb; i <= ub; i++)
{
    x = h * (i - 0.5);
    tmp2 += (4.0 / (1.0 + x*x));
}
if (rank == 0)
    for (i = 1; i < size; i++)
    {
        MPI_Recv (&tmp, 1, MPI_DOUBLE, 1, 1001,
            MPI_ANY_SOURCE, MPI_COMM_WORLD, &s);
        tmp2 += tmp;
    }
else
    MPI_Send (&tmp, 1, MPI_DOUBLE, 0, 1001, MPI_COMM_WORLD);
if (rank == 0)
    pi = h * tmp;
MPI_Finalize();
```

```
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);

MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

lb = 1 + rank*n/size;
ub = MIN((rank+1)*n/size,n);
h = 1.0 / n;
tmp2 = 0;
for (i = lb; i <= ub; i++)
{
   x = h * (i - 0.5);
   tmp2 += (4.0 / (1.0 + x*x));
}

MPI_Reduce (&tmp2, &tmp, 1, MPI_DOUBLE, MPI_SUM, 0,
  MPI_COMM_WORLD);

if (rank == 0)
   pi = h * tmp;
MPI_Finalize();
```

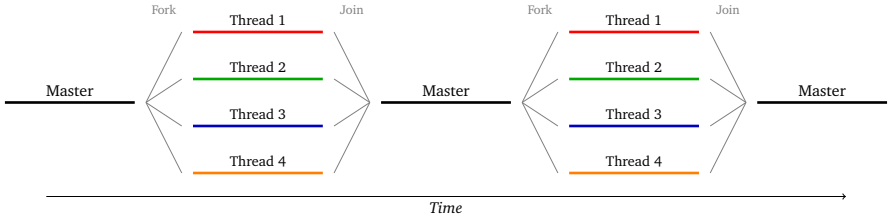


Figure 2.2
The fork-join execution model in OpenMP.

finally let the master thread continue the execution. While in the Figure threads are depicted in such a way that threads are created at every parallel region, it is very common that OpenMP implementations create the team of threads once and then reuse it in the next parallel construct in order to save the expense of the team creation. In fact, since every process has at least one thread (named master), the OpenMP run-time only creates three threads and let the master thread become one of the worker threads when entering into the parallel regions.

Up to OpenMP version 2.5, this programming model mainly focused on extracting parallelism from loops because of two reasons. First, these are the regions of code where applications invest most of their time and, second, because the loops easily offer parallelism opportunities if data dependencies between iterations are independent (i.e. the data produced from one iteration is not consumed in another iteration). Usually, the team of threads shares loop iterations (also known as work units) of a parallel loop through the use of work-sharing constructs. Each work unit, with the corresponding data environment, is bound to a particular thread for its whole lifetime, becoming impractical to execute loops with dynamic conditions. Since OpenMP version 3.0, this model allows expressing irregular parallelism through the tasking constructs, which allows executing in parallel loops where the number of iterations cannot be determined at the entry of the loop, and also, dynamically generating units of work.

The use of shared-memory paradigms imposes the developer to care about accesses to shared data because modifications done by a thread are visible to the remaining threads. Consider again the pi kernel shown in Listing 2.3 while being executed by multiple threads, if two threads simultaneously read the value of `area` (line 5) and then both update the value of this variable, one of the updates will be lost. OpenMP provides ways to synchronize threads, which include lock constructs (through the OpenMP API) as well as `critical`, `atomic`, `single`, `master` clauses. The lock constructs provide the most general approach to coordinate execution between threads because only one thread at a time executes the code that is wrapped by these constructs. `Critical` and `atomic` directives specify that the region delimited and the following sentence are expected to be carried out in mutual exclusion, respectively. The `single` clause indicates that the enclosed code is executed by one thread of the team, whereas the `master` clause specifies that the code is executed by the master thread, only. All these constructs do not come for free, though, as they penalize the application performance because they increase the serialization of the application if multiple threads face a synchronization region at the same time. Sometimes the source code allows transformations to avoid adding these synchronization points. For instance, as the addition is a commutative operation, each thread updates the `area` variable privately with its own partial value and then leave master thread to accumulate each thread's copy into the actual variable at the end of the parallel construct. This approach makes unnecessary to synchronize the threads when accessing the shared variable, but requires a commutative operation and a mechanism that allows the master thread to access the private copies of the remaining threads. This latter technique receives the name of reduction and it is available in OpenMP as an extension for the parallel construct by providing the target variable and the operation.

Table 2.3 illustrates the implementation of the application that calculates the π value using OpenMP in which the work is distributed among threads but the way to reach the solution differs. In the application on the left, the master thread starts executing and when it encounters the `#pragma omp parallel for` it spawns the team of threads to distribute the iteration trip count among them. In such a distribution, each thread accesses to a private copy of `x` and `i` variables

Table 2.3
Comparison of two OpenMP versions that approximate the π value. On the left, the application uses a synchronization mechanism that introduces serialization. On the right, the application uses a reduction clause to accumulate each thread's private copy of area at the end of the parallel region.

```
1  h = 1.0 / n;
2  #pragma omp parallel for \
3    private(x) shared(n,h,area)
4  for (i = 1; i <= n; i++)
5  {
6    x = h * (i - 0.5);
7  #pragma omp atomic
8    area += (4.0 / (1.0 + x*x));
9  }
10 pi = h * area;
```

```
1  h = 1.0 / n;
2  #pragma omp parallel for \
3    private(x) shared(n,h) \
4    reduction(+:area)
5  for (i = 1; i <= n; i++)
6  {
7    x = h * (i - 0.5);
8    area += (4.0 / (1.0 + x*x));
9  }
10 pi = h * area;
```

(because loop index is always privatized) and also to the shared variables `n`, `h` and `area`. Within the parallel region, each thread calculates the value for `x` and then updates the `area` variable exclusively (lines 6 to 8). When all threads finish, the master thread continues alone. In the code shown on the right, the application behaves mostly the same except for the parallel region which does not include synchronization constructs but uses a reduction clause (lines 2 to 4) that applies the add operation to each private copy of the variable `area`. Therefore, when the parallel region finishes, each thread has its own partial summation of `area` and then the master thread aggregates them all into master thread's variable `area`.

2.3.3 Instruction-level parallelism

The next system level of parallelism exposed is the processor-level because one single processor also executes instructions in parallel. In this direction, instruction-level parallelism (ILP) accounts for how many operations are performed simultaneously during the execution of an application. Current processors implement several techniques to extract parallelism from a sequential instruction stream, i.e. a processor is capable of overlapping the execution of multiple instructions at a time so producing more than a result per unit of time. Consider the code shown in Listing 2.4 that consists of three assembly instructions. In this sequence of instructions, the third instruction depends on the results of the first and second instructions but the second instruction does not depend on any other instruction, so the two first instructions can be calculated simultaneously. Assuming that several instructions can be executed in a unit of time and that instructions one and two are executed simultaneously, the code above would only require two units of time, which would represent an ILP of 3/2 instructions per cycle (IPC), or inversely 2/3 cycles per instruction (CPI).

Listing 2.4
Simple sequence of assembler instructions.

```
1  ADD R1, R2        ; R1 = R1 + R2
2  ADD R3, R4        ; R3 = R3 + R4
3  MUL R1, R3        ; R1 = R1 * R3
```

The amount of existing ILP of a workload depends on the application because it dictates the instruction dependencies and also, depends on the ability of the compiler to schedule the application instructions. Compilers use techniques such as loop unrolling and software pipelining techniques to increase the ILP, but still the hardware may have limited resources to execute the independent instructions. This section highlights the most prominent techniques to achieve a high ILP in hardware, which include pipelining, out-of-order and speculative execution, as well

		Clock cycles →								
PC	**Instruction**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
i+0	`LOAD R0, X[4]`	F	D	X	M	W				
i+1	`ADD R1, R2`		F	D	X	M	W			
i+2	`SUB R3, R4`			F	D	X	M	W		
i+3	`MUL R5, R6`				F	D	X	M	W	
i+4	`ADD R7, 1`					F	D	X	M	W

Figure 2.3
Example of diagram for a sequence of six instructions starting at a given Program Counter (PC) executed in a processor where instructions traverse five stages. The stages are **F**etch the following instruction, **D**ecode the instruction, e**X**ecute the instruction (or calculate the address of it the instruction references to memory), access the **M**emory and **W**rite (or commit) the results.

as superscalar processors. The main benefit of these approaches is that they are transparent to the programmer[1] so that parallelism is automatically extracted by the processor.

2.3.3.1 Pipelining

Pipelining is a technique that allows the overlapping of the execution of multiple instructions. This technique is achieved by dividing the work of the instructions into smaller parts so that instructions need several steps (or stages) to complete. The stages are physically connected, each one to the next, so that instructions enter and progress through the stages to finish when they exit the last stage. The number of steps in a pipeline depends on factors like the instruction set architecture (ISA) and the instruction semantics. The processor designer must to balance the work at each stage because the slower stage determine the duration of the machine cycle. The main benefit of breaking down instructions into smaller stages is that it allows a significant reduction in the machine cycle. Comparing the amount of time per instruction, it does not matter if the processor is pipelined, a single instruction requires the same amount of time to complete, but in a pipelined processor, the benefit comes from that the following instructions are finished in every cycle and as the cycle time is shorter the overall execution time is then shorter.

The diagram shown in Figure 2.3 exemplifies the instruction progression in a 5-staged pipelined processor. The diagram shows a time-line in which the X-axis represents cycles, the Y-axis represents the Program Counter register (PC) and the instructions as they enter in the pipeline and each instruction faces five instruction to complete. As the reader can observe, every cycle a new instruction enters the pipeline and after the fifth cycle an instruction finishes every cycle. Compared to a non-pipelined processor, the same instructions would require five cycles, however in this example it requires nine cycles, so pipe-lining the processor requires more cycles to complete the program. However, while the number of cycles required in a pipelined processor is higher, the shorter cycle time implies a reduction in the required time to complete the instructions. For instance, if the cycle time of the non-pipelined processor is T_{cyc} and the pipelined processor allows dividing the stages in a such way that the cycle time is $T_{cyc}/5$, then the total time required for executing the application will be $5 \times T_{cyc}$ and $9 \times T_{cyc}/5$, respectively and for a reduction in 64% of the execution time.

While the pipelined processor is meant to ideally finish an instruction every cycle, there are situations, known as hazards, that restrict a pipelined processor to achieve this rate. Hazards are classified as: structural, data, or control. Structural hazards exist when the hardware cannot execute the simultaneous instructions in the pipeline due to lack of resources. For instance, if the processor only has one arithmetic-logic unit (ALU), it becomes impossible to calculate two additions at a time. Data hazards arise when an instruction depends on the results of a previous instructions. There are three types of data hazards, true dependencies (an instruction generates a value in a register that a following instruction needs as an input), anti-dependencies (an instruction requires a register that is latter updated) and output dependencies (two instructions modify the same register). Violating dependencies between instructions breaks the semantics of

[1] While the application programmer can hint the hardware on the application behavior, these techniques are meant to be valid without requiring any modification from the programmer.

the code and thus yields to an incorrect execution. Finally, control hazards are related to the pipelined execution of instructions that modifies the PC. No matter the type of hazard, when the processor detects them it needs to stall the execution of incoming instructions and continue the execution of the instruction within the pipeline until the hazard disappears by generating *p*ipeline bubbles (represented as ∘). While structural hazards can be avoided by replicating functional units within the processor, the rest cannot be avoided directly by the hardware, though there are opportunities for reducing the impact of data and control hazards. The following sections discuss techniques used for limiting the impact of these hazards in modern processors.

2.3.3.2 Out-of-order execution

So that instructions can be actually executed in the pipeline, it is imperative that the operands are ready once the instruction faces the execution stage (X). If operands are not ready (because data are brought from memory or it is computed in a multi-cycle functional unit, for instance), the instruction has to wait (or stall) inside the pipeline until operands are available, so reducing the ILP. Although in the exemplified pipeline instructions write the results back in the write stage (W), since the results are actually computed at earlier stages, there is a chance for the processor to forward (or bypass) the values from latter stages to the input of earlier stages of forthcoming instructions. Forwarding results from one stage of one instruction to another stage of a following instruction simply requires additional logic and does not require adding bubbles into the pipeline. Consider the time-lines in Figure 2.4 which represent the execution of the same code with two data dependencies but executed on a machine that cannot (top) and can (bottom) forward the result as soon as it is generated at the execution stage (X). The first dependence exist between instructions at PC i+0 to PC i+1. If the processor cannot forward the result immediately after finishing the execution stage (X), then it is necessary to stall the processor until the data are being written (as seen in Figure 2.4a). However, if the processor forwards the result of execution stage (X) of instruction at PC i+0 to the beginning of the execution stage (X) at PC i+1, at cycles three and four, then there is no need to stall the processor as seen in Figure 2.4b. The second data dependence involves instructions at PC i+2 and PC i+3. In this dependence, the instruction at PC i+2 does not get the data from memory until the memory stage (M) has finished for instruction i+2, which occurs at cycle six in Figure 2.4b. However, instruction i+3 requires that data at the execution stage (X), which would be cycle five. Due to the design of the pipeline, the processor has to stall the execution of the forthcoming instructions by injecting a bubble into the pipeline at cycle five. In fact, it is even possible that the memory needs more than one cycle to provide this data if the memory hierarchy requires more cycles to access these data. So when a cache miss occurs, more bubbles are injected in the pipeline according to the time wait for the data and consequently, reduce the achieved ILP.

Thornton proposed a method named Scoreboarding [217] for the CDC 6600 computer that uses an extra logic (named scoreboard) to orchestrate instructions during their live within the pipeline and executes instructions out of order if sufficient resources are available. The scoreboard is fully responsible for fetching instructions, executing them and for detecting all possible hazards between them. It records all the dependencies of the instructions in the pipeline and then determines when instructions can read their operands and commence the execution. If the scoreboard determines that an instruction cannot be executed immediately, it monitors all the modifications in the register file so it determines when the instruction will be executed. Yet an instruction is stalled in the pipeline, the scoreboard continues fetching instructions as long as the processor has enough functional units to execute the instruction and the target register does not match any instruction inside the scoreboard that writes on the same register (to avoid violating output dependencies).

Tomasulo also proposed an algorithm to allow proceeding with the execution of instructions even if there are hazards between some instructions in the IBM System/360 model 91 [218]. This approach combines some elements from Scoreboarding, but also introduces a key feature such as the register renaming. Register renaming avoids anti-dependencies and output dependencies data hazards by transparently extending the number of available physical registers (named also as reservation stations) compared to those available in the ISA. In addition to register renaming, Tomasulo's algorithm introduces a distributed hazard detection and a common bus to transfer

		Clock cycles →									
PC	**Instruction**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
i+0	`ADD R1, R0`	F	D	X	M	W					
i+1	`ADD R1, R2`		F	D	∘	X	M	W			
i+2	`LOAD R3, X[0]`			F	∘	D	X	M	W		
i+3	`ADD R1, R3`				∘	F	D	∘	X	M	W

(a) Needs to wait until instruction completes.

		Clock cycles →									
PC	**Instruction**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
i+0	`ADD R1, R0`	F	D	X	M	W					
i+1	`ADD R1, R2`		F	D	X	M	W				
i+2	`LOAD R3, X[0]`			F	D	X	M	W			
i+3	`ADD R1, R3`				F	D	∘	X	M	W	

(b) Forwards instruction results as soon as they are generated.

Figure 2.4
Comparison of pipeline executions with dependencies, either from registers or memory without (top) and with (bottom) result forwarding. The processor needs to wait for such data introducing a bubble (∘) in the pipeline thus stalling forthcoming instructions.

results directly from functional units to reservation stations. With respect to the execution flow, the processor fetches instructions into an out-of-order buffer and sends the operands to a reservation station if operands are in registers. Once instructions reach the out-of-order buffer, they are executed as soon as the operands are ready and there are functional units to execute them. If operands are not ready, the algorithm observes the bus monitoring for the registers needed by the instruction. Once the instruction result is available, it is transmitted through the bus to the registers and to any reservation station waiting for it.

2.3.3.3 Speculative execution

As mentioned above, control flow instructions arise control hazards because they do not only depend on the results of previous instructions but are able to change the execution flow or the instruction stream that the processor is executing. When executing in a pipelined processor, the evaluation of control instructions and the value of the forthcoming PC happen at advanced stages of the pipeline, which implies that the processor has to wait for the result that determines which instruction stream to follow. In such control hazards, the number of bubbles added into the pipeline is determined by the number of stages (or cycles) that the control instruction has to traverse to determine whether the control flow changes, so the sooner it is resolved, the higher ILP achieved.

Consider the diagram shown in Figure 2.5 which contains a branch instruction at PC i+2. Here two situations may occur when reaching PC i+2, if the condition is met then the PC is set to i and the instruction flow has to be restarted from there, or if the condition is not met then the instruction flow continues at i+3. Independently from this, a non speculative processor fetches at the beginning of cycle four the following instruction pointed by the PC because, at that point, the processor has not yet decoded that the instruction at i+2 is a conditional branch. At cycle five, the processor has decoded the branch instruction and it has to stall subsequent instructions from entering the pipeline until it calculates whether the branch has to be taken or not (at stage X in this example). If the condition is met (not shown in the diagram), then the PC has to be set to i+0 and restart the execution from there after discarding the instruction that entered in cycle four, which means that the processor has to stall at least for two cycles (decode and execute stages). If the condition is not met (shown in the diagram), then the PC has to be set to i+3, which means that the processor has to stall for one cycle.

The branch predictor is a component of the processor that speculate whether the control flow will change and the address of the following instruction stream, so allowing issuing additional instructions into the pipeline without stalling the processor. As current processors

		Clock cycles →								
PC	**Instruction**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
i+0	`ADD R1, R2`	F	D	X	M	W				
i+1	`SUB R3, 1`		F	D	X	M	W			
i+2	`JNZ` i+0			F	D	X	M	W		
i+3	`ADD R2, 1`				F	∘	D	X	M	W

Figure 2.5
Pipelined execution of a sequence of instructions including a branch instruction. The evaluation of the following PC is unknown until later stages of the instruction, so the processor has to wait, injecting a bubble (∘), until its evaluation to determine which is the next instruction to execute.

have deep pipelines nowadays, it is important to have such a correct prediction as soon as possible in order prevent penalization of the performance. Branch predictors are based on either a statically fixed prediction [68] or the history results of the control instruction that entered in the pipeline [125, 234]. Control flow instructions are ultimately evaluated, which allows confirming if the speculation of the branch predictor was correct. If branch predictor hits on the result, the instructions fetched during the evaluation (speculative) are executed normally; however, if the branch predictor missed, then the instructions that entered speculatively into the pipeline need to be flushed and the control flow needs to be changed accordingly, with the consequent penalty.

2.3.3.4 Superscalar and VLIW processors

The techniques from the previous sections are used to reduce the stalled cycles in the pipeline and so increase the chances to achieve an ILP of 1 IPC. An ILP of 1 IPC is the maximum reachable value if only one instruction enters in the pipeline at a time, so to increase the ILP it is mandatory to fetch multiple instructions in the pipeline. There are two types of processors that fetch several instructions to the pipeline in a single cycle: superscalar and very long instruction word (VLIW) processors. Superscalar processors fetch a varying number of instructions into the pipeline that are either scheduled by techniques such as Scoreboarding and Tomasulo's algorithm and sometimes assisted by the compiler. VLIW processors, in contrast, fetch a fixed number of instructions typically formatted as either one large instruction or as a fixed set of independent instructions. In terms of architecture, VLIW processors are simpler to design because the instructions themselves expose the parallelism among them, i.e. if they are in the same set, they are meant to be executed at the same time. This means that for VLIW processors, the compiler is responsible for explicitly stating which instructions are executed in parallel, whereas in superscalar processors this decision is made at run-time, dynamically. Whatever the processor type, to effectively execute the fetched instructions the processor requires the replication of processor block to execute the instructions. For instance, several instructions may execute at the same time, many arithmetic-logic units may be necessary. Also, many instructions may read from the bank of registers at a time, requiring additional number of bank ports to read and write from it. Still, there is a fundamental and simple limit on available ILP: the code to be executed. If the code to be executed does not expose enough instructions to fill the available slots, then does not matter the how many instructions are sent to the pipeline.

2.3.4 Vector processors

Vector processors provide an additional way to achieve parallelism through the architecture. These processors provide high-level instructions that work on vector operands instead of scalar operands. Compared to the previously seen architectural techniques that achieve instruction parallelism from a serial code, one vector instruction stipulate a bunch of work. Vector-processors provided several benefits when they appeared. First, a single vector instruction specifies such an amount of work that sometimes a single vector instruction replace a whole loop from the application source code. In this direction, there is a reduction on the number of control instructions executed (mainly branches and instructions to calculate the next iteration index) and, the consequent savings on the pressure on the instruction bandwidth. Second, vector

instructions access memory with a known access pattern, so in systems where memory banks are interleaved, the high latency of accessing the main memory can be amortized because accesses are initiated for the whole vector and not for every single scalar. The last advantage, is that as vector instructions involve multiple parallel and independent operations, these operations are performed in parallel if several functional units can do the operation. There are two types of vector processors, depending whether the operands are located in memory (the so-called memory-memory vector processors) or in registers (named vector-register processors). The discussion in this section focuses on vector-register processors, although a similar discussion can be extrapolated for memory-memory vector processors just considering that operands are stored in memory instead of registers.

To fulfill these design objectives, vector-register processors require the following components: vector registers, vector-control registers, functional units, and load-store units as well as a set of scalar registers. Vector registers are fixed-length registers that hold multiple elements of basic data types (such as integer and floating-point values). With respect to the vector-control registers, the Vector-Length and Vector-Stride registers are highlighted. As data vectors in applications do not need to match the length of the registers, there must exist a Vector Length register in the processor ISA that tells the processor the number of useful elements present in the vector registers. Also, positions in memory of adjacent elements of a vector may not refer to consecutive addresses, so there is a Vector Stride register indicates the distance in memory addresses between adjacent elements when executing vector load or store instructions. Finally, scalar registers are needed as in regular processors, to provide data as input to the vector functional units and to calculate addresses for the vector load-store unit.

Listing 2.5
Source code for the SAXPY/DAXPY routine.

```
1 for (i = 0; i < N; i++)
2   Y[i] = a*X[i]+Y[i]
```

To exemplify the behavior of a vector processor, consider the piece of code shown in Listing 2.5 which calculates $Y=a\times X+Y$ (being a scalar and X and Y vectors) and which is commonly known as SAXPY or DAXPY, depending on whether the values are single or double-precision, respectively. When the above code is compiled for a scalar processor, the assembly code may look like the code found in the left column of Table 2.4. On the scalar version, each index is calculated individually (instructions 4-8) and then the loop index is advanced and repeats the loop if necessary (instructions 9-11). To execute the code, the application needs to traverse the loop N times, which in this case accounts for a total of $2+8\times N$. In contrast, in a situation where the vector processor supports up to 64 elements on its registers, the number of instructions to be executed is reduced to $3+(11\times N)/64$. That value can be further shortened if the compiler knows at compile-time whether N is 64 or less, or even if N is multiple of 64. In these circumstances, the compiler might suppress unnecessary control instructions.

Giving the vectorization example of the DAXPY routine helps us to understand not only how a vector processor works, but also why some of the Top500 systems pointed out in Table 1.1 (on page 3) achieve such a good efficiency. The DAXPY routine represents a small fraction of the source code of the Linpack benchmark but accounts for most of its execution time, which means that supercomputers built on top of vector processors, such as Earth Simulator and K computer systems, achieve a higher Flop/s rate than systems not using vector processors.

During the last years, processors have benefited from architectural techniques that increased the performance significantly and have left the original vector processors schemes behind. The introduction of architectural features such as pipelining, superscalar and out-of-order execution, which combined with efficient memory hierarchies built on top of several levels of caches and the flexibility of combining processors into clusters made these systems more competitive in terms of performance/price ratio. Today, however, most of the current processors are vector-capable due to the introduction of the originally self-styled multimedia instructions. These multimedia

Table 2.4
Comparison of scalar and vector codes for DAXPY using a pseudo-assembly language. On the left there is a version for serial processors, whereas on the right there is a version for vector processors. The vector code version assumes that vectors are 64 elements long.

```
                                    LOAD  Ra, a[]  ; Ra = a
1     LOAD Ra, a[]      ; R1 = a    MOVE  Rb, N    ; Rb = N
2     MOVE Ri, 0        ; i = 0     MOVE  VL, 64   ; Init VL = 64
3  LOOP:                          LOOP:
                                    CMP Rb, 64     ; Is Rb < 64?
                                    JGE USEVL64    ; If not, use VL = 64
                                    MOVE VL, Rb    ; Else: last iteration
                                                   ; use VL = Rb
                                 USEVL64:
4     LOAD Rx, X[8*Ri]  ; Rx = X[i]  LOADV Vx, X[]  ; Vx = X[1:VL]
5     LOAD Ry, Y[8*Ri]  ; Ry = Y[i]  LOADV Vy, Y[]  ; Vy = Y[1:VL]
6     MUL Rx, Ra        ; Rx = a*Rx  MULSV Vx, Ra   ; Vx = a*Vx
7     ADD Ry, Rx        ; Ry = Ry+a*Rx ADDV  Vy, Vx ; Vy = Vx+Vy
8     STORE Y[8*Ri], Ry ; Y[i] = Ry  STOREV Y[], Vy ; Y[1:VL] = Vy
9     ADD Ri, 1         ; i = i+1    SUB Rb, 64     ; VL = VL-64
10    CMP Ri, N         ; i < N ?    CMP Rb, 0      ; VL > 0?
11    JNE LOOP          ; Continue?  JG LOOP        ; If so, continue
```

instructions are almost the same as vector instructions, but with a vector-length far more reduced. For instance, the Cray Y-MP had eight vector registers that held 64 64-bit elements [41, 40], whereas the Intel MMX instruction set has eight vector registers holding two 32-bit integer-only elements [102] and the Altivec instruction set has 32 vector registers holding up to four 32-bit elements each [185]. Intel has continued developing their multimedia instructions and their forthcoming instruction set is named *AVX512*. This instruction set contains up to 32 vector registers that accommodate eight 64-bit elements [103], which considering the number of bits in the vector register bank is half of the number of bits compared of the vector register of a Cray Y-MP, showing that there is still a road ahead of vector processing.

2.3.5 Coprocessors and accelerators

This chapter concludes with an illustration of the use of coprocessors to achieve a higher degree of instruction execution rate. Coprocessors are, by definition, devices that extend (or complement) the functions of the processor. There is a variety of coprocessors that perform different types of operations such as floating-point arithmetic, signal and graphics processing and I/O interfacing. Coprocessors not only allow executing a portion of the application, but they run independently from the processor so the application advances in parallel, executing one part in the processor and another in the coprocessor. The fact that some of these coprocessors, such as graphic processing units (GPUs), have arisen as devices capable of running software in parallel to the processor faster than the processor itself, has resulted in naming them as accelerators.

Today's well-known accelerators include the IBM PowerCell [36], the Intel Xeon Phi [109] and the NVIDIA CUDA [162]. Each accelerator provides its own specific development platform proposed by its vendor but there are some efforts to unify and simplify the development of applications that use these accelerators. For instance, the most dominant programming models for GPUs are CUDA and OpenCL [114] but programming models such as HMPP [48], OpenACC [165] simply extend programming languages to delegate work to GPUs in an easy manner. Other approaches, such as OmpSs [56] and OpenMP 4.0 [166] not only support GPUs as well as other accelerators such as the Intel Xeon Phi, but also allow parallelizing for the shared-memory multiprocessor and the use of accelerators in the same application.

Table 2.5 depicts the combined use of shared-memory parallelism and accelerator using the OmpSs programming model. So as OmpSs leverage work to the accelerator, the routine meant to be executed in the accelerator needs to be written in the native language for the accelerator. In this example, the code on the left shows the actual implementation of the kernel that is intended to be executed in the accelerator using the CL language in which objects x and y are globally

Table 2.5
An hybrid application that uses OpenCL acceleration plus OmpSs shared-memory parallel programming. The code for the accelerator is written in CL language on the left. On the right, the code executed on the host and decorated with OmpSs directives to take benefit of the accelerated version of DAXPY.

```
1  __kernel void daxpy (
2    int n,
3    double a,
4    __global double *x,
5    __global double *y)
6  {
7    int i = get_global_id (0);
8    if (i < n)
9      y[i] = a*x[i] + y[i]
10 }
```

```
1  #pragma target device(opencl) \
2    ndrange (1, n, 128) copy_deps
3  #pragma omp task \
4    in([n]x) inout ([n]y)
5  __kernel void daxpy (
6    int n,
7    double a,
8    __global double *x,
9    __global double *y);
10
11 int main (int argc, char *argv[])
12 {
13 #define N 1024
14   double x[N], y[N], z[N];
15   double r[N][N], s[N][N], res[N][N];
16
17   daxpy (N, a1, x, y);
18   dgemm ('N', 'N', N, N, N, 1.0,
19     r, N, s, N, 1.0, t, N);
20
21 #pragma omp taskwait
22
23   dgemv ('N', 'N', 1.0, y, N, t,
24     N, 1, 1.0, res, 1);
25
26   return 0;
27 }
```

accessed. The acceleration code is executed in such a manner that each iteration is executed by a different thread and each thread identifies which iteration index to refer using the call `get_global_id(0)`. The code on the right shows the code for the main processor written in C with some OmpSs directives. The code shows first a declaration of the routine that serves as an execution point for the `daxpy` routine in the accelerator and follows the code to be executed in the processor. The declaration to the routine for the accelerator code is subject to two directives. The first directive indicates that the following declaration of `daxpy` is implemented as an OpenCL kernel. The declaration specifies uni-dimensional data array of length `n` that is being executed by 128 threads (or more accurately to the OpenCL nomenclature, work-items) and that ensures that a copy of the data is within the accelerator address space when the routine is about to run. The second directive instructs the OmpSs run-time how data is accessed in the code, so that the OmpSs run-time order of execution of this routine according to the data dependencies. In this example, vectors `x` and `y` are input variables and `y` is an output variable, as well. In the main code, a call to `daxpy` (line 15) occurs, so the OmpSs run-time copies the input parameters of the routine and then lets the accelerator run the `daxpy` routine asynchronously while the processor continues and executes the matrix by matrix multiplication (routine `dgemm` in line 16). The processor finishes the `dgemm` call and then executes the `#pragma omp taskwait` to wait for the completion of the `daxpy` task and brings the results in vector `y` back from the accelerator. The execution flow continues in the processor, which calculates the matrix by vector multiplication invoking the `dgemv` routine and then finishes.

3 Performance analysis

> `There are three things all wise men fear: the sea in storm, a night with no moon, and the anger of a gentle man.`
>
> — Patrick Rothfuss, THE WISE MAN'S FEAR

As computational power of supercomputers increases in every generation, developers need to keep pace with technology and adapt their applications to get most of systems through the hardware and software stack. These efforts lead to a better use of resources and benefit end users with a shorter time-to-solution, allowing them to focus more on research instead of waiting for results. By all means, developers can take the logical, but innocent, approach to measure the duration of the application execution after applying one modification at a time. However, if developers put these efforts into use without judgment then the optimization process is likely to result in a daunting task. Experience shows that many applications output additional timing results to report the time consumption of several parts of the application. However, these results do not answer the fundamental questions such as: what is the nature of the performance bottlenecks? or, where are such inefficiencies within the application code?

3.1 Description

Performance analysis is a multidisciplinary subject that has been adopted in system and software development and which involves measurement capabilities, simulation and system modeling. The importance of performance analysis is threefold:

1. There may be performance bugs as there are logical bugs, so if performance is important it should be debugged by measurement.
2. When designing new or improved systems or programs, a good understanding of the performance of the base system is needed for avoiding future performance bugs.
3. To avoid spending resources on fixing obvious, but minor, inefficiencies rather than searching the real reasons for the poor performance.

Performance analysis tools are pieces of software that help to measure and deliver comprehensive details of the application behavior on a given system. The details provided include a

number of metrics associated with the application structure and the underlying execution. A performance tool helps their users (hereafter, analysts) to understand the unknown behavior of the application. In addition, these tools compare the collected measurements against a theory, model, or even previous executions and ultimately help with fixing performance bugs in the application that they consider worth mending.

When it comes to analyzing the application performance on a system, the analyst may explore the performance from the application or the system point of view. This distinction begets two different types of analyses as one part can be changed while the other remains immutable during the experimentation in order to achieve proper comparative analyses. On the one hand, it is possible to consider the software as a fixed part and so explore the processor design space for the purposes of improving forthcoming processor generations by applying architectural simulations. On the other hand, the software developer (or the analyst) may need to tweak the application somehow to take the maximum profit of an immutable system. The scope of this thesis focuses on the latter approach in which developers execute their application on a system and need to adapt the software to the underlying processor.

With respect to the performance tools, there are different aspects that help to classify them. The first classification scheme depends on how data are stored and presented to the user. Performance tools either store time-stamped metrics begetting a time-series stored in a trace-file, or summarize all the measurements with the consequent space savings, but loosing time-dependent issues. A secondary way to classify performance tools refers to the mechanism used for executing the monitors (or probes): instrumentation and sampling. Instrumentation refers to the ability to inject monitors to specific application locations therefore it provides accurate metrics to these regions of code. On the other hand, sampling takes advantage of mechanisms to periodically invoke monitors so that the results are statistical inferences of the application behavior. While some of the tools expose the captured metrics with minimal or none manipulation, there are tools that process the metrics by applying additional mechanisms and extract conclusions automatically without major intervention from the analyst.

This chapter covers most of the topics relating to performance analysis tools. First, there will be an extended summary of a variety of state of the art tools, discussing their design points and presenting the reports they provide to the analyst. The summary of the tools will also includes the investigation of many techniques to evaluate metrics automatically to ease the analyst's experience. A section will then follow with a detailed dissertation of the mechanisms available to record the application measurements. Afterwards, there will be a brief discussion about applied techniques to correlate performance with the application source code. Finally, the chapter will also cover the node-level metrics that help understand what the exact processor-related performance issues exist in the application.

3.2 Overview of performance analysis tools

Performance analysis tools need to answer two major questions that analysts face when studying an application: what are nature of the performance inefficiencies and where are they located within the application? Performance tools describe the application performance behavior using either summaries (profiles) or detailing variations in performance across time (trace-files). Profiles apply first order statistics (such as `mean`, `min`, `max`, `stdev`) to the measurements captured to simplify the metrics provided to the analyst. Tools that use trace-files allow expressing variability of performance issues exposing sequence of metrics as well as multi-modal behavior.

3.2.1 Profile based tools

gprof [88] is the *de facto* profile-based tool because it is available by default on several operating systems and widespread across many compilers. The profiler works in conjunction with a compiler (typically using the special flag `-pg`) so that the compiler instruments user routines to count the number of invocations and to emit the relationship between routines (in terms of caller-callee) in a call-graph. In addition, during the application execution, the profiler also uses sampling mechanisms to estimate the execution time per routine. Listing 3.1 shows an

Listing 3.1
`gprof`'s flat profile.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 29.83     18.33    18.33 146029716     0.00     0.00  binvcrhs_
 17.38     29.00    10.68 146029716     0.00     0.00  matmul_sub_
 12.81     36.88     7.87       201     0.04     0.09  z_solve_
 12.79     44.73     7.86       201     0.04     0.09  y_solve_
 12.32     52.30     7.57       201     0.04     0.09  x_solve_
  7.96     57.19     4.89       202     0.02     0.02  compute_rhs_
  5.15     60.36     3.17 146029716     0.00     0.00  matvec_sub_
  0.60     60.73     0.37       201     0.00     0.00  add_
  0.50     61.04     0.31   2317932     0.00     0.00  lhsinit_
  0.29     61.22     0.18   2317932     0.00     0.00  binvrhs_
```

Listing 3.2
`gprof`'s call-graph profile focusing on the routine `adi_`.

```
index % time    self  children    called     name
                0.00   61.19     201/201         MAIN__ [1]
[3]     99.6    0.00   61.19     201         adi_ [3]
                7.87   10.89     201/201         z_solve_ [4]
                7.86   10.89     201/201         y_solve_ [5]
                7.57   10.89     201/201         x_solve_ [6]
                4.87    0.00     201/202         compute_rhs_ [9]
                0.37    0.00     201/201         add_ [11]
```

output of gprof when applied to the bt.A benchmark from the NAS benchmark suite [10] on an Intel®Core™i7 running at 2.4 GHz and compiled with the GNU compiler suite 4.8.1. The output is a sorted list of routines where each has associated set of performance metrics such as the cumulative seconds and the number of calls.

As mentioned, gprof reports a call-graph output as shown in Listing 3.2. The call-graph provides information regarding the routine invocations besides the callee and the caller routines. For instance, in the given example, the routine `adi_` is invoked by `MAIN__` and calls `x_solve`, `y_solve`, `z_solve`, `compute_rhs` and `add_`. The output also shows the exclusive time spent on each of the children calls and a relation with respect to the number of invocations. The combination of both outputs helps an analyst to understand the organization of an application in terms of routines, offering an approximation of the routines are the most time-consuming as well as which is their duration per invocation ratio. There are tools such as IBM's Xprofiler [121] that interpret the results of gprof and displays the output in a graphical user interface (GUI) so that each routine is represented in a box, as in Figure 3.1. In Xprofiler, the size and shape of each function indicate its CPU use. The height of each function box symbolizes its execution time (i.e. exclusive time), whereas the width of each function box illustrates the amount of time it has spent in addition to its descendants (*i.e.* inclusive time).

There are other profilers such as perf [144, 111] and OProfile [110] that extend the capabilities from gprof and attribute values of several processor related activities monitored through the Performance Monitoring Unit (PMU). The PMU is a hardware component available in modern processors that observes the activity of the processor and counts a set of events (commonly referred as *hardware performance counters*), and allows third party software to retrieve their value. This activity covers a wide range of events specific to the architecture, as the number of instructions fetched, instructions finalized, cache misses, branch mispredictions, or cycles stalled, to name a few. For instance, Listing 3.3 shows the attribution of three performance counters (namely instructions, cycles and L1 data-cache misses) sorted by number of events when applied to the bt.A benchmark using the perf tool. Notice that the benchmark shows a similar routine distribution for both instructions and cycles counters, although the actual attribution presents some variations, but routine distribution changes significantly when correlating with L1 data-cache

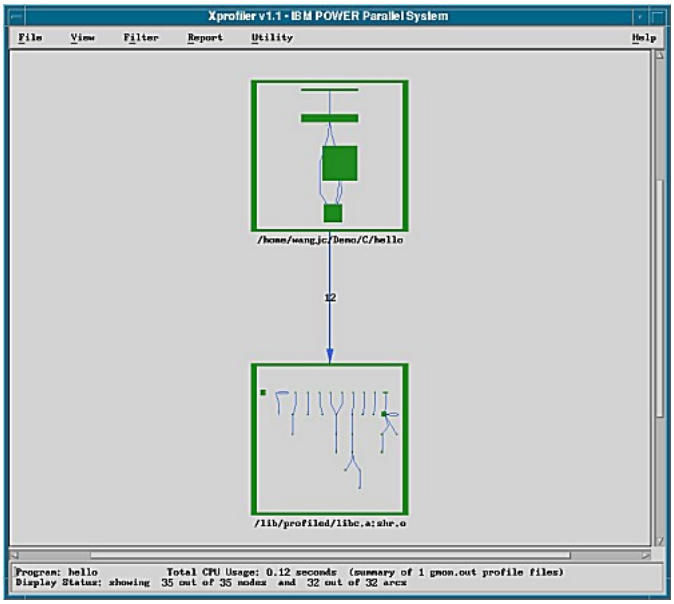


Figure 3.1
Example of results provided by the Xprofiler performance tool (picture obtained from [101]).

misses. The Listing indicates that the routine `binvcrhs_` is the most observed routine when the sampling uses instructions or cycles and the tool attributes most of the event counts (33.07% and 29.83%, respectively) to this routine. In contrast, the analysis of the application when sampled using cache-misses indicate that most of the misses (up to 33.87%) occur in `compute_rhs_`. These results point out that while the routine `compute_rhs_` experiences more cache misses, the most time-consuming routine is `binvcrhs_` because it executes more instructions. These profilers also report detailed attribution of performance counters to the assembly code (and to the source code if debugging information is available). Listing 3.4 shows the attribution of amount of time spent on each assembly instruction.

OmpP [75] is a profiling tool for OpenMP applications and differs from profiling tools such as gprof, perf and OProfile in two ways. First, ompP only relies on instrumentation and does not use sampling; so ompP enables direct measurement of program execution events. Second, data collection and representation follows the OpenMP user model of execution. In this sense, the results not only include the execution time, but also list the time to enter and exit OpenMP regions, including critical constructs, for example. In addition, the results include the accumulated time that each thread spends inside the construct and the number of times each thread enters the construct. This profiler is also capable of breaking down the execution time into overhead classes (synchronization, load imbalance, thread management and limited parallelism).

mpiP [227] provides a lightweight profiling tool for MPI applications by collecting statistical information. This profiler developed techniques that focus the user's attention on communication operations, so it detects what message-passing routines may limit the application scalability. Although this tool provides a good overview of the performance problems, it lacks the temporal order of data needed for in-depth performance analysis. IPM [3] is another performance tool for MPI binaries that provides temporal ordering in performance data using event flow graphs. The use of event flow graphs allows capturing MPI events and their temporal order as in trace-files while storing it in files that are significantly smaller.

Scalatrace [159] provides online trace compression of MPI communication trace-files at two different levels: intra-node and inter-node. Intra-node compression is achieved by describing loops with regular section analyses (RSA) [90] and compressing call-path information. On the other hand, Scalatrace compresses at inter-node level by combining the information from the

Listing 3.3
`perf`'s flat profiles showing the most representative routines according to the number of instructions executed, the number of cycles and the number of L1 data cache misses.

```
# Samples: 245K of event 'instructions'
# Event count (approx.): 348581349586
# Overhead  Command      Shared Object                  Symbol
# ........  .......  .................  ......................
    33.07%     bt.A  bt.A               [.] binvcrhs_
    20.55%     bt.A  bt.A               [.] matmul_sub_
    11.25%     bt.A  bt.A               [.] y_solve_
    11.06%     bt.A  bt.A               [.] z_solve_
    10.77%     bt.A  bt.A               [.] x_solve_
     7.44%     bt.A  bt.A               [.] compute_rhs_
     4.50%     bt.A  bt.A               [.] matvec_sub_
     0.59%     bt.A  bt.A               [.] add_
     0.21%     bt.A  bt.A               [.] exact_solution_
     0.21%     bt.A  bt.A               [.] lhsinit_

# Samples: 246K of event 'cycles'
# Event count (approx.): 145846400668
# Overhead  Command      Shared Object                  Symbol
# ........  .......  .................  ......................
    29.83%     bt.A  bt.A               [.] binvcrhs_
    16.45%     bt.A  bt.A               [.] matmul_sub_
    12.85%     bt.A  bt.A               [.] z_solve_
    12.73%     bt.A  bt.A               [.] y_solve_
    12.44%     bt.A  bt.A               [.] x_solve_
     8.13%     bt.A  bt.A               [.] compute_rhs_
     5.68%     bt.A  bt.A               [.] matvec_sub_
     0.62%     bt.A  bt.A               [.] add_
     0.50%     bt.A  bt.A               [.] lhsinit_
     0.25%     bt.A  bt.A               [.] exact_solution_

# Samples: 128K of event 'cache-misses'
# Event count (approx.): 71881607
# Overhead  Command      Shared Object                  Symbol
# ........  .......  .................  ......................
    33.87%     bt.A  bt.A               [.] compute_rhs_
    25.71%     bt.A  bt.A               [.] z_solve_
    18.71%     bt.A  bt.A               [.] binvcrhs_
     5.82%     bt.A  bt.A               [.] y_solve_
     5.81%     bt.A  bt.A               [.] add_
     4.71%     bt.A  bt.A               [.] matmul_sub_
     3.02%     bt.A  [kernel.kallsyms]  [k] 0xffffffff8104d24a
     1.59%     bt.A  bt.A               [.] matvec_sub_
     0.32%     bt.A  bt.A               [.] exact_rhs_
     0.30%     bt.A  bt.A               [.] binvrhs_
```

involved processes into a single one for the whole application.

Other profilers like TAU [194] capture information from parallel programming models (including hybrid MPI and OpenMP, for instance) and display the whole gathered in a simple GUI, rather than analyzing each process individually, as it happens in the previous profilers. Reporting information from several processes at the same time helps to identify performance variations between them. To illustrate a classical TAU report, Figure 3.2 shows a subset of 24 out of 96 processes from a CGPOP mini-application [204, 203] when executed in MareNostrum3 supercomputer [138]. In the Figure, the tool reports that the sixteen first processes (named node 0 to 15) spend more time in the orange and cyan routines compared to processes 16 to 27, but lesser time in the green and purple routines.

TAU has added sampling capabilities in its tracing measurement system [154]. In their combined approach, TAU also uses hardware counter overflows to gather performance information periodically. Although they collect performance counter information, their work is mainly focused on instrumentation cooperating with the sampling to augment the TAU profiles with call-stack

Listing 3.4
`perf`'s flat profile.

```
 Percent |  Source code & Disassembly of bt.A
------------------------------------------------
         :                  lhs(1,1,bb,k) = 1.0d+00
         :       >               + tmp1 * 2.0d+00 * njac(1,1,k)
         :       >               + tmp1 * 2.0d+00 * dz1
    0.39 :    40b762:       movapd %xmm2,%xmm9
    0.00 :    40b767:       mulsd  0x3390(%rax),%xmm9
    0.00 :    40b770:       addsd  %xmm3,%xmm9
    0.02 :    40b775:       addsd  %xmm14,%xmm9
    0.35 :    40b77a:       movsd  %xmm9,0x68b0(%rdx)
         :                  lhs(1,2,bb,k) = tmp1 * 2.0d+00 * njac(1,2,k)
    0.21 :    40b783:       movapd %xmm2,%xmm9
    0.00 :    40b788:       mulsd  0x33b8(%rax),%xmm9
    0.00 :    40b791:       movsd  %xmm9,0x68d8(%rdx)
         :                  lhs(1,3,bb,k) = tmp1 * 2.0d+00 * njac(1,3,k)
    0.23 :    40b79a:       movapd %xmm2,%xmm9
    0.09 :    40b79f:       mulsd  0x33e0(%rax),%xmm9
    0.00 :    40b7a8:       movsd  %xmm9,0x6900(%rdx)
         :                  lhs(1,4,bb,k) = tmp1 * 2.0d+00 * njac(1,4,k)
    0.00 :    40b7b1:       movapd %xmm2,%xmm9
    0.15 :    40b7b6:       mulsd  0x3408(%rax),%xmm9
    0.14 :    40b7bf:       movsd  %xmm9,0x6928(%rdx)
         :                  lhs(1,5,bb,k) = tmp1 * 2.0d+00 * njac(1,5,k)
    0.00 :    40b7c8:       movapd %xmm2,%xmm9
    0.00 :    40b7cd:       mulsd  0x3430(%rax),%xmm9
    0.23 :    40b7d6:       movsd  %xmm9,0x6950(%rdx)
```

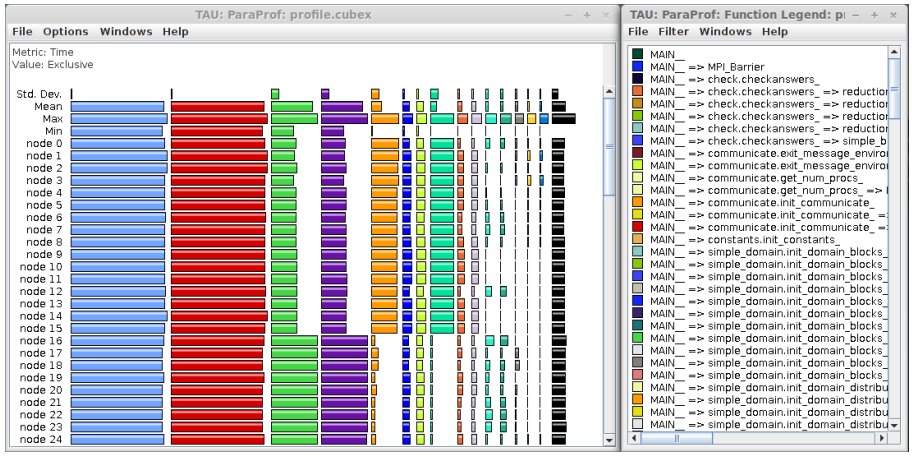


Figure 3.2
Example of results provided by the TAU performance tool.

information.

The Scalasca tool-suite includes the Cube visualization tool that summarizes the information gathered from the tool-suite instrumentation package. The Cube display shows three panels containing tree browsers, each of them representing a dimension of the performance space. The left tree presents the metric dimension (such as time, number of visits or occurrences and performance counters). The panel in the middle exhibits the (source code) program dimension where the tool user finds references to the captured routines. The tree on the right displays the system dimension, so that performance is inspected according to the resources involved during the execution. Every node has an associated value (named *severity*) and the value

is displayed simultaneously using a text, as well as, a colored square according a gradient shown in the bottom part of the tool. When the analyst changes the selection on a panel, the application updates the information on the remaining panels according to the selection. Figure 3.3 shows a screen-shot of the Cube visualizer when displaying the results of CGPOP used in the previous example for TAU. For example, the Figure shows that the `MPI_Allreduce` invoked from `reductions.global_max` is the routine that executed more instructions during the application execution, but the distribution is uniform across the execution nodes.

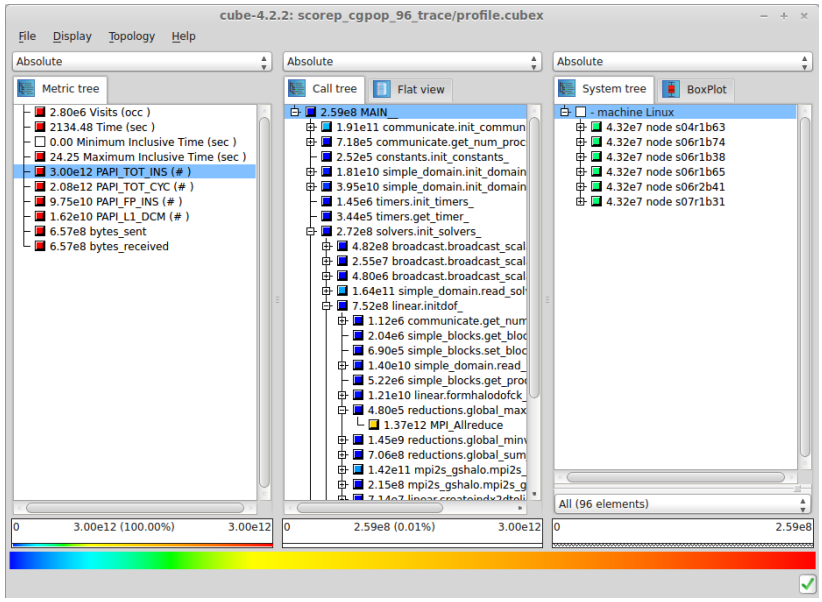


Figure 3.3
Scalasca's Cube visualization tool.

Periscope [80] is an online profiler that automatically searches for bottlenecks based on previous optimization experts' knowledge. Periscope offers several bottleneck analyses depending on the application type and the architecture where the application runs including load imbalance, excessive time MPI time due to several reasons, spent time in OpenMP exclusion regions and so. The results of Periscope are integrated into Eclipse through a plugin that helps to associate the obtained metrics with the source code in a single environment (see Figure 3.4).

It is worth mentioning that TAU, Scalasca, Periscope and Vampir joined efforts to define Score-P [147], a community effort for a common measurement infrastructure. From the developer perspective, a unique infrastructure helps to save manpower by sharing development resources and invest these resources in new analysis techniques. From the user perspective, there is only one framework to learn and it allows interoperability and data exchange between performance tools. However, at the moment of writing this thesis, Score-P does not offer all functionalities that are present in each tool's instrumentation package, such as sampling capabilities.

HPCToolkit [210] is an integrated tools framework for measurement and analysis of program performance on computers using a variety of parallel run-times (MPI, OpenMP, pthreads, among others). The framework uses statistical sampling of timers and hardware performance counters, in order to attribute accurate measurements of the program's work, resource consumption and inefficiency to the full calling context in that they occur. Data presentation in HPCToolkit enables rapid analysis of the execution costs, inefficiency and scaling characteristics both within and across nodes of a parallel system. The HPCToolkit output, as depicted in Figure 3.5a, is divided into two panels in `hpcviewer` [1]. The panel at the top displays the program source code whereas the bottom panel associates a table of performance metrics with the source code.

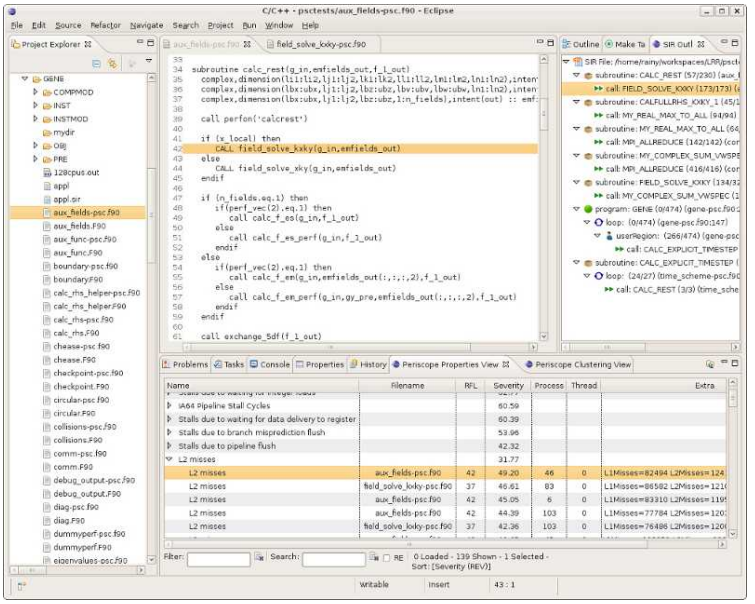


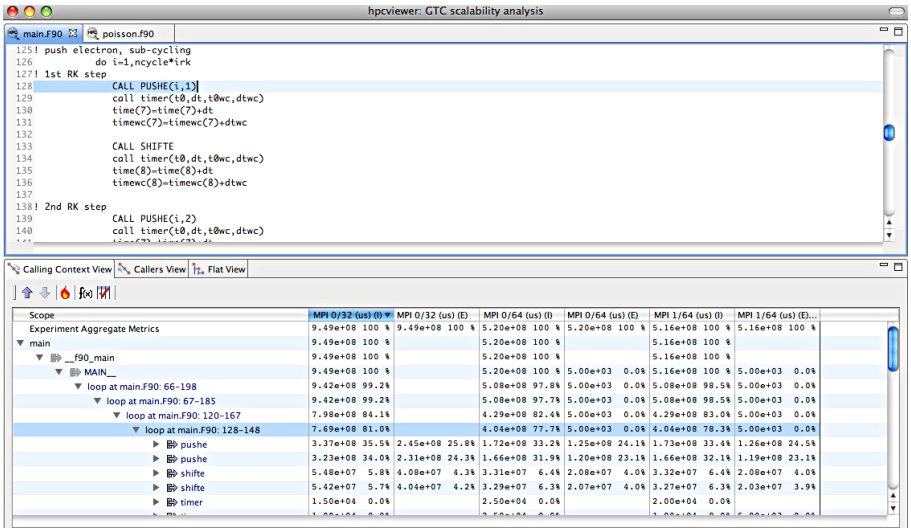
Figure 3.4
Periscope results shown in the Eclipse plugin (picture obtained from [213]).

3.2.2 Trace-file based tools

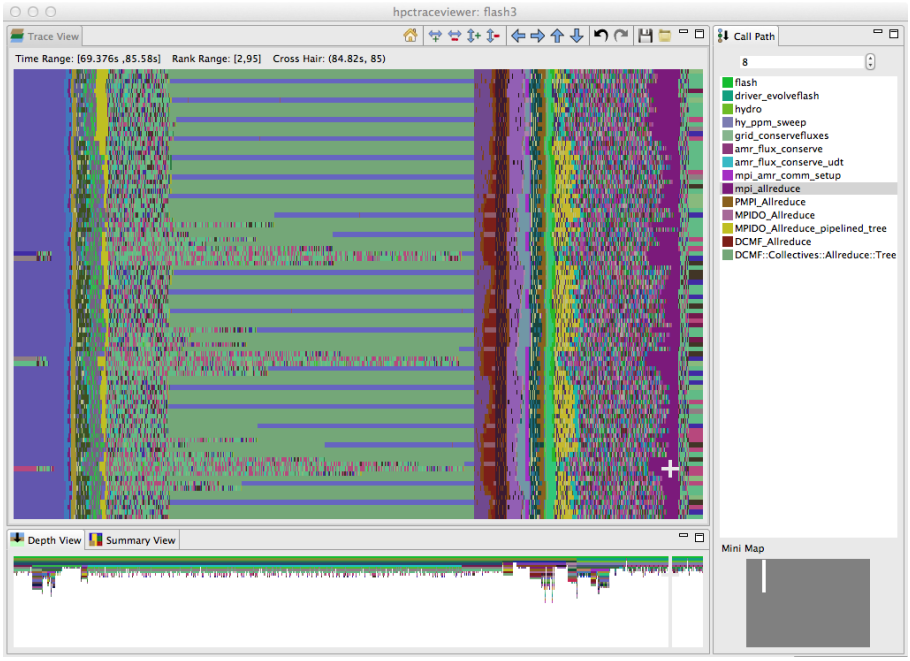
Paraver [120, 118] is a flexible data browser that does not offer hardwired metrics but which allows their programming according to the semantics of the data contained in a trace-file. Using filter and semantic modules, the analyst creates time-lines, profiles and histograms from trace-files to selectively display a huge number of performance metrics. The views can be combined to find correlations among the causes of performance drawbacks. To capture the expert's knowledge, any set of views can be saved as a Paraver configuration file, to be reused either in subsequent analysis or in the other trace-files loaded within the framework, allowing multi-experiment comparison. Extrae [66] is an instrumentation and sampling tool that collects metrics from several parallel and accelerator run-times (such as MPI, OpenMP, CUDA, OpenCL, OmpSs, among others) that generates Paraver trace-files, although other tools have created their own trace-file translators into Paraver format.

Figure 3.6 shows two Paraver time-lines. Time-lines represent the activity over time on the X-axis, the processes in the Y-axis and the color indicate the type of activity of a given thread in a given time span. The screen-shot at the top shows the executed MPI calls during the application run where Paraver has assigned a unique color to each MPI routine. The picture below shows the duration of the computation regions (i.e. in between two consecutive MPI calls) using a gradient that represents low values in green and high values in blue. Due to the definition of the data represented, the two windows show complementary information. Note that the processes that reach MPI calls earlier (mainly tasks 17 to 33) result in a shorter computation time; so the duration is represented in a color that tends to green. Paraver depicts the remaining tasks in a more intense blue because they take more time to complete.

As mentioned above, Paraver also represents data using tables (matrices or histograms). Paraver shows tables depicting processes on each row, but data represented in columns varies. Figure 3.7 shows different uses of tables in Paraver when representing different data in columns. On the table at the top (Figure 3.7a, columns refer to discrete values such as MPI routines. In this example, the table associates the percentage of time spent on each routine by each process and uses numerical value as well as colors the cells to represent the value of a particular cell. On the histogram in the middle (Figure 3.7b), columns refer to ranges of values and more specifically to

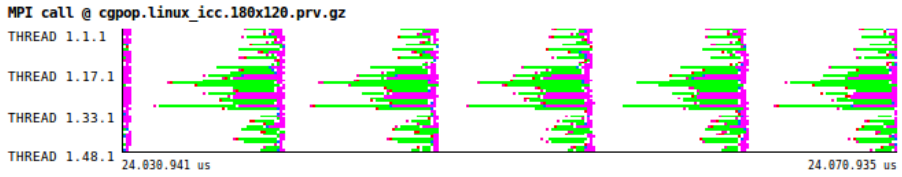


(a) HPCToolkit assessing the hot-spots in an application.

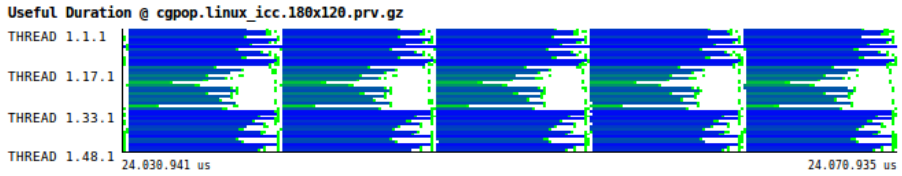


(b) HPCToolkit showing a part of the execution of a parallel application.

Figure 3.5
Visualization mechanisms available within the HPCToolkit suite (images obtained from [2]).



(a) Paraver displaying MPI activity.



(b) Paraver displaying a histogram.

Figure 3.6
Paraver representing trace-files as time-lines.

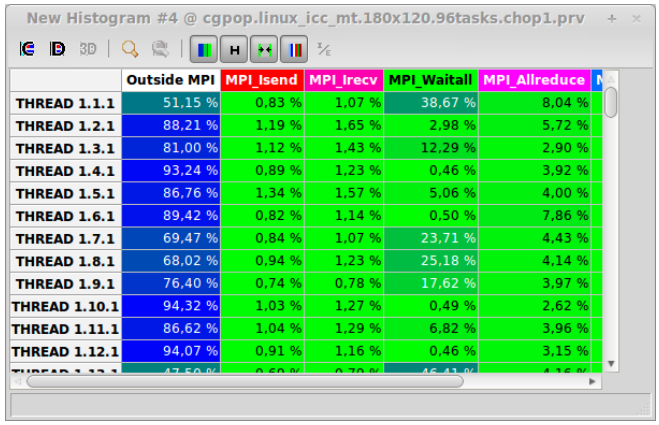
this example, it refers to ranges of duration. The cells with data are shown in gradient according to the time spent on that particular combination of thread and range, while the cells without data are shown in gray. In this particular case, the columns refer to the duration of computing regions and the data represented in the cell refers to the achieved IPC. This plot shows that the duration of the computing regions differ from process to process because points at different columns. It is also noteworthy that the achieved IPC depends on the duration of the computing region, the low (green) IPC on the left of the plot while the high (blue) IPC is on the right part. Finally, on the histogram at the bottom (Figure 3.7c) columns refer to processes so allowing the table to depict a communication matrix. In this plot, the color represents the average number of bytes sent from one process to another. The shape of the results shows that adjacent processes are responsible for most of the transferred bytes.

Vampir [157] is a performance analysis tool for parallel applications with a graphical data representation that enables detailed understanding of dynamic processes. Vampir features many functionalities such as adaptive statistics, support for referencing source code locations, hierarchical grouping of computing resources as well as filter techniques for processes, functions and messages. The framework also relies on a scalable design that distributes performance data visualization and increases the scalability and the responsiveness compared to the sequential approach [22].

Figure 3.8 shows different analysis techniques using the Vampir performance analysis tool. More precisely, Subfigure 3.8a shows a process time-line with detailed information about functions, communication and synchronization events where each row represents a single process; whereas Subfigure 3.8b is a function summary that gives an overview of the accumulated time consumption across all function groups and routines. For example, the tool accumulates the elapsed time of every call to `MPI_Send` into the MPI function group time. Finally, the screen-shot in Subfigure 3.8c illustrates the invocation hierarchy of every monitored function in a tree representation by revealing information about the number of invocations of a given function, the time spent in the different calls and the caller-callee relationship.

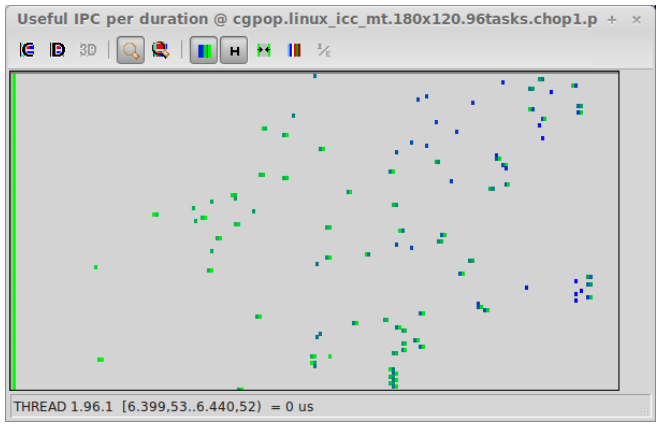
Pajé is an interactive visualization tool designed to allow inspection of every object displayed, to cope with a large number of threads and to allow its extension with new functionalities [113, 117]. The tool also limits the tracing intrusion by compacting events during the application execution, with space savings about 50%. Pajé offers several filtering and zooming features to help programmers to cope with large amounts of information. These features include event grouping, event removing according to the type of the visual object, object repositioning to save screen use, production of synthetic views and change the type and color of entities.

HPCToolkit has recently added a tool into its suite that allows users to visualize the sampled

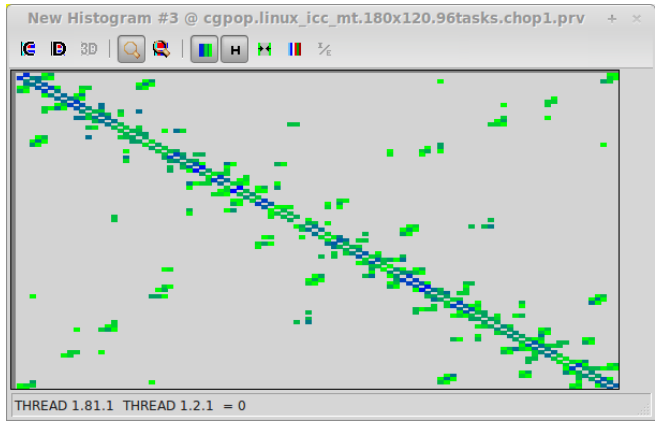


	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Waitall	MPI_Allreduce
THREAD 1.1.1	51,15 %	0,83 %	1,07 %	38,67 %	8,04 %
THREAD 1.2.1	88,21 %	1,19 %	1,65 %	2,98 %	5,72 %
THREAD 1.3.1	81,00 %	1,12 %	1,43 %	12,29 %	2,90 %
THREAD 1.4.1	93,24 %	0,89 %	1,23 %	0,46 %	3,92 %
THREAD 1.5.1	86,76 %	1,34 %	1,57 %	5,06 %	4,00 %
THREAD 1.6.1	89,42 %	0,82 %	1,14 %	0,50 %	7,86 %
THREAD 1.7.1	69,47 %	0,84 %	1,07 %	23,71 %	4,43 %
THREAD 1.8.1	68,02 %	0,94 %	1,23 %	25,18 %	4,14 %
THREAD 1.9.1	76,40 %	0,74 %	0,78 %	17,62 %	3,97 %
THREAD 1.10.1	94,32 %	1,03 %	1,27 %	0,49 %	2,62 %
THREAD 1.11.1	86,62 %	1,04 %	1,29 %	6,82 %	3,96 %
THREAD 1.12.1	94,07 %	0,91 %	1,16 %	0,46 %	3,15 %

(a) Paraver displaying a profile.

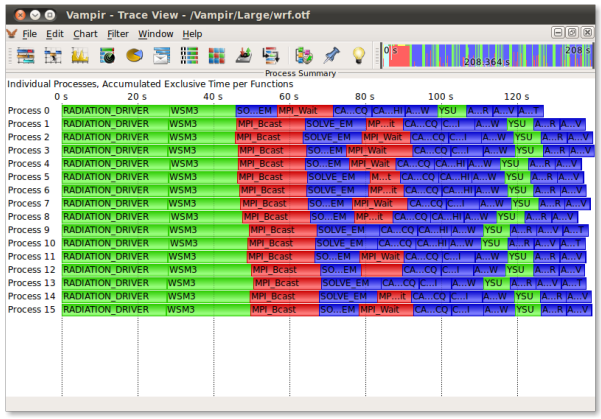


(b) Paraver displaying a histogram.

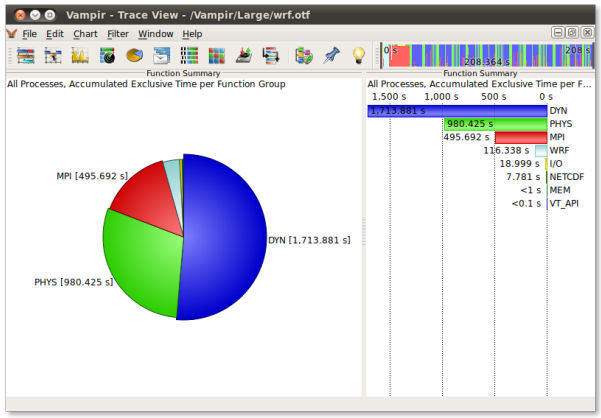


(c) Paraver displaying a communication matrix.

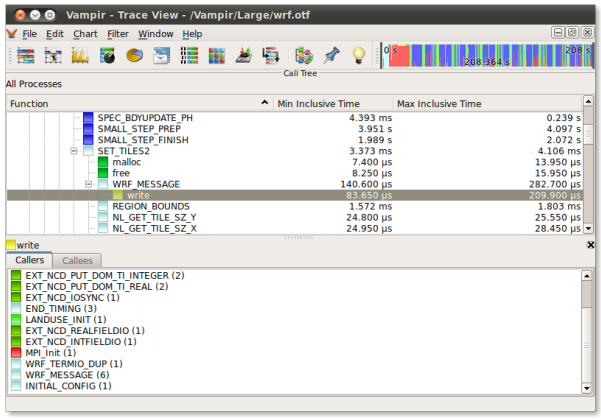
Figure 3.7
Paraver tabulating results.



(a) Representation of the execution in a time-line.



(b) Summary of the executed function timings.



(c) Invocation hierarchy of instrumented routines.

Figure 3.8
Multiple views from the Vampir tool-suite, showing from time-lines to profiles. Images obtained from [34].

call-stack data in a time-line using the `hpctraceviewer` [212] tool. `hpctraceviewer` displays the samples for process, the time dimension and the call depth in a GUI as shown as in Figure 3.5b. In the Figure, the time-line is shown in the top plot of the screen-shot. The X-axis and the Y-axis represent time and the processes of the application, respectively and the colors in the time-line represent the routines that were executing at that time.

The Oracle Solaris Studio [39] (formerly known as Sun Studio Performance Analyzer) is a performance tool framework that comprises a set of tools for collecting and viewing application performance data using tracing and profiling mechanisms. The framework provides many tools, each responsible for analyzing a specific subset of problems such as causes of communication delays in MPI applications and the efficiency of multi-threading in a program. These tools instrument synchronization calls, heap allocation and deallocation routines, OpenMP constructs, MPI routines and also uses a sampling mechanism to collect information regarding the user routines. The framework also provides a GUI that provides views as time-lines, call-trees and source code disassembly to name a few.

3.2.3 Time-series profiles

Some profile-based performance tools have extended their collection and presentation mechanisms to explore profiles as a function of time (begetting the time-series profiles). These profiles allow the analyst understand adaptive applications where performance changes from one time-step to the next during the execution, resulting in a combination of both profiles and trace-files. To achieve this functionality, performance tools separate each of the profiles in phases so that each phase is summarized and explored independently. In such profilers, phases are either provided by the analyst or determined by the tool automatically.

For instance, OmpP is capable of providing phase-based profiles by providing two different approaches to detect phases. The first approach relies on sampling techniques to obtain profiles in regular, uniform intervals. This approach allows adapting the capture rate depending on the behavior of the application, increasing the dump rate if the application behavior changes. OmpP also enables detecting phases by exposing an API for dumping profiles at user request, which in a typical scenario would be aligned to outer loop iterations.

Tools such as TAU, Scalasca and Periscope share the need for the analyst to separate application run into phases, but each tool focuses on phase analysis pursuing different objectives. Mainly, they evaluate how parallel application performance evolves at simulation time-steps, identifying phases where the application re-balances its workload. While TAU allows distinguishing performance phases along different execution time-steps, it requires having former knowledge of the application in order to enclose the time-step loop body with entry and exit events that instructs the profiling system when a phase starts and stops. The Scalasca team extended this idea and developed a lossy compression mechanism for time-series call-path profiles [207] to establish a link between a performance problem and the context where it occurs with respect to the time-steps of the application. Their approach uses incremental clustering to generate a condensed version of the profile data to reveal temporal evolution of performance phenomena. Finally, Periscope also included into their framework the times-series profiling to automate the analysis and tuning process using plugins [164] that allows measuring performance metrics at each phase and capture the evolution over the iterations of the simulation.

3.2.4 Overview of simulation tools

Although this thesis focuses on performance analysis from the application perspective, it is valuable to examine performance exploration alternatives when evaluating the hardware design space. While this thesis does not aim at providing the same level of insight as the tools that belong to this group, it is worth to provide a small overview of these tools. Despite the large quantity of categories to be simulated in current HPC systems, this brief summary only covers the simulation of the two most representative aspects in these type of computers: processors and networks.

With respect to the processors, their full simulation require on the order of months to complete due to the cost of the simulation speed and there has been extensive research related

to application workload reduction. Simpoint [195, 196] is a tool that automatically finds sets of simulation points gathered by instrumentation tools such as Atom [65], SimpleScalar [25] and PIN [135]. These simulation points are referred as *sampled traces* but there are issues to be dealt with when selecting trace samples so that these samples represent the program performance. These issues include determining the sample length, the number of samples needed, the hardware state when simulating the sample (commonly referred as warm up) and the sample selection itself. Eeckhout and others presented an algorithm that determines the optimal sample length per benchmark in different warm up scenarios for sampled cache simulations [58]. Nowadays, multithreaded architectures have become so popular that all new processors are capable of executing multiple processes simultaneously. To evaluate these capabilities, some computer architecture researchers use benchmark suites that serve as a workloads composed by non-cooperative threads instead of analyzing truly parallel applications [224]. Since simulating all the benchmark combinations is excessively time-consuming, there has been research to analyze a benchmark suite and find all the distinct behaviors that occur when pairs of benchmarks run together [223]. FAME [226] addresses this problem with a new methodology aimed at fairly measuring the performance by re-executing all sampled traces in a multithread workload until all of them are fairly represented in the measurements. More recently, Carlson *et al.* have considered a general purpose multi-threaded application sampling methodology that takes into account thread synchronization such as locks and barriers [29].

Regarding the simulation of the network subsystem, the literature shows several approaches with variable degree of simulation detail. Packet-level simulation tools [216, 236, 169, 44, 225] have been considered the most detailed simulation tools with respect to the details of simulation because they consider a large number of aspects to accurately model the network behavior. However, these simulation tools generate large amounts of traffic and incur in high overheads, resulting in simulations that take longer to run than the actual application, which makes them inappropriate to evaluate in-production applications. To circumvent the expenses of detailed packet-level simulation approaches other simulation tools use simpler models ignoring the network complexity. Dimemas [119, 9] is a simulation tool for message-passing programs aimed at providing fast simulations without simulating all the system's details and using a trace-file from the MPI activity. Dimemas replays such a trace using an architectural machine model consisting of a network of SMP nodes and generates various statistics as well as a Paraver trace-file. The model is highly parametrizable, allowing the specification of parameters such as number of nodes, number of processors per node, relative CPU speed, memory bandwidth, memory latency, number of buses. The work described in [6] employs static code analysis to simulate only one iteration of communication patterns with the consequent in the data captured volume. MPI-SIM [172] is a parallel simulator for performance evaluation of MPI programs that uses direct execution to obtain the computation time of programs but simulates communication and I/O times during the execution by swapping MPI calls to a library of the simulator. There are co-simulation efforts such as Venus [151] in which the boundary of the co-simulation lies between the detailed simulation of the network (Venus) and the replaying of an application's trace (Dimemas). There are some models to define flow-based network models considering network topology and network contention to predict the performance in a wide range of parallel applications [14]. These models consider several network aspects such as: communication architecture, collective communication translation into point-to-point messages, interconnect topology and contention when sharing a network link.

3.3 Performance analytics

As we have seen in this chapter, performance tools gather lots of information. Since analysts invest considerable time delving into such information, it is worth considering the application of techniques from other research (such as data analytics) into the performance analysis area to assist in finding performance answers. Some performance tools have introduced data analytics methods into their frameworks to be applied to performance data (begetting what they call performance analytics), so enabling mechanisms to expose patterns of interest, detect application structure and compare multiple experiments, for instance.

The Pablo Performance Analysis Environment [179] provides unobtrusive performance data capture, analysis and presentation. The instrumentation mechanism of this tool monitors and alters the volume, frequency and types of event data recorded depending if the tool generates more data than the user requested. The tool also identifies a few equivalence classes of processor behavior from SPMD applications and then only records data from them, saving the tool from collecting a huge amount of performance data.

Scalasca [78] is a performance analysis tool-set designed for use on large-scale systems. The tool-set offers an incremental performance-analysis procedure that integrates run-time summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. A distinctive feature is its ability to identify wait states that occur on unevenly distributed workloads, which limit scalability on large processor counts. Scalasca detect such wait states and related performance properties even in very large configurations of processes using a parallel trace-analysis scheme [77]. For instance, Figure 3.9 depicts the output of the Scalasca tool-set for an execution of the CGPOP application on 96 tasks using the Cube visualizer. The analysis of Scalasca indicates that approximately half of the total sends were issued before their receive counterpart (a metric named *Late receiver*) and most of these Late receivers occurred within the subroutine `mpi2s_gshalo_update_2d_dbl`.

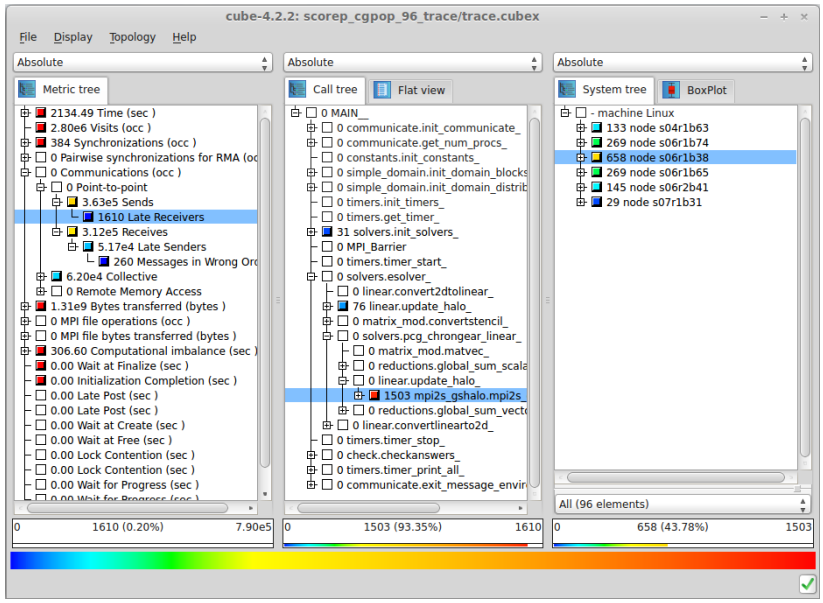


Figure 3.9
Example of results provided by the Scalasca performance tool.

Since a time-stamped sequence of events is a function of time, signal processing techniques (in particular the wavelet transform) are worth applying on trace-files. Casas *et al.* describe an autonomous phase detection mechanism of MPI application's execution based on the frequency behavior extracted from the sequence of events [32]. This phase detection mechanism also enables authors to study the scalability of the application and determine the limiting factors within these phases. This processing enables on-line analyses on which tracing only emits information for the detected periodic phases [132], with consequent savings in space and post-processing time.

Ocelotl [54, 53] proposes an innovative visualization technique that provides a consistent overview of the temporal and resource dimensions. The tool uses an aggregation model that detects and merges areas of a trace-file that are temporally and spatially homogeneous to provide higher-level visualizations while preserving the microscopic information content. The algorithm

of the tool computes a partition of space and time that optimizes a trade-off between the representation complexity and information loss. The aggregation consists of a two-step process that partitions the entity set into disjoint aggregates first and then, reduces the microscopic data and proves an overview according to the partition. Aggregation is a complex task that consists in optimizing a trade-off between data reduction and information loss. Several information metrics help in this optimization while measuring the information loss, data reduction and to measure the balance between this trade-off.

It also occurs that applications expose different performance behavior when they run in a variety of environments. The user may wonder which compiler version generates the best code, which block size improves the application efficiency, or how the application behaves at different processor scales. PerfExplorer [95] is a framework to conduct comparison of several experiments based on data mining techniques built on top of PerfDMF [96] analysis infrastructure for parallel performance profiles. Figure 3.10 shows a comparison of four CGPOP executions using 48, 96, 192 and 360 processes in MareNostrum3. On the top plot, the tool shows that the application does not scale ideally among experiments (called trials in PerfExplorer) and the plot below decomposes exclusive time among several components (mainly user routines and MPI calls). Note that the decomposition indicates that `solvers.pcg_chrongear_linear_` scales superlinearly, `matrix_mod.matvec_` and `MPI_Waitall` scale linearly, but `MPI_Allreduce` does not scale.

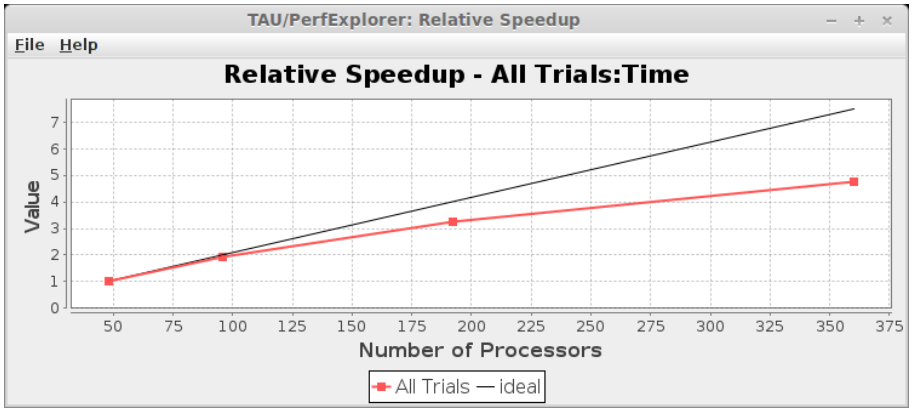
Huck describes an automatic process for parallel performance experimentation, analysis and problem diagnosis [97]. Such a process is built on top of the PerfExplorer performance data mining system combined with the OpenUH [128] compiler infrastructure. PerfExplorer eases the comparison of several experiments using the same application and the selective instrumentation of TAU [194] helps to avoid excessive overhead during the execution and gives the opportunity to provide optimization suggestions to the user.

Llort *et al.* introduces a mechanism based on object tracking techniques from the computer vision field that help to identify computation regions across multiple experiments [131]. This work provides performance trends of regions and insight on the influence of different parameters and execution conditions. This way, the tool presents a summarized result that indicates whether scalability expectations are met, how the compiler affects the performance and if application parameters alter the performance of specific computation regions.

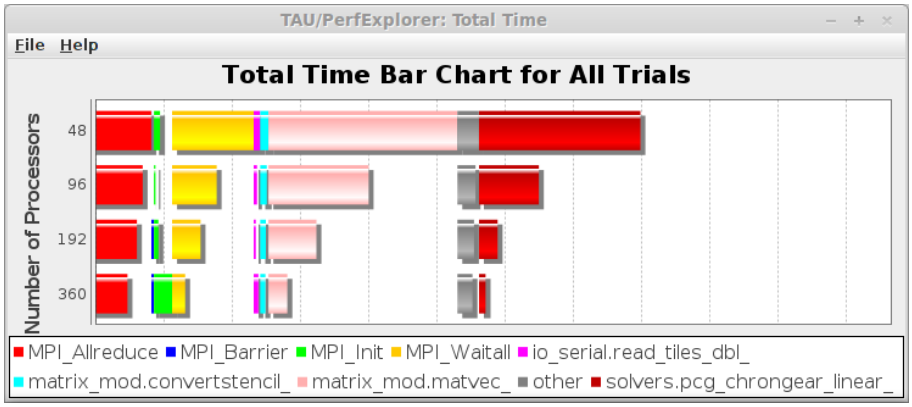
Finally, expert systems are also used in the performance analysis area. For instance, Paradyn [149] employs expert's knowledge to drive the analysis of the instrumented application according to the W^3 [94] search model for the *why*, *where* and *when* information regarding the search for performance bottlenecks [149]. The Performance Consultant module from Paradyn has a well-defined notion of performance bottlenecks and program structure, so that it associates bottlenecks with specific causes and specific parts of a program.

The PerfExpert [26] tool employs the HPCToolkit measurement system to execute a structured sequence of performance counter measurements to detect probable core, socket and node-level performance bottlenecks in important procedures and loops of an application. For each bottleneck found PerfExpert provides an assessment and suggests steps to the programmer to improve the performance. Authors introduce the LCPI metric (local cycles per instruction) which combines performance counter measurements with architectural parameters to make measurements comparable. In order to calculate the LCPI, the tool executes the application several times before analyzing in order to capture all the necessary performance measurements.

Periscope also employs expert's knowledge to drive the analysis of the instrumented application while the application runs. Depending on the automated analysis of the performance data and some properties provided by formalized expert knowledge, the tool determines which additional performance issues and search strategies to follow. For example, Periscope may analyze the proportion of stalled cycles during the execution and evaluate the cache miss and branch misprediction ratios to identify the sources of such stalls and suggest changes during the application execution, or skip to the next type of analysis.



(a) Scalability analysis in PerfExplorer.



(b) Scalability decomposition as depicted in PerfExplorer.

Figure 3.10
ParaProf showing an scalability analysis and its decomposition using multiple experiments of the CGPOP application ranging from 48 to 360 MPI processes.

3.4 Measurement collection techniques

Tools gather performance metrics during the application's execution to unveil the application behavior. To illustrate measurement collection mechanisms, the reader may consider two naive examples that include timing and inspecting the amount of memory allocated using the `time` and `top` commands, respectively. The former command measures the time spent executing a command whereas the second displays information from every process in the system periodically. To measure the time spent for a particular command, the `time` command obtains the time before and after the execution and then calculates the difference between the two. With respect to the `top` command, the metrics displayed are not computed by the `top` utility itself but the operating system when executing activities related to memory allocation. For instance, every time a process invokes a memory-related call (such as `malloc`, `free` and `realloc`) the operating system tracks the allocated memory and exposes this value through a certain set of calls that `top` uses to collect and display periodically.

Performance analysis tools apply these techniques to record the application activity by injecting monitors and letting the process to execute them (i.e. first-person monitoring). Therefore, these monitors interrupt the program to collect metrics. There are two ways to invoke the monitors: when an application activity occurs during the application execution or when an external

application event reaches the application process. The first mechanism, called instrumentation, refers the ability to inject probes at specific program locations (such as routines, loop bodies, sentences and even machine instructions). Due to the nature of the instrumentation, it provides accurate association between the gathered metrics and the application locations. The second mechanism to invoke monitors is named sampling and it runs independently whatever the application activity. The cost associated to each sample interval is correlated with the activity that was interrupted; therefore, the results obtained are statistical approximations. This inference requires a certain quantity of samples; that is, the obtained results approximate the actual distribution the application needs to run for a sufficient amount of time, yet highly volatile metrics may get lost. Sampling typically uses alarms to create a periodic signal, which consequently, simplifies the cost attribution because the cost associated with one sample is the sampling period. Both mechanisms have their advantages and disadvantages. When instrumenting, the captured metrics are directly related to the application code; so on the positive side measures are easily associated with the program structure (and any programming models applied). However, on the negative side, the granularity and the volume of measurements gathered are related to the program execution flow, so it is either possible that a short instrumented run captures lots of data, or *vice-versa*, a long instrumented run collects few data. Therefore, instrumenting the appropriate application points to unveil the performance may need some additional insight. In contrast, sampling easily lets the user tune the volume of data gathered since the quantity directly depends on the periodicity: the smaller the period (and granularity), the higher volume of data gathered. However, since sampling results are approximations to the actual distribution, highly volatile metrics may not be captured unless choosing a small period. Whatever the alternative, either instrumentation and sampling, both approaches need to interrupt the application, so the more frequent the monitoring occurs, the more overhead suffered. This additional overhead poses a problem because it alters the application performance observed and then mislead the analyst during the analysis of the results.

3.4.1 Instrumentation alternatives

As discussed, instrumentation consists on modifying the application at specific points to inject monitors. Currently, there are several methods to modify application codes. The probes are added either in the source code, during compile-time, in the link stage, when generating the program binary, or even at the start of the execution.

Modifying source code It is the most fundamental method to add monitors into the application because it involves changing the source code to add calls to the monitoring routines. This form of instrumentation allows adding probes into any syntactical level of the application (ranging from routines to sentences) at desired application points resulting in a very precise injection mechanism. This flexibility comes at a price because this approach requires access to the source code to modify it, it requires some understanding of the application behavior in order to know where to inject probes and also alters the compiler optimization process. For instance, Table 3.1 shows a simplistic mechanism to instrument the entry and exit points of the function named `fact` by adding two calls to the monitors called `monitor_routine_entry` and `monitor_routine_exit`, respectively.

Compiler-assisted Compilers translate the application source code into machine instructions to generate the program binary. Since compilers have full control of the executed instructions, they can inject calls to performance monitors during the code generation. While nothing except overhead limits where to inject probes, practice shows that the commonplace compiler suites (including IBM XL [100], Intel [104], GNU [72] and clang [133]) only offer callbacks at the entry and exit point of the routines so that performance tools report the performance at user function level. Some source-to-source compilers, such as OPARI2 [222] and Mercurium [23], are capable of injecting additional instrumentation monitors to the parallel programming model and so provide performance metrics associated with the OpenMP and OmpSs constructs, respectively. Similarly to the previous approach, using compile-time instrumentation alters the compiler optimization process.

Table 3.1
Instrumentation of entry and exit points of a routine. On the left, there is the original source code for the `fact` routine. On the right, there is the instrumented routine where monitors are injected at two points: first, before any instruction of the routine occurs, and second, before returning from the routine.

```
1  int fact (int f)
2  {
3    int result;

4    result = 1;
5    while (f > 0)
6      result = result * f--;

7    return result;
8  }
```

```
1  int fact (int f)
2  {
3    int result;
4    monitor_routine_entry (fact);
5    result = 1;
6    while (f > 0)
7      result = result * f--;
8    monitor_routine_exit (fact);
9    return result;
10 }
```

Link-stage Libraries are files that contain common functionalities (such as the MPI and OpenMP programming model run-times) and are used by the compiler and linker to generate the final program object. The use of these libraries in conjunction with a particular annotation that denote an overridable (or *weak*) symbol[1] allows injecting monitors at specific routines offered by the libraries. As a result, performance tools take advantage of weak symbols to inject calls to the measurement procedures and allow injecting code without modifying the source code or altering the compilation process; however, this instrumentation is restricted to libraries that allow substituting their symbols. For example, the MPI specification enforces the use of weak symbols to provide instrumentation capabilities of its implementations. The specification results in two symbols: strong symbols, which implement the MPI specification and weak symbols, which serve as instrumentation points to the MPI routines [70].

Binary patching Although the aforementioned mechanisms inject monitors during any step of the binary-object making process, it is possible to modify already built application binaries, even if they are optimized. An application binary contains, among many other data, a collection of routines formed by sequences of byte-code instructions. Binary patching consists on altering these sequences so that the application does not behave exactly as originally expected but adding invocations to monitoring routines in between byte-code instructions. These modifications either remain during the application execution by modifying the process image, or become persistent by storing the modified binary for subsequent executions. Therefore, the main advantage of this mechanism is that it allows injecting monitors into optimized application binaries even if the application source code is not available or cannot be compiled or linked. However, binary patching is an elaborated technique because it involves parsing the byte code, so most of the existing binary patching tools and library only focus on one architecture, such as ATOM [65] for Alpha processors and PIN [135], MAQAO [107] and Dynamorio [21] for Intel processors. The exception to the former tools is the DynInst instrumentation package [24] because it encapsulates the instrumentation mechanism and separates it from the instruction decoding and source code parsing. As a result DynInst handles different architectures, such as Intel x86 and IBM®PowerPC®(and not mentioning the recently discontinued Intel®Itanium®), transparently. Performance tools based on DynInst analyze the structure of the application source code to determine what to instrument. For instance, the instrumentation package based on DynInst developed by Mußler *et al.* [156] instruments routines according to their size in number of lines. Listing 3.5 exemplifies how to instrument a routine using DynInst. The example is meant to inject at the routine entry and exit points by using independent monitoring functions (`wrap_begin` and `wrap_end`, respectively). Lines 4-10

[1] See the following URLs for further references on this topic:
http://www.keil.com/support/man/docs/armlink/armlink_CACCEEIF.htm,
http://docs.oracle.com/cd/E19963-01/html/819-0690/chapter2-90421.html#chapter2-11 and
http://www.akkadia.org/drepper/dsohowto.pdf

Listing 3.5
Dyninst-based instrumentation code to instrument a given routine that receives `nparams` parameters. The instrumentation code adds a call to `wrap_begin` and `wrap_end` functions at the entry and exit points of the instrumented routine. For the sake of simplicity, error checking is omitted within the code.

```
1  void Instrumenter::wrapRoutine (BPatch_image *appImage, string routine,
2    string wrap_begin, string wrap_end, unsigned nparams)
3  {
4    BPatch_Vector<BPatch_function *> found_funcs;
5    appImage->findFunction (routine, found_funcs);
6    BPatch_function *function = found_funcs[0];
7    BPatch_Vector<BPatch_point *> *entry_point = function->findPoint (BPatch_entry);
8    BPatch_Vector<BPatch_point *> *exit_point = function->findPoint (BPatch_exit);

9    appImage->findFunction (wrap_begin, found_funcs);
10   BPatch_function *entry_function = found_funcs[0];
11   BPatch_Vector<BPatch_snippet *> entry_args;
12   for (unsigned u = 0; u < nparams; u++)
13     entry_args.push_back (new BPatch_paramExpr (u));
14   appImage->getAddressSpace()->insertSnippet (
15     BPatch_funcCallExpr (*entry_function, entry_args), *entry_point);

16   appImage->findFunction (wrap_end, found_funcs);
17   BPatch_function *exit_function = found_funcs[0];
18   BPatch_Vector<BPatch_snippet *> null_args;
19   appImage->getAddressSpace()->insertSnippet (
20     BPatch_funcCallExpr (*exit_function, null_args), *exit_point);
21 }
```

Listing 3.6
Sample instrumentation for the libc's `close` routine using the shared-library interposition technique.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>

4  #define __USE_GNU
5  #include <dlfcn.h>

6  int close (int fd)
7  {
8    static int (*close_real) (int) = NULL;
9    int res;

10   if (!close_real)
11     close_real = dlsym (RTLD_NEXT, "close");

12   monitor_routine_entry (close);
13   res = close_real (fd)
14   monitor_routine_exit (close);

15   return res;
16 }
```

locate the entry and exit points of the routine to be instrumented, while lines 11-17 and 18-22 inject calls to independent monitoring functions at those points. In the particular case of the entry point, the monitoring functions receives the same parameters as the routine instrumented (as listed in the loop in lines 14 and 15) so as to allow the monitor to intercept the value of the parameters.

Shared-library interposition There exist an additional instrumentation mechanism that combines the two previous mechanisms. Modern operating systems, such as Linux, FreeBSD, AIX and Solaris, support shared libraries, which are libraries focused on reducing the

overall memory use by running multiple applications and sharing the regions of code without requiring multiple instances of the same routines. These operating systems also allow loading a shared-library into an application process before the process is loaded[2], so the *preloaded* library loads the symbols first into the process address space and overwrites the implementation from the original shared-library. This type of instrumentation wraps the substituted routine and accesses to the original implementation using the dynamic loader API.

Listing 3.6 depicts a full example of how this interposition mechanism to inject monitors wrapping the routine named `close` from the `libc` shared library. The Listing shows the code that is injected into the application process before loading the rest of the application. Line six refers to the routine that is meant to be interposed (`close`) and it must have the same signature as the replaced routine. Line eight declares a pointer to a routine to the actual implementation of the routine. The routine `dlsym` is responsible for searching the address for the entry point of `close` within the existing shared libraries in the process address space. Since two libraries offer the symbol (the preloaded and the original), the instrumentation must search the routine on the second library. This search is achieved by passing the parameter `RTLD_NEXT` to the `dlsym` call. Finally, line 13 invokes the actual code for the substituted routine and which is wrapped (in lines 12 and 14) by the measurement code.

Additional instrumentation mechanisms To conclude this enumeration, it is worth mentioning alternative mechanisms. One of these mechanisms includes establishing callback functions at particular application spots. Whenever the processor reaches these points, the processor executes the associated callback before continuing with the original sequence of instructions. Since callbacks are added manually into any module or library, this requires manual instrumentation though it only serves the purpose to end up invoking the actual monitoring routine. For example, a recent specification proposal for the OpenMP run-time (OMPT) [59] allows executing monitors when certain OpenMP activity occurs as well as querying the status of the run-time during the application execution. Similarly to OMPT, PERUSE [112] describes the specification and implementation within the OpenMPI message-passing library of a mechanism to extend the basic PMPI capabilities by providing low-level information such as when a message left and arrived at the physical layer, when the buffers got allocated, and so.

Switching to accelerators, despite the fact that CUDA and OpenCL allow using callbacks, they also provide an additional approaches that work with events to collect timing information of the activity performed by the accelerator. CUDA allows injecting events into the device and they are processed with all the remaining commands by the accelerator. When the accelerator processes the injected events, the accelerator assigns a time-stamp that the performance tool collects to determine the elapsed time between events. While in OpenCL the behavior is similar to CUDA, the events are part of the OpenCL API calls so the application developer may, or may not, use them. In contrast, on CUDA applications the event injection is transparent to the user code by the performance tool.

To wrap-up this section, Table 3.2 summarizes the instrumentation possibilities of the aforementioned mechanisms. Considering the alternatives, compiler-assisted instrumentation and binary patching offer the wider range of possibilities. Follows manual instrumentation, which cannot add monitors in-between machine instructions[3]. Finally, the link-stage and library interposition can only measure at routine level. With respect to the source code access, neither binary patching nor library interposition needs to modify the source code; thus the analyst can proceed only with the application binary.

[2]The preloading is achieved using environment variables such as `LD_PRELOAD` in Linux systems.

[3]Unless the region to be instrumented is written in assembly code, but this is not considered for practical reasons.

[4]This alternative only allows instrumenting routines existing in libraries but not routines from the application code itself.

[5]Only applies to shared-libraries.

Table 3.2
Instrumentation alternatives available depending on the instrumentation mechanism used.

	User routines	Loops	Statements	Machine instructions
Modify source code	✓	✓	✓	
Compiler-assisted	✓	✓	✓	✓
Link-stage	✓[4]			
Binary patching	✓	✓	✓	✓
Shared-library interposition	✓[4,5]			

3.4.2 Sampling

The number of sampling alternatives to collect punctual performance measurements is limited when compared to instrumentation. Sampling mechanisms are implemented on top of operating system signals, which are asynchronous notifications sent to a process (or thread from a process) in order to notify that an external event has occurred. When the operating system sends a signal, the process receiving the signal stops its execution flow temporarily and starts executing the signal handler which invokes the measurement monitor and returns back to the normal execution flow when the monitor finishes. There is an assortment of signals that can be hand over to the application process, some of them are generated from the operating system when the application raises an error, but some other generated from external sources, such as devices or the user itself. However, performance analysis tools mostly rely on signals that occur many times during the application execution to periodically accumulate metrics. Even though time-based signals are the general approach to query periodically the status of the application process, hardware performance counters (described below in this section) provide additional periodic signaling. This way performance counters operate in such a way that the sampling handler executes every certain number of instructions, cycles, or cache misses, for instance [152].

While this thesis does not focus on architectural simulations, it is worth mentioning that in this field samples are not punctual information but a contiguous interval of dynamic instructions during the program execution. In architecture simulation, sampling techniques are typically split into two general types: statistical sampling and representative sampling. The former samples the execution in a random pattern without any consideration regarding the sample selection, while the latter involves choosing samples to uniquely represent repetitive patterns during the program's execution.

3.5 Performance metrics

Performance metrics indicate the application efficiency and help to figure out what are the existing bottlenecks. Although monitors collect a wide range of metrics during an execution, the execution time is the *de facto* measurement that determines the performance of an application. Due to the variety on the programming models and the complexity of the systems, time measurements solely do not provide enough insight with respect to the nature of the underlying problem.

So as performance analysis tools give relevant information related with the application structure, these tools must provide performance information related to the application nature. The most basic examples include detailing the number of invocations of each application routine so as to focus on a particular region of the application code. When analyzing parallel applications, performance tools may provide information related to the work balance to uncover situations where some processors do more work than the rest. In addition, it would be helpful to know the number of messages sent and their size, or which locks are limiting the parallelism, in message-passing or shared-memory applications, respectively.

Focusing on the node-level performance, performance tools need to expose node-level activity so as to unveil how the processor behaves because architectural design of the processor may limit the application performance. The PMU provides information relating to the processor activity through a wide variety of performance counters. Using the data observed by the PMU, the

Table 3.3
Number of performance counters on a wide variety of architectures.

Manufacturer	Processor	Availability	# pmcs[7]
IBM®	PowerPC®970	2,002	214
	PowerPC®440	2,004	329
	Power5®	2,004	448
	PowerPC®970MP	2,005	231
	PowerPC®450	2,007	455
	Power6®	2,007	553
	Power7®	2,010	545
	PowerPC®A2	2,012	406
	Power8®	2,014	1,144
Intel®	Xeon®E7450	2,008	130[8]
	Xeon®X5570	2,009	135[8]
	Xeon®X5680	2,010	229[8,9]
	Xeon®E5-2690	2,012	631[8,9]
	Xeon®E5-2670	2,012	166[8]
	Xeon®E5-2620 v2	2,013	1,056[9]
	Xeon®E5-2670 v3	2,014	194
	Itanium®9030	2,006	637
AMD	Opteron™2354	2,009	126
	Opteron™6172	2,010	147
	Opteron™6276	2,011	219
	Opteron™6274	2,011	219[10]
ARM	ARMv7 Cortex™-M3	2,006	81
	ARMv7 Samsung Exynos5 Dual	2,012	147

analyst detects whether the system either achieves a good ILP (in terms of cycles per instruction [CPI], or inversely, instructions per cycle [IPC]) or there are issues due to improper data accesses, branch misspeculations, to name a few. Unfortunately, event availability varies substantially among processor vendors and even within the same processor family. The increasing complexity of processors translates into an increased number of available performance counters, as shown in Table 3.3. The Table summarizes the number of available performance counters on different processors from several vendors that have reached the market recently. Note that many processors offer hundreds of events and separating those that measure valuable information from others that measure esoteric data is a complex task.

PAPI [20] is a widespread middle-ware for providing uniform access to performance counters on a variety of operating systems across architectures and it is employed by the tools used in this thesis. This software also promotes the definition of a standard performance counter set (called preset counters in PAPI jargon) that represent the typical *phenomena* observed in performance analysis, in contrast to the native counters (those provided by the underlying substrate) that are used to calculate the preset counters. The definition of these preset counters present two drawbacks. First, there are more than one hundred preset counters at this moment, so it is still difficult to choose which are relevant or not. Second, the availability of PAPI presets is restricted by the semantics of the native performance counters due to the complex mapping between the PAPI preset counters and the native performance counters and as a result it is possible that not all the presets are available.

As a consequence of the large event set count and the difficult interpretation of native counter, there has been research on defining analytical performance models based on a subset of the

[7] Number of performance monitoring counters.
[8] Several of these counters can be extra-qualified using two bit fields (named `umask` and `cmask` that slightly changes the semantics of the counters, thus increasing the number of available performance counters.
[9] Also include the uncore events.
[10] This is the processor used in the Cray XK7 Titan supercomputer. PAPI reports additional 686 performance counters related to the Cray component available in such supercomputer.

Table 3.4
The full Power7 CPI breakdown model.

Total cycles	*Completion cycles*	Completion cycles		
		Overhead of expansion		
	Completion Table empty cycles	due to I-cache miss		
		due to branch mispredict		
		due to I-cache miss and branch mispredict		
		other		
	Completion stall cycles	by LSU	by reject	Translation Stall
				other
			by D-cache miss	
			by Store	
			other	
		by FXU	by multi-cycle instruction	
			other	
		Stall due to IFU		
		Stall due to SMT		
		by VSU	by scalar	long
				other
			by vector	long
				other
		Others: Stall by BRU/CRU, flush penalty, *etc.*		

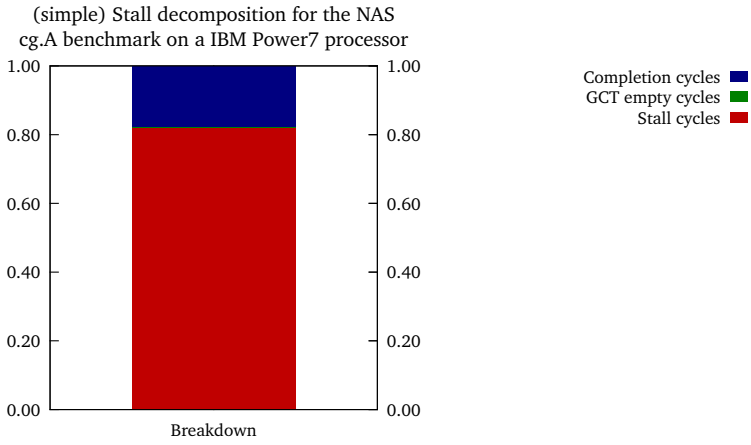
native counters. These models help the analyst to deal with a small assortment of simpler metrics rather than using hundreds of native performance counters. Following this section, there is an introduction to several performance models based on performance counters. Some of these models apply to specific processors because they require specific performance counters and their result is heavily tailored to the architectural disposition, but the remaining models rely on PAPI preset counters that enable them for a wide range of processors. For instance, the Statistical Stall Breakdown [8] describes a mechanism that samples hardware counters and dynamically multiplexes hardware counters to compute a breakdown model for a PowerPC based microprocessor. Such mechanism is implemented using the sampling capabilities in the K42 operating system. This operating system provides a performance monitoring infrastructure that allows on-line performance monitoring and uniform access from the OS and the user applications.

3.5.1 Models based on performance counters

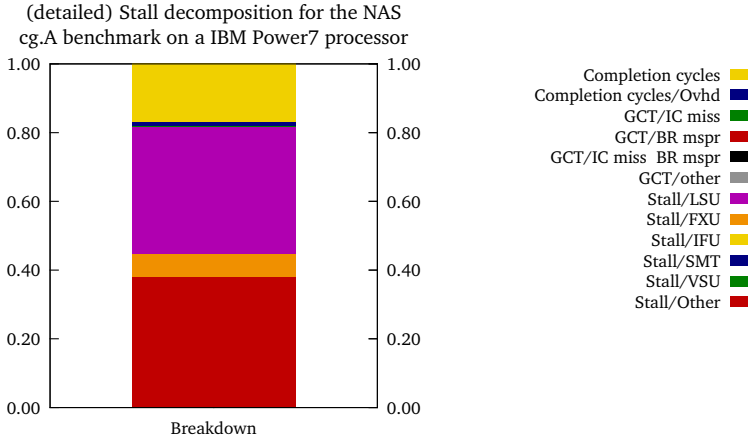
3.5.1.1 IBM CPIStack model

IBM published the CPIStack model for the IBM®Power5®processor originally [202] and for the Power7®and Power8®processors later [62, 145, 33]. The CPIStack model breaks the CPI down into several categories, which can be further classified depending on the category and focuses on identifying the type of stalls that prevent achieving a low CPI (or inversely, a high IPC). Table 3.4 summarizes the complete CPIStack model for the IBM®Power7®processor. The model for this processor divides the execution cycles into three top-level categories: completion stall cycles, Global Completion Table (GCT) empty cycles and completion cycles. Completion stall cycles report the number of stalled cycles in the processor and it is further categorized according to the processor component responsible for the stall (LSU for Load/Store Unit and VSU for Vector-and-Scalar Unit, among others). The GCT is a table representing a group of instructions being processed by the processor, so it must contain instructions to let the application progress. The GCT may be empty due to a branch misprediction or an instruction cache miss. Finally, completion cycles refer to cycles that have finalized an instruction. Although the model has been criticized by [67] because it cannot detect dependency chains and coupled effects between counters, it has proved useful in attributing stall causes on different applications [231, 47, 139].

The plots shown in Figure 3.11 depict the break-down on the first and second level of categories within the CPIStack model when analyzing the cg.A benchmark from the NAS benchmark



(a) Basic Power7 breakdown model showing the three metrics of the first categorical level.



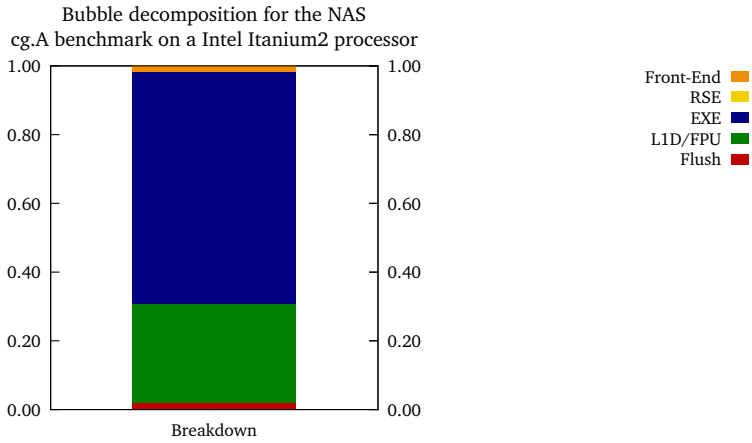
(b) Detailed Power7 breakdown model showing the twelve metrics that compose the second categorical level.

Figure 3.11
Two different approaches on the IBM Power7 CPIstack breakdown model depending on which categories are used. The models are applied to the conjugate gradient (CG) benchmark from the NAS suite.

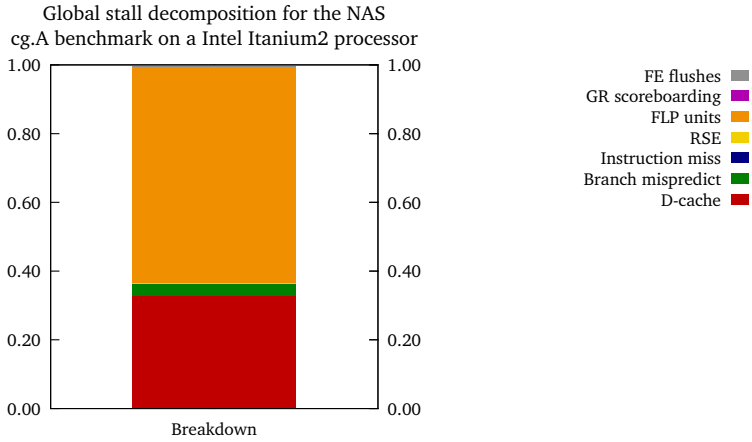
suite. In the simple decomposition plot (as depicted in Subfigure 3.11a), results show that Stall cycles dominate in the benchmark and that the processor always has work to do because the GCT stall cycles tends to zero. A detailed view on the stall decomposition (illustrated in Subfigure 3.11b) shows that instructions executed by the Load/Store Unit (LSU), the Fixed Point Unit (FXU) and non-classifiable instructions cause most of the stalls in the execution. With respect to completion cycles, a small fraction of them are related to the so-called overhead of micro-decoding instructions while the rest finishes instructions and so produces work.

3.5.1.2 Intel Itanium2 model

Hewlett-Packard defines two performance models for the Intel®Itanium2®processor [108]. The first of these models enables understanding the nature of the stalls suffered by associating to



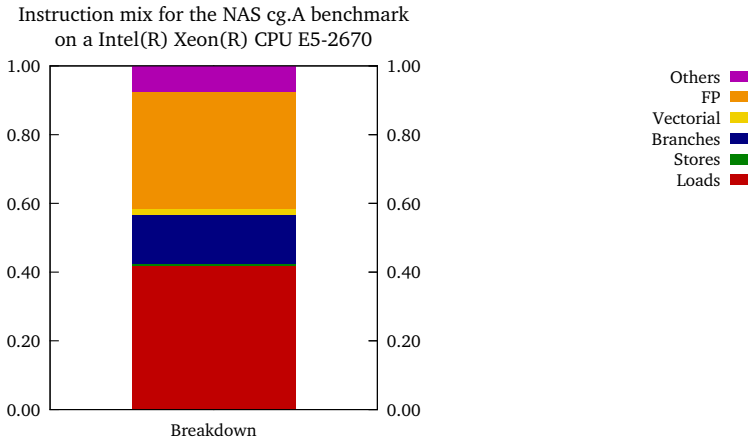
(a) Decomposition of injected bubbles in the pipeline according to the pipeline component.



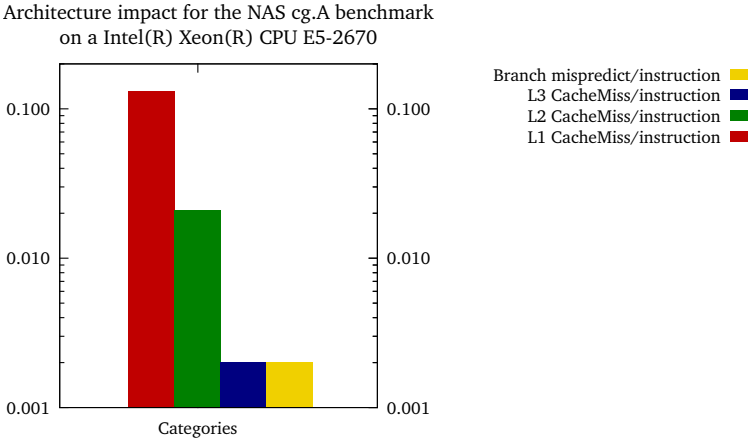
(b) Global stall decomposition.

Figure 3.12
Two breakdown models for the Intel Itanium2 processor. The model at the top categorizes which pipeline component (Front-End, RSE, EXE, L1D/FPU and Flush) injected bubbles in the pipeline. The model at the bottom decomposes the stalled cycles according to several components from the processor.

each block of the pipeline the number of bubbles that were inserted into the pipeline by that particular block. For the particular case of the Intel Itanium2 processor, the pipeline is organized in five blocks: the front-end (FE), the register-stack engine (RSE), the execution engine (EXE), the common L1 data-cache and FPU block (L1D/FPU) and the branch predictor (FLUSH) [143]. Figure 3.12a exemplifies the result of the basic model when applied to the cg.A benchmark and shows that the execution engine and the L1 data-cache and FPU are the blocks that inserted the most bubbles into the pipeline. As in IBM CPIStack, this performance model allows the provision of additional categories that specify the meaningful entities from the processor that required inserting the bubbles into the pipeline. For example, the detailed model (shown in Figure 3.12b) would to shed light on whether the FPU or the L1 data cache injected more bubbles, or which type of registers are causing the scoreboard to inject bubbles in the execution engine.



(a) Instruction mix.



(b) Architecture Impact.

Figure 3.13
Simple models that use common performance counters. The model at the top decomposes the instructions executed according to their nature, while the model at the bottom shows the efficiency of the architectural components such as the cache levels and branch predictor.

3.5.1.3 Simple performance models using common performance counters

While not every processor has a performance model available, it is noteworthy that simple, and yet helpful, models provide insight of how the application behaves on a particular processor. These simple models are built as ratios of common performance counters such as number of instructions, cycles, data-cache misses at several levels and branch mispredictions. While these ratios are not as expressive as the previous models, they still give lots of information with respect to the behavior of the application.

For instance, Figure 3.13 depicts two plots with two independent models applied to the cg.A benchmark from the NAS benchmark suite. The plot on the top provides information with respect to the instruction mix (i.e. the instruction type ratio) out of the total number of instructions and dissects the instruction type that is more frequent in a similar way to the Gibson

Mix [82]. The results show that Loads, followed by Floating-Point instructions, dominate the benchmark. The plot below establishes a relation between architectural components that may limit the performance, such as several levels of cache and the branch predictor and show their efficiency in terms of their proportion of cache misses and branch mispredictions, respectively. For example, the cg.A benchmark suffers from a relatively high percentage of L1 cache misses per instruction, surpassing the 10%, but the branch predictor anticipates the destination correctly most of the times. With these two plots, the analyst easily identifies which type of the activity is more frequent in a particular application and how processor components are behaving.

3.5.2 Active performance measurements

Currently, although computers have replicated many of their components to achieve higher performance, there are still some blocks connected to the processor that are shared among different processor blocks. An improper dimension of these shared blocks may result either in bad performance or in high production costs if the size is too fit or huge, respectively. Several shared blocks can be outlined: the Last-Level Cache, the cores within a processor and the network switches, among others.

Recently, a new set of methodologies to measure and model the performance the system have been proposed in different areas in order to predict the performance slowdown (if any) when moving to a machine with reduced characteristics. For instance, Eklov *et al.*, have created a framework to actively evaluate the memory bandwidth requirements of an application [60]. Casas and Bronevetsky have independently created a similar method [30], but also have extended their approach to measure the application requirements in terms of network bandwidth [31]. The frameworks include a special piece of software that utilizes a defined portion of the shared resource (i.e. memory or network bandwidth), so the framework calculates the sensitiveness of applications to different resource sizes and extrapolates how well the application would work on a different machine. Similarly, some of the recent Intel processors include the so-called TurboBoost technology which allows the processor to increase its frequency on certain conditions. When this option is enabled, a component of the system continuously monitors the cores activity, the thermal status and the voltage conditions to determine any frequency increase. Some experiments have demonstrated gains up to 6% in execution time when activating this technology but resulting in a higher energy drain [35].

3.6 Source code references

Anybody who recognizes the performance flaws and their causes in a piece of software, will realize that the correlation between performance and source code is essential. Such a correlation helps to understand the application behavior and enables to have a chance to improve it. This fact is more relevant as systems and applications are becoming more complex; and also, because the scenario in which analysts know little about applications and have to report how to improve the application performance is becoming more frequent.

Performance analysis tools perform these associations between performance bottlenecks and the source code and enable the analyst understand what the performance flaws are and which part of the code is responsible for them. These tools rely on either instrumentation or sampling techniques to monitor the application as the application progresses from one routine to the next. Instrumenting every routine from an application leads to exaggerated overheads when monitoring very fine grain routines, therefore requiring an additional execution to know which routines are to be excluded. For instance, a previous study of gprof shows that instrumenting all the routines of the integer version of the SPEC CPU2000 [91] benchmarks results in an overhead of 93% [74]. To avoid these drawbacks, many tools have promoted sampling to collect information regarding the representative routines of an application. Even though sampling typically provides the interrupted routine, some tool developers consider that reporting routines simply is insufficient because the results tend to be circumstantial clues. Some tools have been to provide a call-path (or sequence of nested routines) in order to differentiate between routine invocations [89, 122, 116].

The processor maintains a data structure name call-stack that helps with identifying which routine is being executed. The call-stack is updated every time the processor enters (or leaves) a subroutine by creating (or destroying) stack frames. These frames contain information such as the arguments received, the return address back to the routine's caller and space for local variables. *Call-stack unwind* is the process of inspecting these data structures and it involves accessing the processor structures to analyze the stack frames to emit relationships between callers and callees. The libunwind library [215] is considered the industry standard mechanism for call-stack traversals, due to its portability and accessible nature. The call-stack information is converted into human-readable information that includes the routine name, its container file-name and the source code line through the compiler added debugging information and the binutils package [71]. Since capturing the whole call-path is costly, some performance tools mitigate the resulting overhead in different directions.

CATCH [45] is a DynInst-based tool that associate metrics with call-path information for OpenMP and MPI applications. CATCH analyzes the call-graph of the program and uses call-site instrumentation to maintain a representation of the current call-path. The user selects subtrees of the call-graph to profile, allowing the tool to reduce the amount of instrumentation added in the program. As a result, CATCH statically predicts call-paths within the selected subtrees but cannot handle call-paths through dynamic calls.

iPath [15] is a call-path performance tool that allows the examination of the captured data as the application. This tool shares some similarities with CATCH because iPath also uses DynInst to instrument user-selected routines, but instead of instrumenting every call-site as in CATCH, iPath only inserts instrumentation in selected functions. To reduce the overhead of the tool, it performs a stack walk to determine the current call-path and then associates the sampled performance each time the application executes a monitored function.

Complete Call Graphs (CCG) is a mechanism to compress post-mortem call-stack traces [116] developed by Knüpfer and others. In this approach, CCGs build a call graph that replaces similar repeated sub-trees with references to a single instance. As a result, CCGs are very convenient for trace analysis tools as they reduce memory footprint and allow work with larger executions.

TAU explores two different approaches in this area and describes them in [137]. The first approach relates the ability of TAU to instrument routines and methods, but this leads to high measurement overhead when instrumenting short routines. To alleviate this problem, authors propose a new tool that allows the user to write instrumentation rules that will be applied to the measurement data to identify which events to exclude in a following execution. These rules are defined as numerical functions in terms of TAU metrics, such as the number of invocations or the mean duration being larger than a given threshold. The second approach involves implementing a call-path profiling in TAU that adapts to the application execution at every routine entry/exit point. In this case, identifying a call-path requires traversing a k-length structure, which may end up in non-uniform measurement overheads. TAU uses hashes to identify each of the possible call-stacks and associate the performance in order to reduce the measurement expense.

The Scalasca tool has also been extended using time sampling [208]. Like TAU, their major effort has been devoted to providing information about application routines instead of providing performance counters metrics. Scalasca provides accurate measurements when combining PMPI instrumentation and sampling by subtracting the time spent in MPI calls from a user routine collected during a sampling point. Also, Scalasca reduces the overhead during the call-stack unwinding process by inserting trampolines into the call-stack.

Scalasca also offers compressed time-series of call-path profiles to reduce the quantity of data to be collected during the execution [209]. Their approach uses lossy compression mechanisms by using incremental on-line clustering techniques on consecutive application time-steps. The tool only stores the complete performance data for the representative iterations, while the remaining iterations are associated with the representative, with the consequent savings in terms of space. While the tool only saves partial information, the tool reconstructs aggregate profiles when presenting data to the users by weighting clusters by the number of iterations they represent.

HPCToolkit also pursues correlating performance inefficiencies and the source code but do not rely on. Tallent *et al.* describe in [211] a mechanism to present performance data and the observed call-paths combined with the static program structure. This work is achieved using a context-free on-line binary analysis for locating procedure bounds and unwind information and

a post-mortem analysis of the optimized object code and its debugging sections. Still, their work require traversing the stack during the execution as detailed in [73]; however, to further reduce the overhead they rely on inserting hand-crafted trampolines inside the call-stack in order to limit the number of unwinds.

3.7 Memory references

This section describes earlier approaches related to performance analysis tools that have focused to some extent on the analysis of data structures and the efficiency achieved while accessing them. This research is divided into two groups depending on the mechanism used to capture the addresses referenced by the load/store instructions.

The first group includes tools that instrument the application instructions to obtain the referenced addresses. MemSpy [140] is a prototype tool to profile applications on a system simulator that introduces the notion of data-oriented, in addition to code oriented, performance tuning. This tool instruments every memory reference from an application run and leverage the references to a memory simulator that calculates statistics such as cache hits and misses according to a given cache organization. SLO [17] suggests for locality optimizations by analyzing the application reuse paths to find the root causes of poor data locality. This tool extends the GCC compiler to capture the application's memory accesses, function calls and loops in order to track data reuses and then it analyzes the reused paths to suggest code loop transformations. MACPO [176] captures memory traces and computes metrics for the memory access behavior of source-level data structures. The tool uses PerfExpert to identify code regions with memory-related inefficiencies, then employs the LLVM compiler to instrument the memory references and, finally, it calculates several reuse factors and the number of data streams in a loop nest. Tareador [205] is a tool that estimates how much parallelism can be achieved in a task-based data-flow programming model. The tool employs dynamic instrumentation to monitor the memory accesses of delimited regions of code in order to determine whether they can simultaneously run without data race conditions and then it simulates the application execution based on this outcome. Peña *et al.* have designed an emulator based data-oriented profiling tool to analyze actual program executions in an emulated system equipped with a DRAM-based memory system only [168]. They also use dynamic instrumentation to monitor the memory references in order to detect which memory structures are the most referenced. With this setup, they estimate the CPU stall cycles incurred by the different memory objects to decide their optimal placement at object granularity in heterogeneous memory system.

The second group consists of tools that take advantage of hardware mechanisms to sample addresses referenced when processor counter overflows occur and estimate the accesses weight from the sample count. The Sun ONE Studio analysis tool has been extended in [106] by incorporating memory system behavior in the context of the application's data space. This extension brings the analyst independent and uncorrelated views that rank program counters and data objects according to hardware counter metrics as well as shows metrics for each element in data object structures. HPCToolkit has been recently extended to support data-centric profiling of parallel programs [130]. In contrast to the previous tool, HPCToolkit provides a graphical user interface that presents data- and code-centric metrics in a single panel, easing the correlation between the two. Giménez *et al.* use PEBS to monitor load instructions that access addresses within memory regions delimited by user-specified data objects and focusing on those that surpass a given latency [83]. Then, they associate the memory behavior with several semantic attributes, including the application context which is shown through the MemAxes visualization tool.

3.8 Power measurements

The increasing interest in the energy-consumption related topics of HPC systems has also raised interest in analyzing dissipated power and using similar techniques when analyzing performance.

Table 3.5
Power model applied to the Intel Core2 Duo processor.

(a) Association between power and performance components (see [16] for further referencing).

Power Component	Modeled components
FE	Front-end
INT	Integer arithmetic
FP	Floating point arithmetic
SIMD	SIMD arithmetic
BPU	Branch prediction
L1	L1 Cache
L2	L2 Cache
FSB	Front-side bus and main memory

(b) Conversion factors from performance to power to be applied to the power components.

Processor Frequency	Power component								
	FE	*INT*	*FP*	*SIMD*	*BPU*	*L1*	*L2*	*FSB*	*Base*
0.8 GHz	148	10,815	362	169	5,069	55	106	147	1,309
1.6 GHz	336	8,680	769	360	10,613	113	184	335	2,508
2.1 GHz	566	8,553	1,334	620	17,676	193	357	565	6,272
2.5 GHz	789	8,852	1,908	856	24,437	261	502	788	8,701

There are several methods for capturing the consumed energy of a system and this section summarizes some of this work. They are classed under three groups.

The first group includes devices such as PowerMon [13] and PowerPack [76]. These devices provide accurate, fine-grained power measurements in computing systems because they sample the power drained while being connected to the computing systems through the ATX pin configuration. Since these devices access the power supply, they can monitor the power consumption from every component attached to it, including disks, GPUs and network devices among others. However, their main drawback is that they require physical access to the machine, which may not be possible due to security restrictions. The work described in [4] combines a self designed and implemented power meter that is attached to the computer with the Paraver performance analysis tool. The resulting combination allows the authors to enrich the traces that contain information with power information, giving the analyst the chance to correlate the source code and the parallel run-time calls with power metering reads. Their method of work consists on getting accumulated power measurements at node level by using low sampling rates for previously instrumented regions of code but require that all the cores execute one of the instrumented regions to report consumption for every core.

The second group involves instruction simulation. Wattch [19] and SimplePower [233] are cycle-level architectural simulators that estimate the CPU power consumption of every component of the socket by applying power consumption models. This type of simulation requires full instrumentation of the application, making the power estimation of the full execution unmanageable, not only because of the data size to be gathered, but also because of the time needed for producing the results. Consequently, these methods are not appropriate for day-to-day use on in-production binaries.

The third group includes research on the field of providing power models based on performance monitoring counters, such as the independent work developed by Singh *et al.* in [197] and Bertran *et al.* in [16]. The latter, for instance, presents a full methodology for creating power models based on performance monitoring counters that rapidly adapts to power changes. The result is a set of *formulae* that relates the activity on the processor components to conversion factors expressing their consumption and which summed up results in a power estimation of the whole processor. Table 3.5 shows the result of these power models based on performance monitoring counters when applied to the Intel®Core2 Duo®processor. The performance activity

is correlated with power components and the energy consumed by these components is directly related to the processor frequency.

The Intel® SandyBridge processor is the first generation of this brand of processors to introduce the Running Average Power Limit (RAPL) infrastructure which allows monitoring the drained energy by the processor. The Package Control Unit (PCU) is a part of RAPL and it is a combination of dedicated hardware state machines in charge for the RAPL infrastructure. The PCU is connected to power management agents that collect information such as power consumption and temperature and, control transitions between processor performance states and processor operating states. The PCU also monitors performance events from the cores, the I/O and the integrated GPU activity and weights them with energy factors in order to predict the socket's active power consumption. The resulting power consumption is scaled accordingly with operation conditions such as voltage and frequency of execution. Since RAPL is accessible through the PAPI library, performance tools such as Scalasca, Vampir, HPCToolkit and TAU, would easily provide performance and power information using this library.

4 The folding mechanism

> ‘‘Then let’s play,’’ says Enoch Root.
>
> — Neal Stephenson, CRIPTONOMICON

The granularity of the captured data depends on the monitoring frequency when using sampling and on the application activity when using instrumentation. Any attempt to gather performance metrics continuously involves either smartly adding more instrumentation points or increasing the sampling frequency, though both alternatives come with a consequent expense. This chapter introduces the main contribution of this thesis: the folding mechanism. This mechanism takes advantage of the repetitiveness exposed by many applications and smartly combines performance data captured by instrumentation and sampling techniques to provide instantaneous performance measurements.

4.1 Motivation

A performance tool that instruments routine entry and exit points only aggregates data whenever one of the selected routines starts or stops, and similarly, a sampling-based tool only accumulates measurements at sample points. In this direction, the performance data captured are discontinuous in time and any attempt to gather performance metrics more frequently requires either adding more instrumentation points or increasing the sampling frequency. Needless to say, smartly instrumenting more routines requires some previous application knowledge when it comes to choosing the instrumentation points, so this adds additional expense to gain this knowledge. For instance, a previous study of the `gprof` profiler shows that instrumenting every routine of the integer version of the SPEC CPU2000 [91] benchmark suite results in an overhead of 93% [73]. Also, as the reader may expect, the higher sampling rate, the more penalty suffered by the benchmarks. Figure 4.1 confirms this expectation by depicting the time dilation of several benchmarks from the SPEC CPU2006 [92] when increasing the sampling frequency in the Extrae [66] package when measured on an Intel®Xeon®E5 2670 running at 2.60 GHz. The overhead, irrespective of its source, is the result of the first person monitoring that alters the regular application performance, therefore the measurement results may be inaccurate and even misleading.

This chapter describes the main contribution of this thesis: the folding mechanism. This mechanism creates synthetic metrics that finely express the instantaneous node-level performance within an instrumented region at unprecedented levels of detail without incurring in a significant

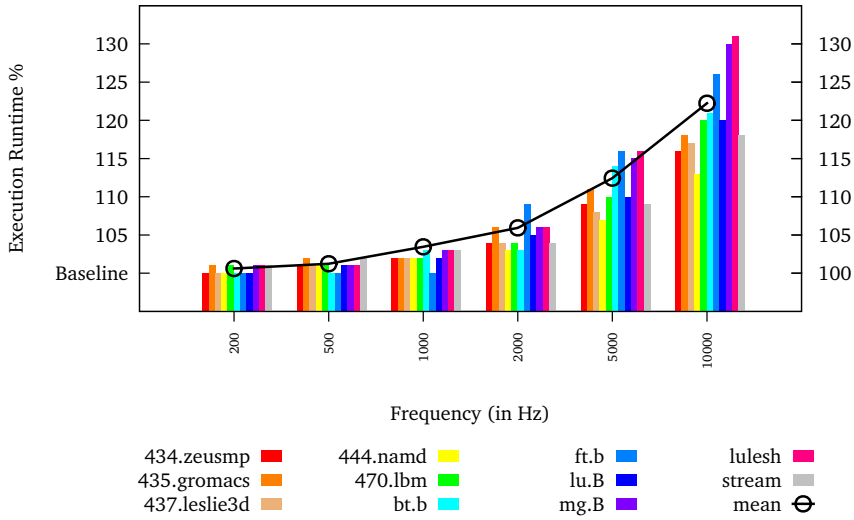


Figure 4.1
Measured sampling overhead in the Extrae instrumentation package when tested using different benchmarks and sampling frequencies.

overhead. The folded information covers data such as performance counters (either individually or grouped), call-path and memory object references and energy-consumption readings. This information proves very relevant when it comes to understand how the application behaves at a very fine-grain granularity, and, the presentation of the results ultimately enables the analyst to improve the application performance as shown in the forthcoming Chapter. This chapter proposes a methodology for evaluating the most dominant computing regions in parallel applications so that any chance to improve these regions should lead to higher returns. To implement this methodology, this chapter presents a framework that helps an analyst to evaluate the performance of parallel applications without requiring any preliminary knowledge of the application.

4.2 Description of the mechanism

High-performance computing applications typically have several well-defined phases. These applications start with an initialization phase where data structures are set up and distributed across processes. This is followed by an intensive and time-consuming phase in which multiple processes cooperate to reach a solution by combining sequences of computation and communication. Once the processes finalize, they output the generated data and the application then finishes.

In these applications, the computation phase tends to present a periodic behavior. Such behavior responds to the application structure because applications consist of a sequence of routines and loops that are executed iteratively for a number of time-steps. As a result of this periodicity, one expects the application performance also to expose a periodic behavior as well, in which every loop uncovers performance phases since each of them undergoes different inefficiencies. Ideally, if the application execution does not suffer any external interference (such as preemptions, interrupts, network and memory contention), it is possible to understand the collection of individual periods as an ergodic system. This behavior leads to a couple of consequences:

1. it is irrelevant which period is analyzed since every period behaves invariantly and,

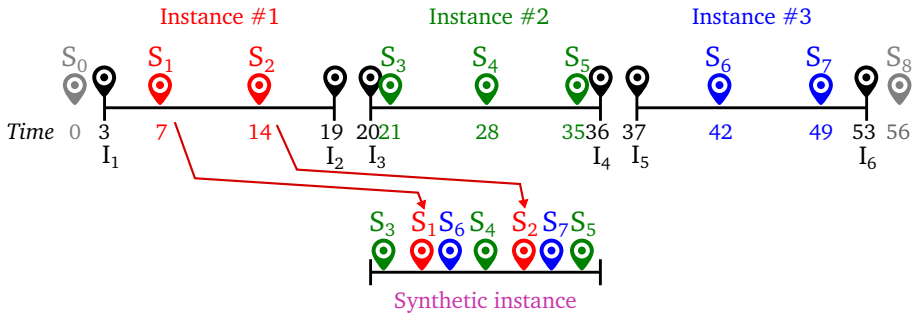


Figure 4.2
Illustration of the folding process.

Table 4.1
Tabulated exemplification of the folded results.

	Instance #1		Instance #2			Instance #3	
	S_1	S_2	S_3	S_4	S_5	S_6	S_7
T_s	7	14	21	28	35	42	49
T_i	3	3	20	20	20	37	37
δ_s	4	11	1	8	15	5	12

2. performance measurements should be the same at any given point within the period irrespective of the period observed.

The folding mechanism takes advantage of this repetitive behavior found in many HPC applications. The mechanism uses trace-files from these applications containing performance information collected through sampling and instrumentation mechanisms. The mechanism gathers and combines data of delimited repetitive regions (henceforth, *instance*) spread along the trace-file and generates a synthetic instance that reports the evolution of the instances. Instrumented and sampled measurements play different roles:

instrumented information delimits regions in the source code. For instance, instrumentation helps to delimit user routines, loop bodies, outlined parallel routines, or any other repetitive region of code.

sampled data are uncorrelated with application behavior. It determines how the performance behavior evolves within the instance it belongs.

The folding projects the collected samples into a synthetic instance preserving their time since the start of their respective instance; so, a sample fired at time T_s within an instance that starts at T_i gets mapped into the representative region at time δ_s, where $\delta_s = T_s - T_i$.

Figure 4.2 provides a visual description of the process in order to help understand it. The top of the Figure depicts a time-line of an application with a repetitive region that has been executed three times during the whole execution whereas the bottom part schematizes the folding results. The analyst instruments the application to determine when each routine invocation begins and ends. These points are represented by black markers and labeled as I_x, where odd and even subindices $x \in \{1, 2, 3, 5, 6\}$ represent entry and exit points, respectively. The analyst also enables a periodic sampling mechanism that freely runs whatever the application activity (in the example, providing measurements every seven units of time). Considering the given Figure, the routine invocations start at times 3, 20 and 37 and end at times 19, 36 and 53, respectively. With respect to the sampling, it has fired nine samples labeled S_0 to S_8, from which only S_1 to S_7 have actually occurred within the instrumented region. The samples are represented by additional markers with colors that correspond to the instance color to ease the description. In the resulting schema,

there is a single instance that summarizes the behavior of the three actual executed instances of the instrumented region. Table 4.1 tabulates the results with respect to the mapped results concisely while showing their respective values for T_s, T_i and δ_s.

As a result of this process, the folding combines information from multiple instances into a single synthetic instance regardless of the sampling frequency used. Therefore the aftereffect of the process is an increase of the volume of performance measurements present in the synthetic instance without adding instrumentation points or increasing the sampling frequency, therefore not incurring on a large overheads. At a constant sampling rate, the increase of the volume is directly related to the number of instances folded. A user may execute its application with more time-steps or even take advantage of parallel executions, where multiple processes execute the same region of code, in order to increase the number of instances to be folded. The folding mechanism as described up to this point requires manual intervention from the user to delimit the repetitive regions in the trace-file. However, the end of this chapter offers a description of a framework that provides very detailed metrics without requiring manual instrumentation. This framework helps following a methodology that allows the analyst to understand and even improve the most dominant compute regions within the application.

Truth to be told, practice shows that executions are non-deterministic even using dedicated systems and applications face interferences each run, which results in slight performance variations along the execution. The folding includes a preliminary step to filter out the perturbed instances by considering the duration of the iteration as a normally distributed variable. This step discards from the process those instances that lie outside the range defined by $\mu \pm X \times \sigma$, where μ and σ refer to the mean and the standard deviation of the duration of the instances. Although the implementation automatically sets up a value that keeps instances with a duration within the interval of confidence of 95%, the user may provide a different value for X. For instance, values 1.0, 1.5 and 2.0 keep those instances with a duration within the interval of confidence of the 68%, 86% and 95%, respectively. Still, removing outlier instances does not result in every instance taking the same amount of time to execute. After the outlier removal, the folding normalizes the duration of every instance in the range defined in [0..1] so as to ensure that every instance lasts the same.

It is convenient to remember that performance measurements include data metrics that cover a broad spectrum of information, such as performance counters, energy measurements and call-stack and memory references. Each metric has its own semantics and characteristics; so, they must be treated independently with different approximations. The forthcoming sections describe the data processing and representation to provide the analyst with insightful data that depict the evolution of the application. Despite the differences between the semantics of the captured information the results are shown in similar plots. The results provided by the folding span across time of the representative (or synthetic) instance on the X-axis and the data represented on the Y-axis (even in the secondary Y-axis [on the right]) depend on the nature of the metrics.

4.3 Detailed performance counters evolution

Performance counters present two characteristics with respect to measurement. First, their value are monotonically increasing, which means that

$$\forall x, y \in \mathbb{R} : x \leq y \implies C(x) \leq C(y) \tag{4.1}$$

where $C(t)$ is the value of a counter at time t. Second, their value is continuously updated at every clock cycle, therefore they appear to be continuous along time.

There are several performance tools (such as TAU [194], HPCToolkit [210], Extrae [66], to name a few) that capture the necessary information through sampling and instrumentation mechanisms that may apply to the folding mechanism. For the particular processing of the performance counter values in the folding mechanism, the counters need to represent cumulative values. For instance, the Extrae instrumentation package reads the performance counters at instrumentation and sampling points, although the value of the counters represent the number of events since the last read. Consequently, to provide the cumulative counter value ($Counter_s$)

Table 4.2
Tabulated exemplification of the folded results where samples include values of a performance counter.

	Instance #1		Instance #2			Instance #3	
	S_1	S_2	S_3	S_4	S_5	S_6	S_7
T_s	7	14	21	28	35	42	49
T_i	3	3	20	20	20	37	37
δ_s	4	11	1	8	15	5	12
$Counter_s$	4	17	1	8	33	5	21

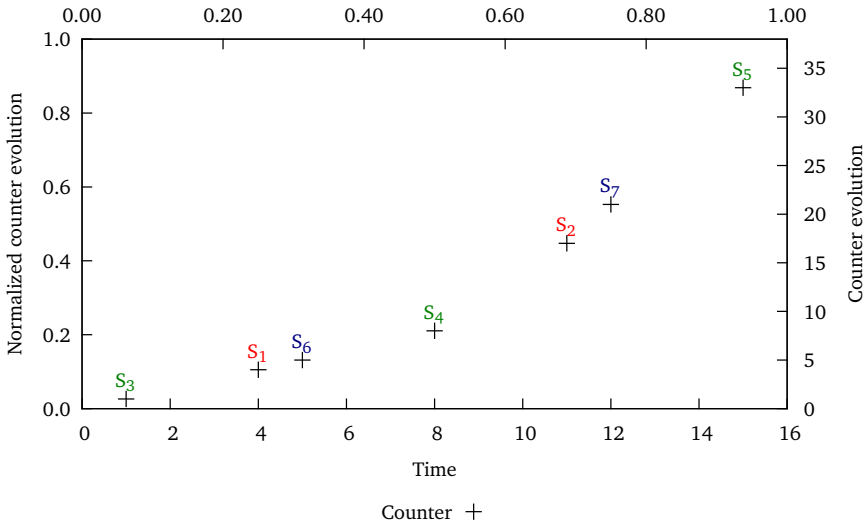


Figure 4.3
Graphical representation of performance counter data from Table 4.2.

it is necessary to convert the readings of the performance tool first so that measurements honor Equation 4.1.

Following the initial example, Table 4.1 has been extended so that each sample contains the cumulative value of an indistinct performance counter since the start of the instance, as depicted by the last row in Table 4.2. It is also possible to represent this information in a plot, as seen in Figure 4.3. The Figure represents the evolution of $Counter_s$ with respect to δ_s and the picture shows that the performance counter values at folded sample points represent an increasing function, even though at this point the result is only a set of discontinuous values. The following step in the folding mechanism focuses on constructing a continuous signal along the synthetic region using an interpolation mechanism. There are several approaches to interpolate the values within a delimited range:

piece-wise constant interpolation is the simplest interpolation method and consists of finding the closest data value at a given time.

linear interpolation uses linear polynomials to guess the returned value between two known points. For instance, for the known points (t_0, c_0) and (t_1, c_1), a value at time t in the interval (t_0, t_1) the value c along the straight line follows the equation:

$$\frac{c - c_0}{t - t_0} = \frac{c_1 - c_0}{t_1 - t_0} \quad \Longrightarrow \quad c = c_0 + (c_1 - c_0) \times \frac{t - t_0}{t_1 - t_0} \tag{4.2}$$

piece-wise linear interpolation partitions the independent variable into intervals and each separate segment is fitted using a linear interpolation.

polynomial interpolation generalizes the linear interpolation by using a higher degree polynomial that connects all the points. For instance, the interpolation polynomial may look like the following function:

$$c(t) = a_n \times t^n + a_{n-1} \times t^{n-1} + \cdots + a_1 \times t + a_0 \tag{4.3}$$

spline interpolation is a form of interpolation that uses low-degree polynomials to connect the intervals and choose the polynomial so they fit together in such a way that the function is derivable at every point.

Gaussian processes interpolation is a non-linear interpolation mechanism where realizations consist of random values associated with random variables with a normal distribution.

This thesis evaluates the use of interpolation mechanisms derived from a Gaussian process regression and piece-wise linear regressions, as depicted in the plots of Figure 4.4. Gaussian process regressions are governed by prior calculated covariances, as opposed to splines which are chosen to optimize smoothness of the connected values. On the other side, piece-wise linear (or segmented linear) regressions study the independent variable looking for segments to express the variable as separate linear regressions on each segment. In the aforementioned Figure, it may be observed that the use of the first type of interpolation results in a smooth curve that adjusts to every point. However, the second type of interpolation connects every point through simple linear regressions (in this particular example, two segments with a break-point in between them).

It is important to note that not every instance needs to account for the same value for a counter across instance. Variations across instances occur as a consequence to different control flows, subtle differences in the processor state every time the computation region begins and even non-deterministic behavior of the performance counters in modern processors [229]. Similarly to the aforementioned process that normalizes the instance duration, the performance counter values are normalized so that each value for a particular counter within an instance lies within the range [0..1]. A caveat here, the normalization process needs the performance counter values at begin and end points, so it is necessary for the monitoring system to capture these metrics in addition to the metrics at sample points. For clarification purposes, Figures 4.3 and 4.4 show on the left Y-axis the normalized accumulated value and the right Y-axis the accumulated value, but henceforth any performance counter folding plot will only show the normalized accumulated value (on the left Y-axis).

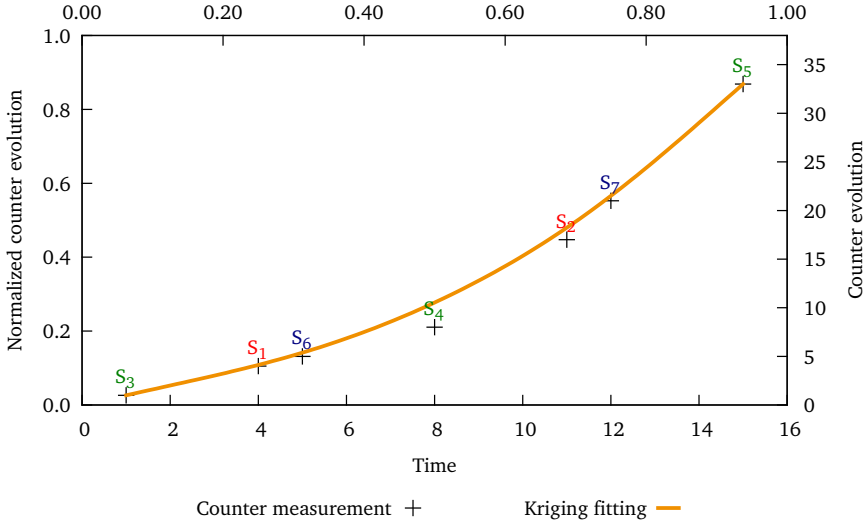
4.3.1 Kriging fitting

The first study focuses on the Gaussian interpolation process known in the geo-statistics community as Kriging [221]. Even though it was originally developed for applications in geo-statistics, it is a general method used in other disciplines such as environmental science [237], hydrogeology [12] and remote sensing [167]. When used in geo-statistics, Kriging is typically used to in the spatial data field such as mineral resource and reserve valuation so it applies to cartographic (2D) data and estimates the values of unsampled locations. The method used in this thesis simplifies this approach because the Kriging fitting applies to one dimension (time) and estimates the value of performance counters in between captured samples.

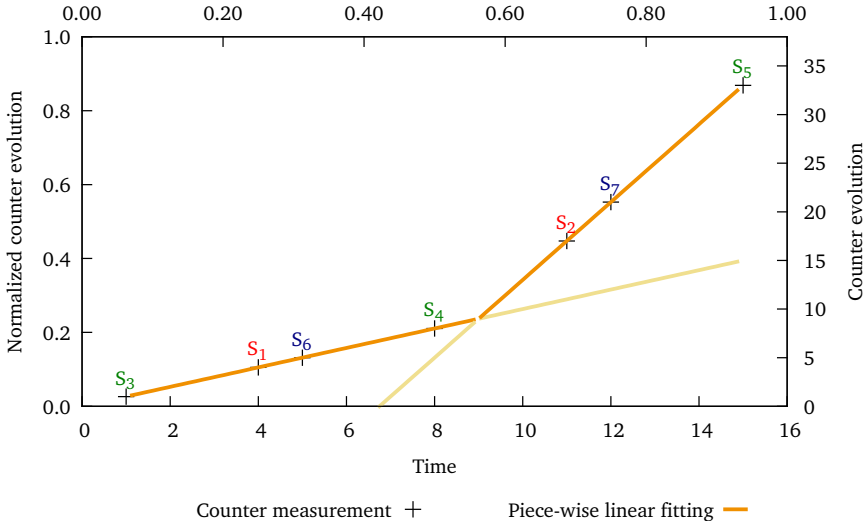
Though there are several variants for the Kriging estimators, all of them derive from the basic linear regression estimator $\hat{Z}(x_0)$ at point x_0 defined as [87]:

$$\hat{Z}(x_0) - \mu = \sum_{i=1}^{N} \lambda_i \left(Z(x_i) - \mu(x_0) \right) \tag{4.4}$$

where $Z(x_i)$ refer to the measured values at the i-th location, μ and $\mu(x_0)$ are the expected values (means) of $Z(x_0)$ and $Z(x_i)$, λ_i is an unknown weight for the measured value at the i-th



(a) Using a Gaussian process-based fitting.



(b) Using piece-wise linear regressions.

Figure 4.4
Alternatives to fit hardware counters folded data using data from Table 4.2.

location as well, x_0 is the predict location and N is the number of given values. The goal is to determine the weights (λ_i) to minimize the variance of the estimator

$$S^2(x_0) = Var\{\hat{Z}(x_0) - Z(x_0)\} \tag{4.5}$$

under the unbiasedness constraint:

$$E\{\hat{Z}(x_0) - Z(x_0)\} = 0 \tag{4.6}$$

The weights are based on the distance between the measured points and the prediction location, but also on the overall spatial arrangement of the measured points. In addition, the Kriging interpolation includes a set of parameters that determine the fitting. Of all these note the nugget parameter, which determines how strictly the fit follows the individual points on the neighborhood. The value for the nugget parameter in the Kriging implementation used in this thesis defaults to 10^{-4}, although sometimes it is necessary to increase this value because the results may exhibit the nugget effect [105] (i.e. oscillatory artifacts when interpolating distant points with respect to performance metrics but very close in time).

Listing 4.1
Summarized code for the Stream benchmark.

```
1  for (i = 0; i < NTIMES; i++)
2  {
3    for (j = 0; j < Ncopy; j++)
4      c[j] = a[j];          /* Copy */
5    for (j = 0; j < Nscale; j++)
6      b[j] = s*c[j];        /* Scale */
7    for (j = 0; j < Nadd; j++)
8      c[j] = a[j] + b[j];   /* Add */
9    for (j = 0; j < Ntriad; j++)
10     a[j] = b[j] + s*c[j]; /* Triad */
11 }
```

Figure 4.5 illustrates the plot generated by the folding mechanism when applied to the instruction counter on a trace-file of the Stream [141] benchmark. Listing 4.1 shows the main loop of the benchmark, which traverses four kernels named copy, scale, add and triad. For the particular execution described here, the external loop trip counts $NTIMES$=100 and the internal loop trips are N_{copy}=N_{scale}=N_{add}=N_{triad}=10,000,000. The application has been instrumented to delimit the loop body as the instance to be folded and the execution has been sampled at 50 Hz on a Intel®Core™i7 running at 2.4 GHz. Each red cross in the Figure corresponds to the cumulative value of the corresponding hardware counter (graduated instructions) since the start of the instance associated with a folded sample when mapped into the synthetic instance. So, one red cross at position (X, Y) represents one sample that occurred at time X from the origin of the instrumented region and has executed Y instructions so far. The contouring Kriging algorithm connects all these points and its results are shown using a green line in the plot. The mechanism also calculates the derivative of the contouring results and shows them in blue within the plot using the right Y-axis as a reference (which in this case represents the MIPS rate since the analyzed metric refers to instructions). Also, the plot also displays a gray marker on the right indicating the average counter rate and some metrics at the top regarding the average duration and the average number of events (instructions here) along the instances folded.

The most prominent information to extract from this Figure is that the MIPS rate clearly exhibits four phases. This insight is likely to indicate that the application traverses four uniform regions of code where the performance differs (matching with the application structure), yet the correlation between performance and source code will be analyzed later in detail in Section 4.5. The results allow being much more precise when reporting the results: the first phase starts at

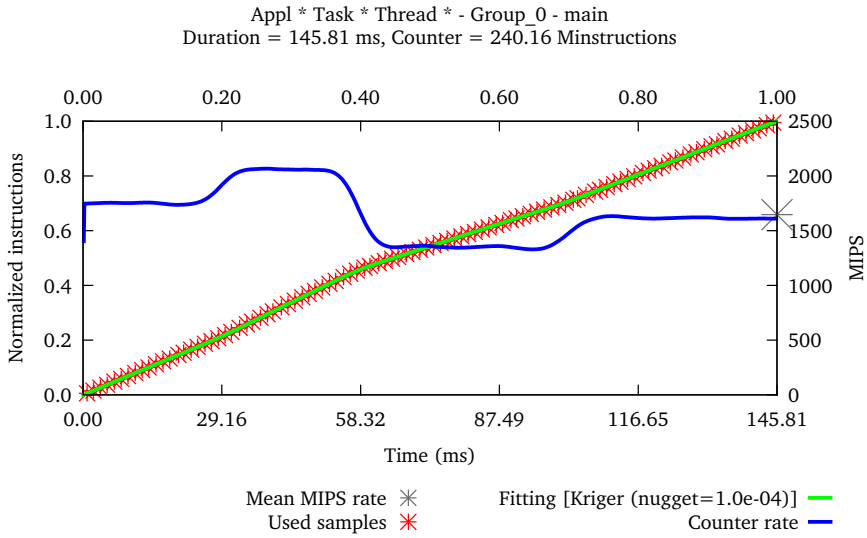


Figure 4.5
Folding results for the instruction counter using Kriging interpolation on the Stream benchmark.

Table 4.3
Experiment setup for the quality study.

Application name	**MHD**		**bt.B**	
Sampling mode	*Detailed*	*Coarse*	*Detailed*	*Coarse*
Sampling period (in cycles)	1 M	1,000 M	50 K	10 M
Samples per second	1,600	1.6	32,000	160
Sampling overhead	4%	<1%	89%	2%
Number of iterations	1	100	1	200
Total number of samples per task	2,870	260	5,661	5,445

the beginning of the region and lasts about 20% of the duration (approximately 14 milliseconds [ms]) and executes at 3,600 MIPS. Similarly, subsequent phases last 13, 18 and 19 ms and execute at 4,250, 3,300 and 3,800 MIPS, respectively, therefore the folding reports phases smaller than the sampling period (20 ms). In conclusion, compared to other performance tools that only summarize the performance metrics between instrumentation or sample points, the folding mechanism expresses the instantaneous evolution of performance within a region delimited by instrumentation.

4.3.1.1 Validation

So as to validate and study the quality of the folding mechanism, this work proposes a twofold experiment. The evaluation compares the resulting trace-files of two executions in which the applications have been sampled with high and low frequencies. The experiment uses the bt.B parallel benchmark from the NAS MPI Parallel Benchmark Suite [10] and MHD, a parallel Lattice Boltzmann magneto-hydrodynamics application, on an SGI Altix machine with Intel®Itanium®2 9030 processors running at 1.6 GHz. Since the results of the contouring algorithm are sensitive to the number of folded samples, the test focuses on validating how similar are the interpolated results from both fine-grain sampling with respect to the folded data. Table 4.3 provides all the details regarding these executions.

The first experiment aims at comparing the shape of the hardware counter metrics in a high-frequency sampled trace with the shape obtained by using the folding on a low-frequency

sampled trace. In this study, only one time-step of the high rate sampled trace-file is used as a reference and afterwards, it is compared with the resulting folded data. Also, the comparison uses several performance counters, namely committed instructions, executed floating-point operations and branches and L2 cache accesses, hits and misses; and compares how these counters differ between the detailed and the folded results. Finally, to prevent the results being affected by the indeterminism introduced by MPI calls (due to network congestion, for instance), the comparison applies to the most time-consuming computation regions found on each binary, that is the region from an MPI call to the following MPI call. Furthermore, to disregard the expense experienced in the fine-grain execution, the average distance is calculated between the normalized results. In order to facilitate the reading, each computation region is named according the routine to which it belongs plus a suffix that indicates how many MPI calls have occurred since entering the routine.

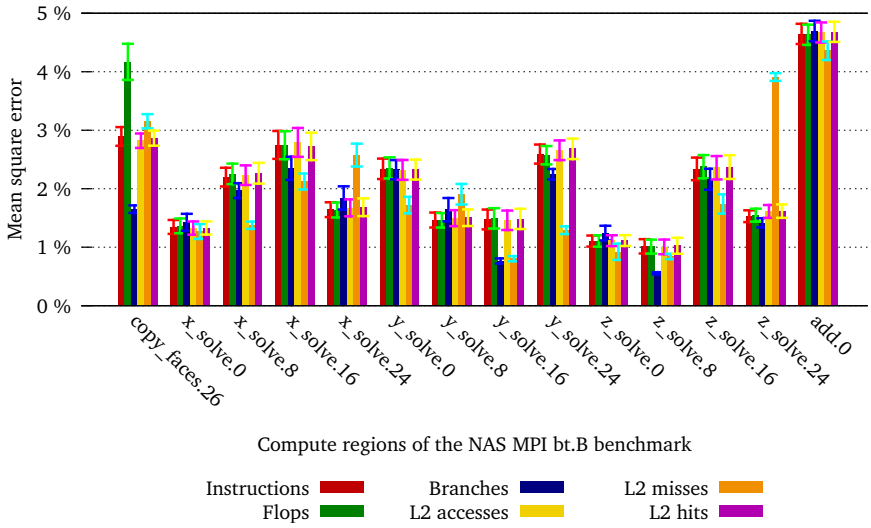
Since the Kriging implementation used in this thesis returns a vector of equidistant points (in terms of time) and not as a function of time, it is not possible to obtain the value of the contouring function at any arbitrary point. Consequently, the results between the samples of the reference time-step and the folded samples by interpolating are both sampled at the same rate. Then the absolute mean difference and the square mean error are computed for each pair of samples.

Figure 4.6 presents the results of this comparison. The X-axis represents the different computation regions found, whereas on the Y-axis indicates the percentage of variation between the detailed sampling and the folded coarse samples using the bars and the square mean error using the whiskers. The plots show a mean difference up to 5% in bt.B and 0.5% in MHD, no matter which performance counters are compared in every computation region. More specifically for bt.B, the folded coarse grain sampling sums up approximately the same number of samples as the reference iteration because the number of time-steps on the coarse grain execution is the proportion between the two sampling frequencies. As for MHD, the situation is even better. Although the difference between the detailed and coarse sampled frequencies is about three orders of magnitude, using a coarse-grain sampling and executing 100 time-steps suffices to get similar results. The conclusion drawn from these results indicate that if the application executes long enough and the same code several times then analyst can use large sampling periods and apply the folding process to obtain a good approximation of the internal performance behavior without requiring highly penalizing periods.

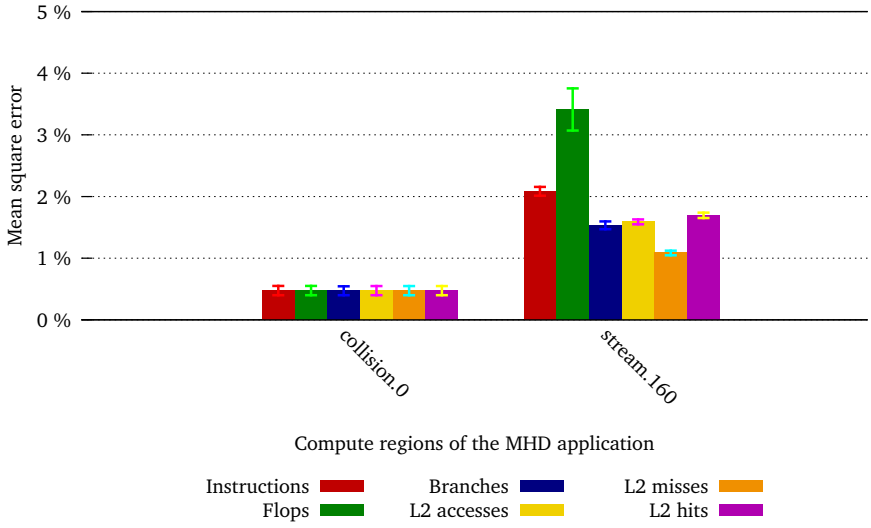
The second test measures how the number of folded samples influences the absolute mean difference. The comparison uses a highly detailed sampled trace-file of the three longest computation regions from the two applications (namely `copy_faces.26` and `x_solve.8` for bt.B and `stream.160` for MHD) and calculates the mean difference by limiting the number of samples used in the folding process. Plots in Figure 4.7 show how the absolute mean difference varies as a function of the number of samples folded in each region. As oneself might expect, these plots indicate that the mean absolute difference decreases as the number of folded samples increase. More specifically, using 100 samples in the folding processes reduces the difference below the 5% for the `copy_faces.26` region. Regions `stream.160` and `x_solve.8` show an even better situation because only 30 and 40 samples provide results differ by under 5% from the highly sampled results, respectively. The results for the region `copy_faces.26` are enhanced when 200 samples are folded (especially on the branch instructions counter) but using more than 200 samples results in a marginal improvement of the results. Something similar occurs in `x_solve.8` and `stream.160`, using more than 40 or 60 samples in the folding process yields small improvements in the comparison. In conclusion, these repetitive applications do not need extremely long runs to get similar results when comparing the results of the detailed sampling to those obtained using folding and coarse grain sampling.

4.3.2 Piece-wise linear fitting

The exploration of the Kriging contouring results in Figure 4.5 raises several questions. First, it becomes evident that the folding helps to expose performance phases within instrumented regions and, by means of example, the aforementioned Figure shows that the MIPS rate traverses four performance phases. Since applications are structured as an iterative sequence of loops

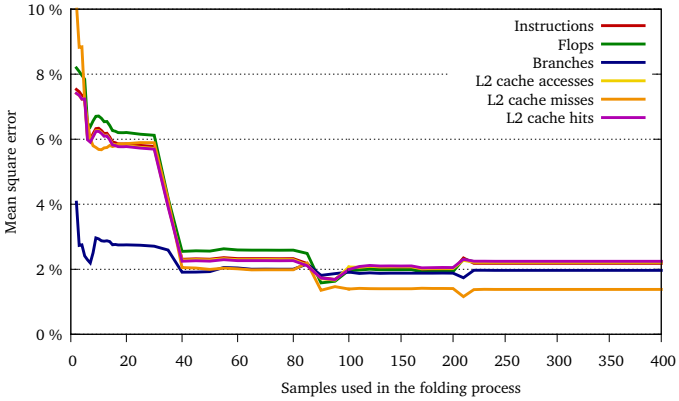


(a) NAS bt.B benchmark.

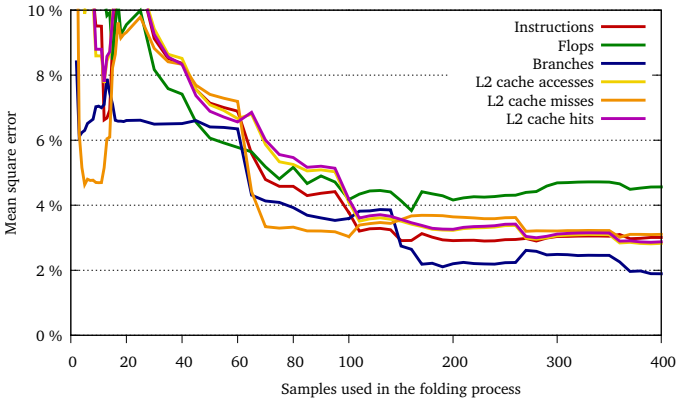


(b) MHD application.

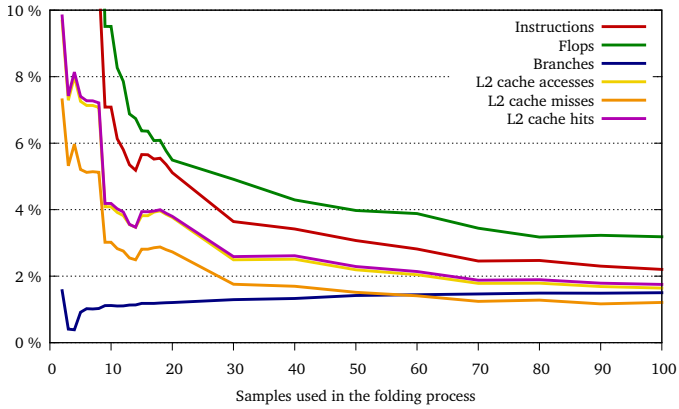
Figure 4.6
Study of the mean square error on different applications comparing high frequency sampling and folded results with coarse grain sampling.



(a) Region x_solve.8 from NAS bt.B benchmark.



(b) Region copy_faces.26 from NAS bt.B benchmark.



(c) Region stream.160 from MHD application.

Figure 4.7
Evaluation of how the number of samples involved in the folding mechanism influences on the mean square error difference of several performance counters in two compute regions when using Kriging interpolation.

and routines, it would be helpful to determine when an performance change occurs, but the Kriging process does not provide this information without further post-processing. Second, the Kriging results expose oscillatory artifacts that resemble noise within these phases (nugget effect). However, in ideal circumstances, the processor is likely to execute the phases at a uniform rate and these can be modeled using linear regressions. Finally, while there exists a transition between phases necessarily, the transition using the Kriging algorithm is mitigated as a result of the selection of a relaxed nugget parameter. Selecting a stricter value for the nugget parameter would generate sharper transitions but increase the nugget effect.

This thesis evaluates the use of piece-wise (or segmented) linear regressions to address these issues when used as a fitting mechanism for the folded data. Piece-wise linear regression is a method aimed at detecting changes in time-series (named break-points, but to avoid confusions with debugging techniques they will henceforth be referred as phase breaks in this thesis); it has been used in bio-medicine [158] as well as financial [153] and ecological [219] studies, among many others. By way of introduction to these regressions, consider the general linear regression model that is expressed as

$$y_i = x_i^\top \beta_i + u_i \qquad (i = 1, .., n) \tag{4.7}$$

where at time i, y_i is the observation of the dependent variable and x_i, β_i and u_i are the explanatory variables, regression coefficients (including the intercept) and error terms, respectively. Segmented linear regression tests whether the regression coefficients remain constant or vary over a time-series, which in the current scope is defined by the folded samples. In general, an unknown number of phase breaks (m) exist and they indicate when the coefficients shift from one regression to the next, which is to say in the environment of this thesis, *when* the performance rate changes within the folded results of the instrumented region. Consequently, there are $m+1$ segments (or phases) in which the regression coefficients become constant and allows rewriting Equation 4.7 as:

$$y_i = x_i^\top \beta_j + u_i \qquad (i = i_{j-1} + 1, .., i_j, j = 1, .., m + 1) \tag{4.8}$$

where j is the segment index (so the range defined in $[i_{j-1}+1, i_j]$ can be approximated by a linear regression) and $i_1, .., i_m$ denotes the set of phase breaks, where by convention, $i_0 = 0$ and $i_{m+1} = n$.

The work described in this thesis uses the *strucchange* library [235] from R statistical package [173] to estimate the optimal phase breaks in regression relationships by using generalized fluctuation tests and minimizing the residual sum of squares in Equation 4.8. Trials on the structural change are concerned with testing the null hypothesis (H_0) that is defined by no structural changes along the temporal series, or $H_0 : \beta_i = \beta_0$ for any i. If the process crosses the boundaries defined by the probability of rejecting the null hypothesis, then the fluctuation should be considered improbably large, and H_0 should be rejected. Finally, the process applies linear regressions to calculate the slope (which refers to the counter rate in this case) of the segments delimited.

Figure 4.8 illustrates the results of the folding mechanism when using piece-wise linear regression on the same trace-file used to generate Figure 4.5. The way to read the plot is similar to earlier plots with a slight extension because it provides information regarding the phase breaks. The value of the instruction counter in folded samples (depicted as red crosses) exposes the evolution from the start of the instrumented region. The mechanism estimates three phase breaks (for a total of four phases) within the counter associated with to these folded samples at approximately 14, 28 and 46 ms since the start of the loop. The segmented linear regressions are shown in green above the red points using the left Y-axis, while the slope of the linear regression (the MIPS rate) is shown using a blue line on the right Y-axis. The results obtained through this method are compared with those obtained with the Kriging contouring algorithm depicted in Figure 4.5 on page 65. Both results are similar in terms of identifying the achieved counter rate and exposing the behavior of the instrumented region. The most notable differences between the two approximations include: first, the piece-wise linear regressions automatically detects the phase breaks; and second, the Kriging contouring results provides a continuous results even in the phase breaks.

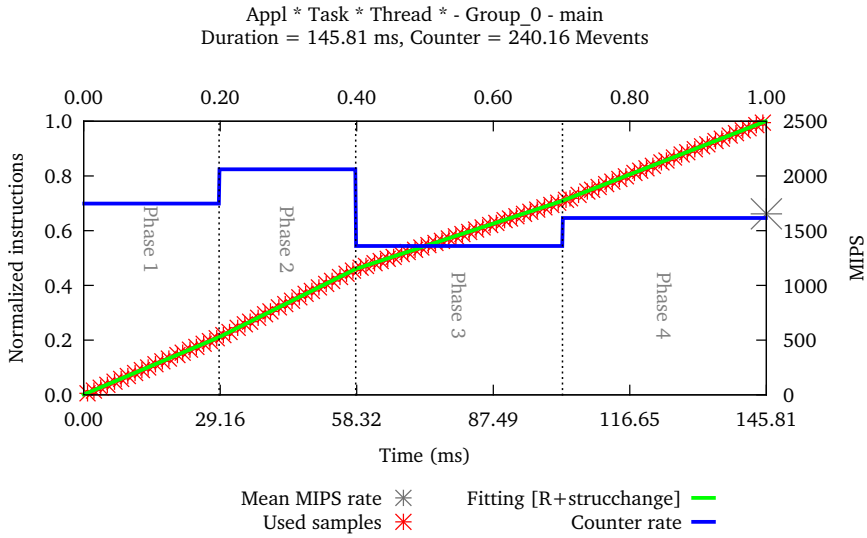


Figure 4.8
Folding results for the instruction counter using piece-wise linear regressions on the Stream benchmark.

4.3.2.1 Validation

Several well-known benchmarks from the NAS benchmark suite as well as the Stream benchmarks help to test the folding mechanism using piece-wise linear regressions in a similar way to that of the validation of the folding using the Kriging contouring approach. The validation comprises two parts which include: first, a comparison between detailed sampling and folded coarse grain sampling when varying the number of folded samples; and second, a comparison of the results obtained using the Kriging and the piece-wise linear regression approaches.

In order to compare the results obtained using the folding with those obtained through detailed sampling, the benchmarks (namely is.B, ft.A, bt.A and Stream) are executed using two sampling frequencies: coarse-grain (50 Hz) and fine-grain (5,000 Hz) on an Intel®Xeon®E5 running at 2.60 GHz. Both trace-files are folded using piece-wise linear regressions and then the resulting interpolations are sampled so as to calculate the difference of the shapes on four performance counters (instructions, branch instructions, L1 data-cache misses and stalled cycles). In addition, the number of samples used in the coarse-grain sampling varies so as to determine how the number of folded samples influences the results. Again, to dismiss the sampling overhead experienced in the fine-grain execution, the average distance is calculated between the normalized results.

Figure 4.9 depicts the evolution of the distance between the detailed and coarse-grain folded results on the four performance counters and varies the number of samples used. It is noticeable that the benchmarks show similar shapes between the detailed sampled executions and the folded coarse-grain sampled executions, with differences below 2% when there are more than 50 samples in the representative region. The only exception to this observation is the branch instruction counter for the bt.A benchmark, which shows a difference about 6%. The difference between the detailed and folded cases in this benchmark is due to the presence of very short phases that execute a large number of branches, which results in a small number of samples and inhibits the *strucchange* implementation to detect the phase breaks and therefore summarize consecutive phases with one linear regression.

In comparing the results using piece-wise linear regressions with those obtained using the Kriging approach, it is worth remembering that one of the parameters of the latter algorithm determines how tight the fit follows the individual points on the neighborhood (acting as a low-pass filter). This effect is illustrated in Figure 4.10 by comparing the folded instruction rate

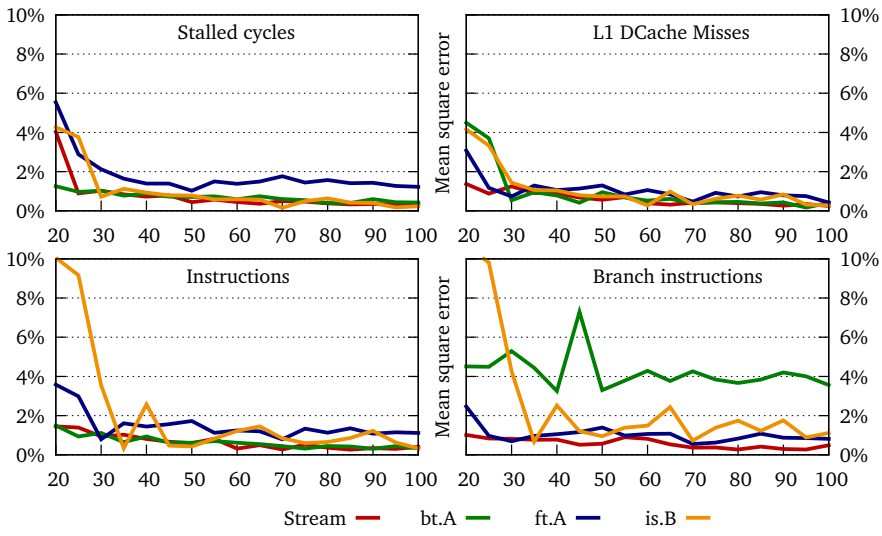


Figure 4.9
Evaluation of how the number of samples involved in the folding mechanism (shown in the X-axis) influences on the mean square error (Y-axis) of several performance counters in two compute regions when using piece-wise linear regressions.

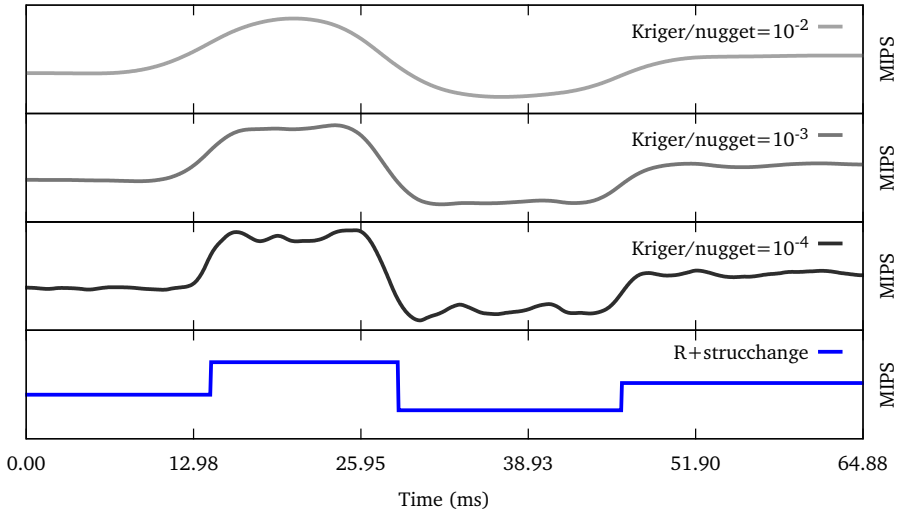


Figure 4.10
Visual comparison of the results obtained using Kriging (and several nugget values) and piece-wise linear regressions for the Stream benchmark.

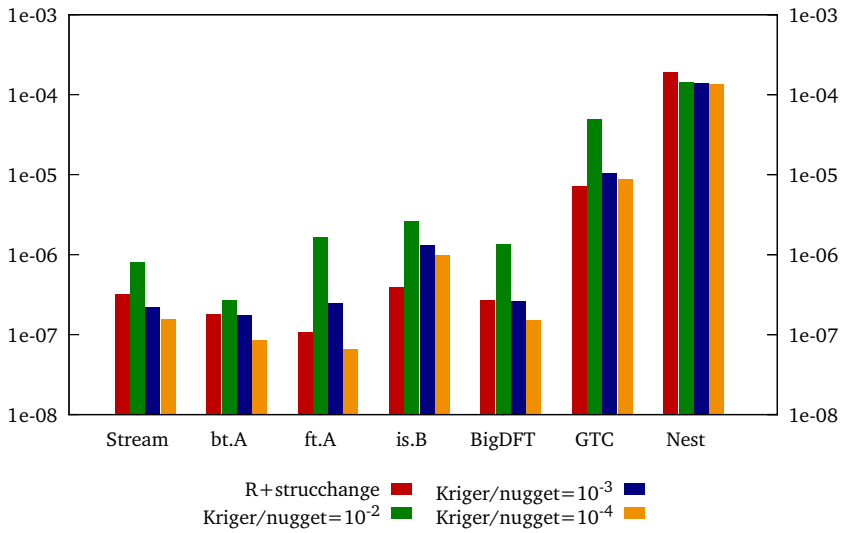


Figure 4.11
Comparison of Mean Square Error obtained using Kriging and piece-wise linear regression on different applications.

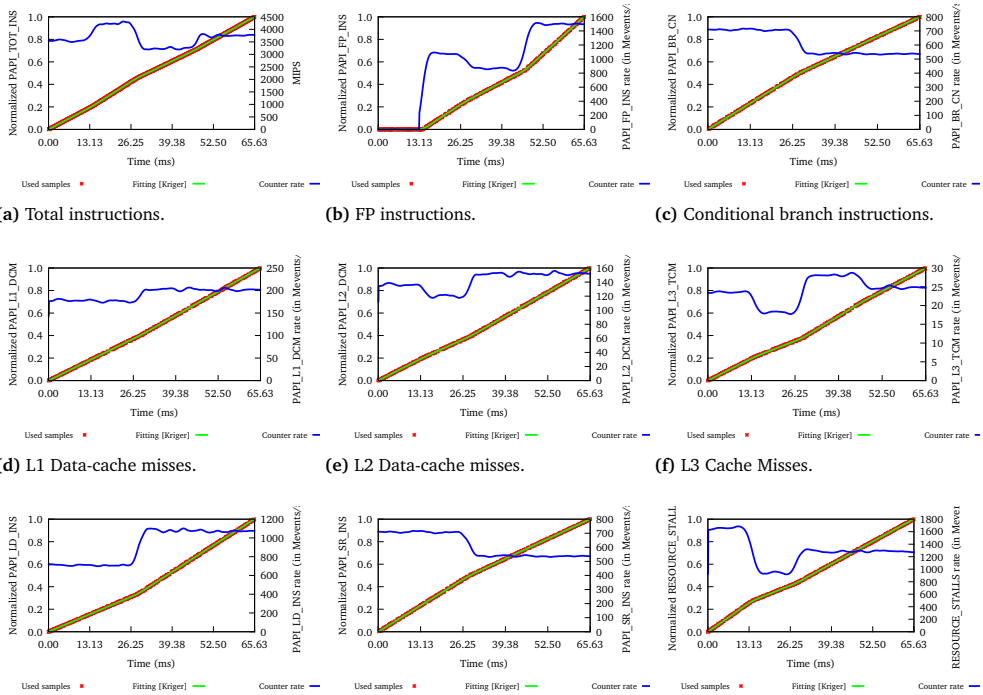
obtained using the piece-wise linear regression (in blue) and the results are calculated using the original Kriging interpolation for several values of nugget (in different gray scales). It may be observed two situations in the plots: a large nugget value results in loosely fitted results and artificial curves near a phase change, while a small value provides fitted results, but at the cost of additional noise due to the variability in terms of instructions among instances and the nugget effect.

Figure 4.11 shows a plot that compares the results from the two explored interpolation methods. The plot shows the mean square error for the Kriging with different nugget values and piece-wise linear regressions for the Stream benchmark, a selection of the NAS parallel benchmarks and the main computation regions of several in-production applications named BigDFT [79], GTC [129] and Nest [81]. The plot indicates that the error from the piece-wise linear regression is similar to the error from the Kriging and the error is application dependent. With respect to the last point, it is worth noting that while the error is related to the variability experienced on the folded instances, the folding results still provide very useful insight.

4.4 Correlation of multiple performance counters

There are several measures to identify the presence of performance bottlenecks in an application. For instance, the introduction of this thesis refers to the floating point operations (Flops) rate which is heavily tailored to measure the performance of mathematical codes. Still, when exploring the performance of an application it is convenient to consider the instruction (MIPS) rate given the fact that not every HPC application is floating-point intensive and that it accounts for all the instruction the processor actually executes. Some people consider this as a weakness in the MIPS rate as there are many instructions in application binaries that do not produce effective work. However the processor has to execute every instruction it encounters in the instruction stream, regardless of whether or not they result in effective work. All the analyses in the forthcoming chapter rely on this metric to evaluate the application performance and compare it to the processor's peak performance.

Even though the exploration of the MIPS rate helps to identify the presence of bottlenecks in the application, this counter rate alone does not help to characterize the nature of the bottleneck.



(a) Total instructions. **(b)** FP instructions. **(c)** Conditional branch instructions.

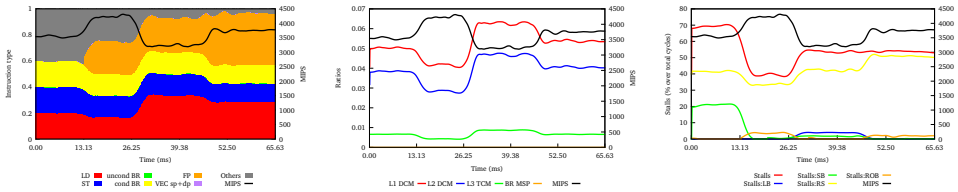
(d) L1 Data-cache misses. **(e)** L2 Data-cache misses. **(f)** L3 Cache Misses.

(g) Load instructions. **(h)** Store instructions. **(i)** Stalled cycles elapsed.

Figure 4.12
Multiple performance views from the Stream benchmark. Each figure shows the progression of a particular performance counter within the main loop of the benchmark.

Other metrics therefore need to be gathered during the application run to contrast results from different counters and understand the nature of the bottlenecks. Consider the results for several performance counter rates shown in Figure 4.12 for the Stream benchmark to exemplify this approach. Interestingly, the MIPS rate for this benchmark (Subfigure 4.12a) shows four phases within the progression of the delimited region (the main iteration loop). However, not every performance counter rate shows the same number of phases. For instance, the rates for the conditional branch instructions (Subfigure 4.12c), the load instruction (Subfigure 4.12g), the store instruction (Subfigure 4.12h) and L1 Data-Cache misses (Subfigure 4.12d) only expose two performance phases in the delimited region. On the other hand, the L2 Data-Cache misses (Subfigure 4.12e) and the stalled cycles (Subfigure 4.12i) show three performance phases. Finally, the floating point instructions (Subfigure 4.12b) and L3 Cache misses (Subfigure 4.12f) do expose four phases. The L3 Cache misses rate may explain (inversely) the instruction counter rate in the middle phases and it may be observed that the four phase achieves a higher L3 cache miss rate than the first phase and that the instruction counter on the first phase is higher than the first phase.

With respect to the collection capabilities, the Performance Monitoring Unit (PMU) is the component of the processor responsible for counting the events associated with the hardware activity and this component imposes certain restrictions on the number and which performance counters that are read at the same time. There are several multiplexing techniques available for alleviating this problem. For instance, the simplest approach tackles this problem by executing the application as many times as performance counters needed and extrapolating the ratios by analyzing the results of every execution. However, it is possible to obtain all the desired counters at once by periodically changing the counter groups during data measurement (either



(a) Instruction mix. **(b)** Architecture impact. **(c)** Stalled cycles distribution.

Figure 4.13
Three different perspectives of the evolution of the main iteration of the Stream benchmark.

by using time-outs or counting the number of collective operations) because of the iterative nature of typical parallel scientific applications. With sufficient coverage, it is possible to collect the counters for each repetitive region and extrapolate their values in only one execution [86]. The major benefits of this approach is that it saves computing resources (and energy), reduces the amount of time to get the results and reduces the variability introduced by dissimilar executions.

4.4.1 Applicability to performance models

Since the folding results are a function of time, it is possible to apply simple operations between them, such as:

$$L1Dperinstruction(t) = L1Drate(t)/MIPS(t) \tag{4.9}$$

in which at time t, $L1Dperinstruction(t)$ represents the L1 D-cache misses per instruction ratio and $L1Drate(t)$ and $MIPS(t)$ refer to the L1 D-cache miss and instruction rates, respectively. Using the counter ratios instead of counter rates helps when comparing several counters at the same time because they are normalized by the instruction rate divisor. These ratios help on the correlation of multiple performance counters in one go. While calculating ratios between performance counter rates and the instruction rate helps to identify the nature of the bottlenecks, it is worth remembering that modern processors provide hundreds of performance counters. The folding mechanism implements another mechanism to describe the models based on performance counters mentioned in Section 3.5.1 (page 48). This functionality extends the folding results because it avoids the need for the analyst to dig into all the available performance counters (and their description) and focus on a bunch of performance metrics instead of searching into multiple plots to locate correlations between counters. For instance, Figure 4.13 uses multiple performance counters in order to offer three different perspectives of the evolution of the main iteration of the Stream benchmark when executed on a system that uses a Intel®SandyBridge processor. In every plot of the set, the MIPS rate is shown in black on the right Y-axis, while the rest of counter ratios are shown on the left Y-axis. More specifically, Figure 4.13a reports the instantaneous progression of the instruction decomposition, giving insight into the code that is being executed. Figure 4.13b reports several architectural components that may limit the processor performance (such as several levels of the memory cache hierarchy and the branch predictor accuracy). Finally, Figure 4.13c provides information on the blocks of the processor responsible for stalled cycles injected into the processor pipeline.

Figure 4.14 shows another example of combining multiple performance using a performance model into a single plot, instead of multiple plots as in the earlier example. The Figure shows the evolution of a compute region of the CCSM [93] application when executed in a cluster of IBM®PowerPC®970MP processors. The combination of the performance metrics honors the IBM®Power5®CPIStack model (detailed in Section 3.5.1.1 [see page 48]). In the Figure, the left Y-axis refers to the break-down of the CPI according to the model and the right Y-axis refers to the achieved MIPS (and depicted through a black line that traverses the plot). An analysis of the Figure shows that the processor faces several performance phases as the application progresses within the compute region. Exploring the performance from the CPIStack perspective, it may be noted that from the start and during 30 ms, the application suffers from the Load/Store Unit (LSU). Most of the issues relating the LSU derive from cache-misses, instructions rejected and to

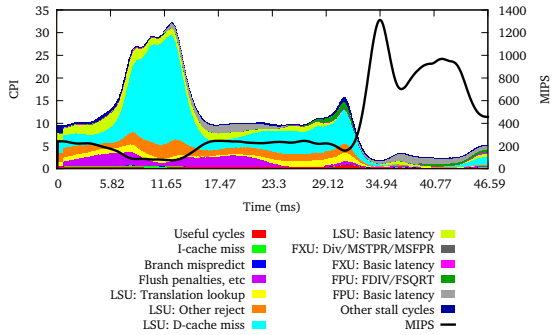


Figure 4.14
Example of applying the IBM CPIstack performance model for the IBM Power5 processor when analysing a compute region within the CCSM application.

a lesser extent, TLB misses. Within this phase, there is a region in time (from 6 to 15 ms from the start) where the performance is extremely low because of cache-misses, showing CPI above 25 (or similarly, running below 100 MIPS at the processor frequency). The second phase starts at 30 ms from the start and lasts until the end of the region achieves better performance, although CPI is above one. This phase is mostly dominated by the floating-point operations although there is an increase of the cache misses at the very end of the region.

4.5 Accurate source code attribution

The folding mechanism also attributes the observed performance to the application source code. Samples must therefore contain source code references that represent the processor call-stack frames at the sample point. Unlike hardware performance counters, source code references cannot benefit from fitting models to detail the evolution of the source code due to the discrete nature of the source code references, though there are alternatives for processing this information.

The first simplistic approach tested to correlate performance and the source code uses the GVIM[1] editor. This approach relies on a manual exploration of the folded trace-file containing the folded results and call-stack information. With this trace-file it is possible to manually filter and then associate the captured call-stack with the folded results, to finally export the correlation of the achieved performance to a given portion of a routine. For instance, Figure 4.15 shows a screen-shot of this editor showing the source code of the Stream benchmark and representing the achieved MIPS rate using a background color that ranges from green (low) to blue (high). The scale and add kernels (second and third loops) therefore achieve the best and worst instruction rates, respectively. While this first approach enables the analyst to better understand the correlation between source code and performance, it presents several drawbacks:

- it relies on a manual analysis and data selection from the trace-file,
- while the coloring provides some information on the metric rate, this approach does not provide the analyst with quantitative results,
- the coloring only refers to one metric but there may be many metrics of interest to the analyst and
- it does not provide temporal a evolution of the performance and the source code.

In order to address all the aforementioned weaknesses, this thesis describes two approaches to conduct the association between performance and source code references. Both approaches

[1] `http://www.vim.org/about.php` - Last accessed March, 2015.

```
222   times[0][k] = mysecond();
223   for (j=0; j<N; j++)
224       c[j] = a[j];
225   times[0][k] = mysecond() - times[0][k];
226
227   times[1][k] = mysecond();
228   for (j=0; j<N; j++)
229       b[j] = scalar*c[j];
230   times[1][k] = mysecond() - times[1][k];
231
232   times[2][k] = mysecond();
233   for (j=0; j<N; j++)
234       c[j] = a[j]+b[j];
235   times[2][k] = mysecond() - times[2][k];
236
237   times[3][k] = mysecond();
238   for (j=0; j<N; j++)
239       a[j] = b[j]+scalar*c[j];
240   times[3][k] = mysecond() - times[3][k];
```

Figure 4.15
MIPS rate shown in the GVIM editor in conjunction with the application source-code.

provide an automated mechanism to correlate multiple performance metrics with the source code and display their temporal evolution. The first approach takes advantage of the phases detected automatically using the piece-wise linear regressions. The mechanism associates the performance of each with the source code loops observed by the top of the call-stack in a phase. The second approach takes into consideration the fact that consecutive folded samples present some temporal and spatial locality, i.e. it is very likely that consecutive samples refer to source code references that are close to each other. This second approach proposes a novel *post-mortem* that grounds on Multiple Sequence Alignment (MSA) algorithms [160, 123, 52] to approximate the duration of the representative routines by grabbing a small, fixed segment of the call-stack.

4.5.1 Using phases determined by piece-wise linear regressions

The fact that most applications are written using a structured programming paradigm means it is conceivable for each loop to show a distinct performance in the folding results, or *vice versa*, that each performance phase corresponds to one loop. This first approach to correlate source code and performance exploits this idea. The approach uses phase breaks obtained from the piece-wise linear regressions when applied to a performance counter. These phase breaks divide the folded synthetic instance into phases that help to attribute independent source code references to the performance.

While there are several performance counters, the current implementation uses the instruction counter to delimit the phases for the source code analysis because the instruction rate is commonly used to determine whether a code runs efficiently or not. However, this can be modified and/or extended to use another performance counter or even a combination of performance counters. Regardless the performance counter used, it is interesting to note that consecutive loops or routines may perform at a similar rate and such cases these loops will be grouped together within a single phase. So as to attribute the performance to a region of code, the process maps the source code lines pointed out by the call-stack of the samples into the application routines or loops. In order to perform such attribution, the process includes AST parsers for C, C++ and Fortran90 codes that delimit the boundaries of routines and loops within the source code. This way, it allows a relationship to be established between one line of code and the loop (or the routine if a loop does not exist) that contains it.

To illustrate this process, consider extending the earlier folding example shown in Table 4.2 (page 59) by adding an additional row (named $CodeLine_s$) that indicates the sampled source code line at time δ_s (shown in Table 4.4). Figure 4.16 shows a time-line that depicts the evolution of the performance counter as well as the line of source code. The plot at the bottom depicts the evolution of the performance counter, while the top part indicates at a given time-stamp the

Table 4.4
Tabulated exemplification of the folded results where samples include values of a performance counter and code line references.

	Instance #1		Instance #2			Instance #3	
	S_1	S_2	S_3	S_4	S_5	S_6	S_7
T_s	7	14	21	28	35	42	49
T_i	3	3	20	20	20	37	37
δ_s	4	11	1	8	15	5	12
$Counter_s$	4	17	1	8	33	5	21
$CodeLine_s$	10	20	11	12	21	11	20

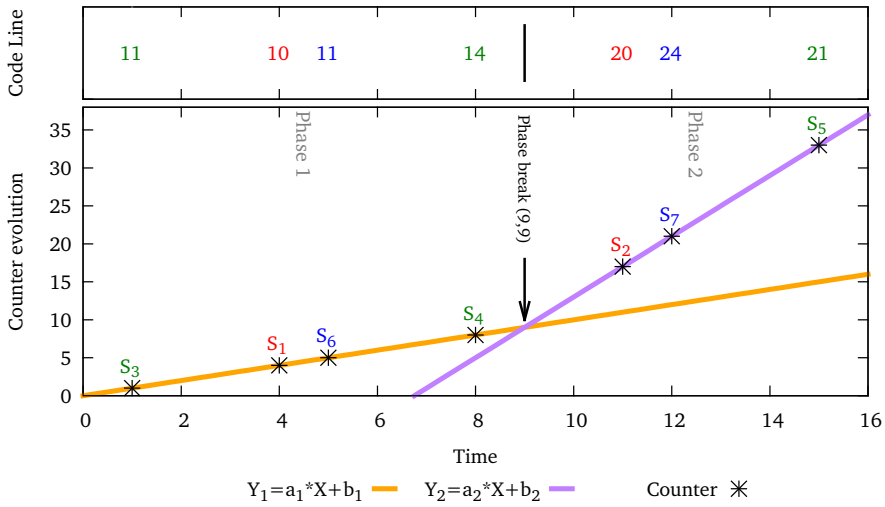


Figure 4.16
Graphical representation from data contained in Table 4.4. The plot at the bottom depicts the counter evolution since the origin of the instance where there are two evident phases with respect to performance. The plot at the top illustrates which source code lines where obtained folded time-stamps of the samples.

observed source code lines. Looking at the plot at the bottom, it becomes clear that two linear models exist (one derived from samples S_1, S_3, S_4 and S_6 and another derived from samples S_2, S_5 and S_7) and that these two linear models intersect at one point. In this case, applying a simple linear regression model to each of the two sets of folded samples (i.e. models expressed as $Y{=}a{\times}X{+}b$) helps to calculate the values for (a_1, b_1) and (a_2, b_2). More precisely, the solution of these values is (1, 0) and (4, -27), respectively and the intersect at point (9,9).

In the previous Figure, the intersection at time nine represents a phase break, so that two phases exist with different performance each. These two phases are processed separately; the performance of each phase is attributed to the source code lines observed in each phase. For instance, the first phase includes samples that observed lines 10-14 and the second phase consists of samples that reference lines 20-24. As a result, the loops (or routines) that cover lines 10-14 and 20-24 will be assigned the performance observed in each phase.

In a more practical way, the attribution process not only has to associate performance of multiple counters with ASTs, but also requires a mechanism to display all the metrics collocated with the source code. Within the scope of this thesis, the Scalasca CUBE source code visualizer has been extended to show the source code correlated with annotations regarding the folded hardware counter metrics, as depicted in Figure 4.17. The tool also allows the user to select the

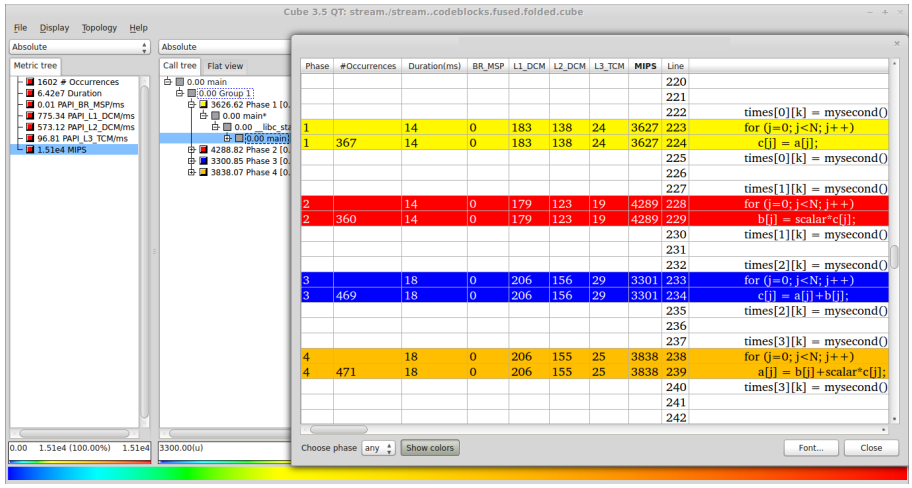


Figure 4.17
CUBE visualizer correlating the Stream source code with branch mispredicts, L1 and L2 D-cache misses, L3 cache misses, and MIPS rates.

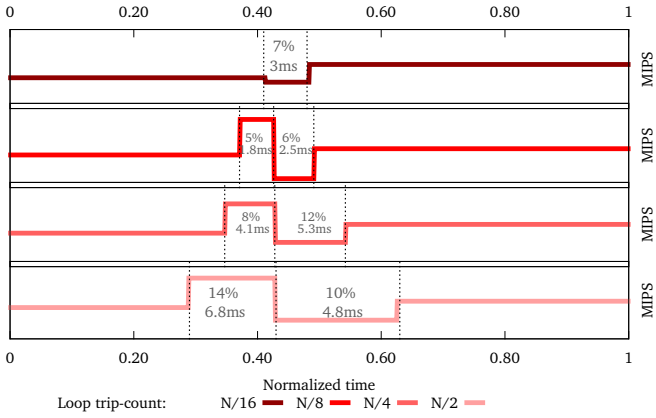
metric for the coloring gradient by clicking on the column of the metric and it shows a profile that indicates the most observed source code lines in the region during the performance data collection.

4.5.1.1 Validation

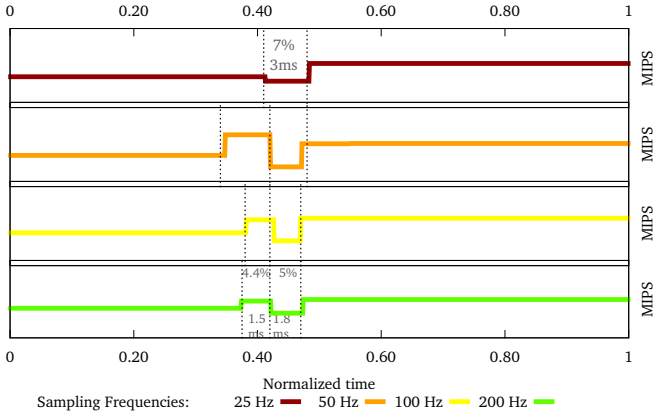
This work uses the Stream benchmark (see Listing 4.1 in page 64) to evaluate the accuracy of attributing the source code and its performance. This evaluation aims at describing how the following three factors influence the phase detection mechanism: 1) the duration of the phase, 2) the sampling frequency and 3) the number of instances folded. The evaluation takes as a reference an execution of the Stream benchmark in which $NTIMES$=200 and N_{copy}=N_{scale}=N_{add}=N_{triad}=10,000,000 and that has been sampled at 25 Hz. The results are then compared to the reference while changing some of the parameters of the benchmark.

Figure 4.18a shows the results obtained on the folded instruction counter rate by progressively reducing (from $N/2$ to $N/16$) the loop trip count of the central loops *scale* (N_{scale}) and *add* (N_{add}) and keeping the sample rate at 25 Hz. The Figure shows that the piece-wise linear regression successfully distinguishes the *scale* and *add* loops when their loop trip count is $N/2$, $N/4$ and $N/8$ and their duration sum up to 11.6, 9.4 and 4.3 ms, respectively. Reducing the trip count to $N/16$ results in a reduction of the number of samples in the central loops and this leads to three effects in the results. First, the loops for *scale* and *add* are still detected and separated, but they are combined into a single phase. Second, the regression generated for the two loops lasts about 3 ms, in contrast to 4.3 ms when the trip count is $N/8$. Finally, the phase break occurs late in time, i.e. the end point of the central phase occurs almost at the same time as the $N/8$ case. These effects are directly related to the fact that the fluctuation tests of the `strucchange` require a number of samples to notice a change in the slope. Therefore, it becomes harder to detect changes of the counter rate if there is a reduction of the number of samples.

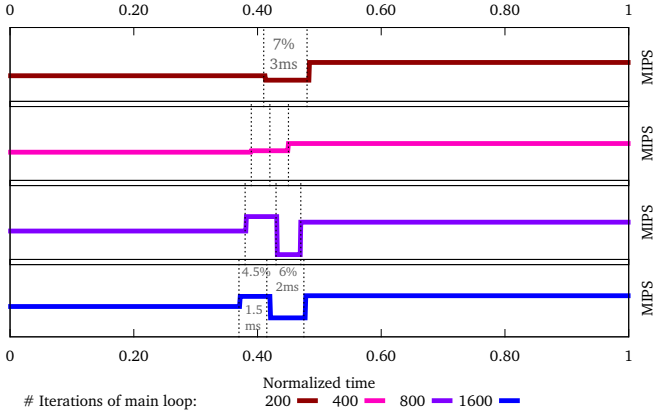
The number of samples must increase to ease the phase detection and this can be achieved either by using a finer sampling frequency (and increasing the overhead) or by increasing the number of folded instances. To increase the number of instances, one may re-execute the application with more time-steps or use instance data of multiple processes in parallel executions. For instance, if analysts change the sampling rate on the version of Stream where the central loops iterate $N/16$ times, they will observe that using a sampling frequency equal or higher than 100 Hz detects four phases, as depicted in Figure 4.18b. Similarly, if the analyst decides to



(a) Reducing the trip count of the central loop and using a sampling frequency of 25 Hz.



(b) Using different sampling frequencies with a trip count $N/16$ in the central loops.



(c) Executing more iterations on the main loop with a trip count $N/16$ in the central loops and keeping the sampling frequency at 25 Hz.

Figure 4.18
Results of the folded instruction counter rate when changing some attributes of the execution of Stream.

Table 4.5
Results obtained through pure instrumentation and `gprof` for the modified Stream and its relationship with phases delimited in Figure 4.18.

	Instrumentation			*gprof*		
Phase	**% Time**	**Self**	**Self/Ntimes**	**% Time**	**Self**	**Self/Ntimes**
1	40.10	2.78 s	13.91 ms	39.52	2.78 s	13.90 ms
2	2.51	0.24 s	1.2 ms	2.28	0.16 s	0.8 ms
3	3.42	0.30 s	1.5 ms	3.28	0.23 s	1.15 ms
4	53.80	3.70 s	18.5 ms	45.80	3.22 s	16.16 ms

execute a longer run on the same version of Stream, they will note that increasing the number of $NTIMES$ ends up in the disclosure of the *scale* and *add* loops as shown in Figure 4.18c.

In addition, the Stream benchmark with a trip count $N/16$ in the central loops has been manually instrumented and also, profiled using `gprof`. The timing results of these executions are tabulated in Table 4.5 and correlate with the phases discovered in the folding results using the piece-wise linear regressions. It may be observed that the time attributed to each phase by either the instrumentation or `gprof` is similar to that of the phase breaks obtained using frequencies over 100 Hz and/or executing 800 iterations on the main loop.

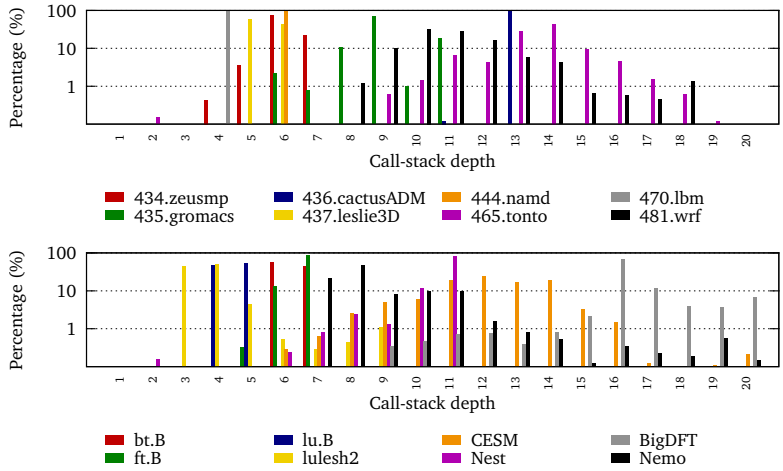
4.5.2 Bio-inspired call-stack reconstruction

The previously stated correlation between the performance and the source code provides performance metrics from the source code point of view; it needs a mechanism to determine the phase breaks, however, and it does not naturally expose the time progression. This thesis proposes an additional for such a correlation without searching within the performance space and exposing the source code time progression. This approach takes advantage of the sequence of the folded samples to determine which portions of the code are being executed according to their call-stack references.

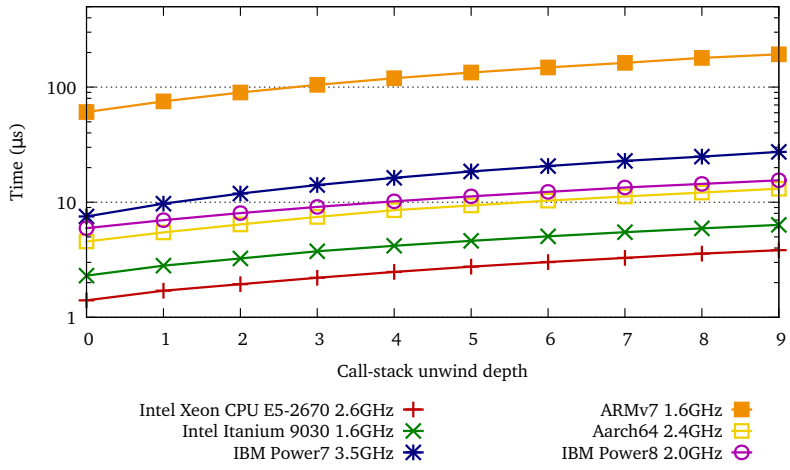
In the context of this thesis, libunwind is used to capture the call-stack information at sample points. While libunwind already provides a *fast unwind* facility on certain systems[2], it is inherently a slow operation that impacts on code performance significantly because it traverses multiple stack frames. To capture the full calling context, the frame inspection needs to unwind every stack frame, resulting in a non-negligible and variable overhead. To give an example, Figure 4.19a shows a histogram of the sample call-stack depth of 16 applications including benchmarks [92, 10, 99] and in-production programs [81, 79, 136, 98]. The plot shows that some applications (such as 465.tonto, 481.wrf, BigDFT, Nemo and CESM) require unwinding more than ten levels of the call-stack to obtain the full call-path and this effect is likely to grow as applications reuse their code. Note too that the number of stack frames varies in a wide range, thus the number of iterations to completely unwind the call-stack fluctuates during the application execution. The cost of traversing the frames without collecting any data are in the order of microseconds and depends on the number of stack frames unwound, as illustrated in Figure 4.19b. Consequently, the call-stack collecting overhead will vary in different application phases and may lead to inaccurate conclusions.

In order to reduce this overhead and keep it uniform as long as the application executes, the work described here proposes capturing a subset of the call-stack frames up to a fixed limit. This approach involves capturing a contiguous subset of the frames from the top of the call-stack. There are two main reasons for applying such a top-down approach (as depicted in Figure 4.20). First, the collection expense becomes fixed irrespective of the number of existing call-stack frames. Second, the collected frames represent functions that are closer to where the activity occurs, helping the tool to point out what routines were being executed during the monitoring stage. However, the top-down approach presents some difficulties when identifying routines in the call-stack as it evolves in conjunction with the routine entries and exits, so the depth for a particular routine varies along the application activity. Figure 4.21a shows the top of several

[2]Mainly x86 and x86-64 architectures.

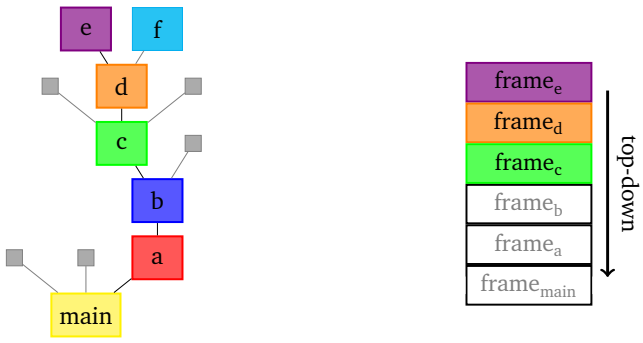


(a) Sampled call-stack depth histogram observed in applications.



(b) Unwind overhead for many call-stack depths in several processors.

Figure 4.19
Costs associated to the call-stack unwinding process.



(a) Call-stack frames activated are represented in color.

(b) Top-down selection of 2 frames (in addition of the top of the call-stack) from the call-graph.

Figure 4.20
Call-stack tree when routine e is active and its selected subset.

captured call-stacks in a time-line of the application with the call-tree shown in Figure 4.20a, exemplifies this problem. In Figure 4.21a, the first sample (s_1) has been interrupted while executing routine f_d which was invoked by f_c and at the same time, f_c was called by f_d. Since the location for a particular routine (e.g. f_b) varies along time, the call-stack analysis must be aware of such an issue.

An algorithm inspired from Multiple Sequence Alignment (MSA) algorithms [160, 123, 52] has been developed in order to reconstruct the approximate complete call-stack from the reduced call-stacks, The MSA algorithms infer biological sequence homology to conduct phylogenetic analyses and study shared lineages from specimens sharing a common ancestor. These algorithms work by arranging sequences of biological molecules (i.e. DNA, RNA or proteins) and identify regions of similarity between the sequences. Their purpose in this research, however, is to provide as inspiration for deducing the necessary line of call-path ancestors.

The simplest alignment involves a pair-wise sequence comparison [199], but this idea is extended to multiple sequences, in which the mechanism searches for the best matching on any number of sequences. The method to align n individual sequences involves constructing an n-dimensional matrix where each column represents a sequence and then applying a pair-wise alignment. The algorithm described here is similar to MSA but substitutes the sequences with the sampled call-stacks. Honoring this metaphor, Figure 4.21a represents the starting point of an alignment of six call-stack samples. In contrast to the biological studies, where the length of the sequences are large and the number of sequences is typically low, the approach described here applies to a limited number of call-stack frames but a large number of call-stacks.

There is a fundamental difference in comparing the two approaches when applying the pair-wise alignment: the presence of mutations. In biology, mutations are either point mutations (i.e. a molecule has been replaced by another) or indels (that is, insertions in or deletions from a sequence). The approach described here disregards both point and deletion-derived mutations but considers insertion mutations to preserve the common call-stack part between two samples. These insertions (henceforth gaps, to keep to the biological nomenclature) are denoted as $\circ(X)$ where X refers to the routine that is being inserted into the deepest part of the target call-stack.

Much like the biological implementation, the call-stack reconstruction algorithm generates an alignment matrix. Algorithm 1 summarizes the algorithmic implementation applied to the call-stacks and Figure 4.21 provides an example on how it works. The algorithm starts looking for the most frequent routine (henceforth referred to here as the *pivot*) in a vector of samples ordered by their time-stamp. This routine serves as a reference for the alignment. Next, the process selects those samples that contain the pivot within their call-stacks and then applies the alignment to this selection. The alignment applies a pair-wise align to consecutive samples by

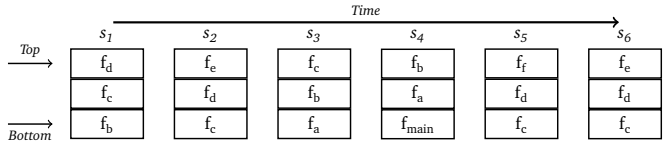
Algorithm 1 Call-stack alignment pseudo-algorithm.

```
1:  procedure CallstackAlignment(vs)
Input: vs: vector(Samples) sorted by their time-stamp
Output: mat: matrix(Samples) aligned
2:      pivot ← vs.searchMostFrequentRoutine()
3:      vtmps ← vs.contain(pivot)
4:      for i ≤ vtmps.size() do
5:          c_c ← vs(i).getCallstack()                    ▷ Current call-stack
6:          if i > 1 then
7:              p_c ← vs(i − 1).getCallstack()            ▷ Previous call-stack
8:              h_cc ← c_c.height(pivot)
9:              h_pc ← p_c.height(pivot)
10:             if h_cc > h_pc then
11:                 for j < i do
12:                     mat(j).InjectGaps(c_c.subset(h_p − h_c))
13:                 end for
14:             else
15:                 c.InjectGaps(c_p.subset(h_c − h_p))
16:             end if
17:         end if
18:         mat(i) ← c
19:     end for
20:     return mat
21: end procedure
```

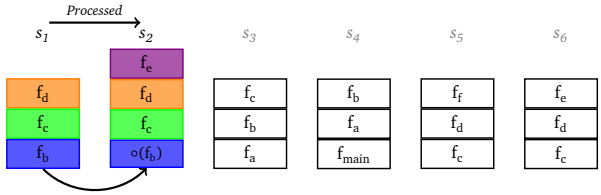
injecting as many gaps as needed to align the pivot of the samples. Consider the processing of samples s_1 and s_2 from Figure 4.21a. In such a case, both samples contain the pivot routine and the pivot routine is at a higher frame in s_1 compared to s_2, so the algorithm injects a gap at the bottom of the call-stack of s_2 (as illustrated in Figure 4.21b) in order to get pivot at the same height. When aligning s_2 and s_3 the algorithm behaves slightly different because the height of the pivot is higher in s_3 and s_3 occurred latter than s_2. In this case, the gaps are back-propagated to those samples that occurred before s_3, as depicted in Figure 4.21c in order to propagate the common call-path. Due to space restrictions, Figure 4.21d depicts in a diagram the alignment between samples s_3 and s_4 and s_4 and s_5, but these are treated one after another. After processing the samples that contain the pivot in their call-stack, an additional step (not shown in Algorithm) aligns the samples that do not contain the pivot but share a portion of the call-stack with the previously treated samples. For instance, sample s_4 in Figure 4.21e has been ignored because it does not contain the pivot, but since its call-stack has routines in common with other processed samples (f_a and f_b) it is aligned in this second step. In this case, the call-stack data from s_4 shows that its bottom frame points to the routine `main`, so the alignment injects a gap into the rest of the samples.

Due to the construction of the alignment algorithm, the selection of the pivot influences the results because it determines which samples are aligned and which are ignored. This fact becomes critical in applications that progress through multiple routines and share a few calling routines in the selected frames of the call-stack. Also, the sampling period affects the results because the larger sampling period, the lesser call-stack activity captured so the call-stacks captured may differ substantially. However, as the folding mechanism gathers scattered samples along an execution then adjacent folded samples refer to adjacent source code references. Consequently, the folded call-stack is likely to exhibit spatial locality and the number of gaps to add should be minimum.

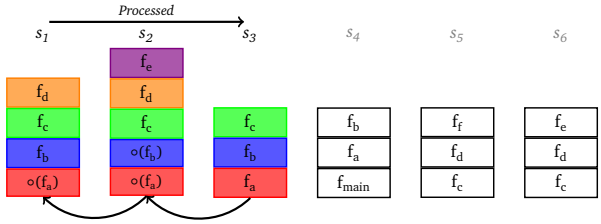
Whenever applying this approach, the folding plot is extended by collocating the call-stack and the performance evolution along the computing region. The analysis of the call-stack discards the small routines that are closer to the application activity (top frame of the call-stack) and focus on those routines that last for a certain period. It is therefore necessary to search within the resulting aligned call-stacks for routines with a certain granularity (determined by a number of samples or



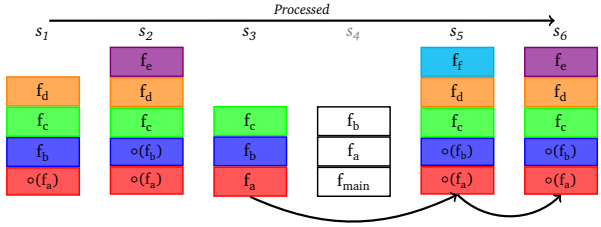
(a) Samples as collected. f_c is the pivot.



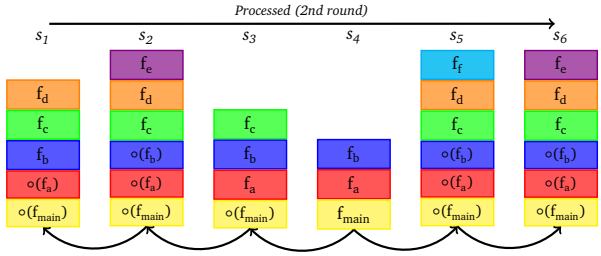
(b) Alignment of s_1 and s_2. Bubble injection into s_2.



(c) Alignment of s_2 and s_3. Bubble injection into previous samples.

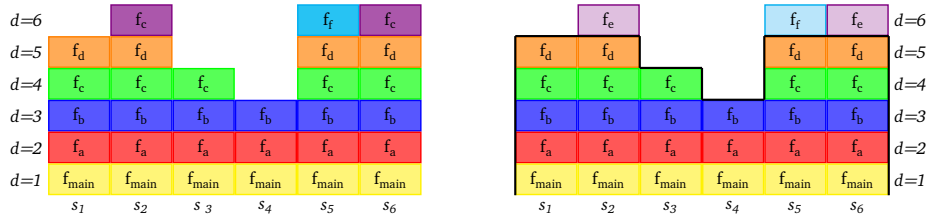


(d) Alignment of s_3 and s_5, s_6. Double gap injection into s_5 and s_6.



(e) Processing the remaining samples s_4 that do not contain f_c. Need to inject gaps into the other samples.

Figure 4.21
Example of alignment of a set of samples.



(a) Matrix representation of the call-stacks depicted in Figure 4.21e.

(b) Selection of routines of interest with a threshold≥2.

Figure 4.22
Post-processing the alignment samples to choose the routines of interest.

a duration). This search processes the aligned call-stacks as if they were stored in a matrix where columns refer to samples ordered by their collection time and rows point to the depth of the call-stack frame. For instance, Figure 4.22a shows the matrix representation of the results shown in Figure 4.21e. The search algorithm looks for the number of consecutive frames at a particular height that point to the same routine. If the number of consecutive samples is higher than the given threshold, then the process is applied recursively to the upper level (d=d+1) within the interval defined by interval of samples that contain the frames pointing to the same routine. For instance, Figure 4.22b shows in solid color and below a black line the routines that are selected for the analysis when the threshold is two samples whereas the transparent are ignored. In this example, all the frames of every sample point to the same routine at levels d=1, d=2 and d=3. When processing the frames at d=4, the mechanism splits the recursive analysis in two parts (one involving samples s_1-s_3 and the other involving s_5-s_6). The analysis finishes at d=6, where routines f_e and f_f are skipped because they only last one sample. In the end, the selected portion of the matrix represents the active routines with a certain granularity and the top of the selection denotes the active routines.

The main loop of the NAS MPI bt.A benchmark (the routine `adi`) was instrumented to delimit the computing region and sampled to exemplify the results of the combined framework. Figure 4.23 shows the combined plot containing the source code and the performance views. The Figure is divided into two plots that represent the progression along the instrumented region for the routines and a profile of source code lines (at the top) and the node-level performance metrics (at the bottom). Both plots share a distinctive background color according to the active routine. Due to plot rendering limitations, the short routines (as in the transition from s_3 to s_4 in Subfigure 4.22b) are depicted with a white background, though they can be analyzed using the trace-file. The plot at the top shows text displaying the name of the active routines and up to two calling ancestors (in the form of $X > Y > Z\ [n]$, where Z is the active routine, X and Y refer to the ancestors and n is the most observed source code line within the active routine). For instance, the green phase represents the routine `x_solve_cell`, which is called by `x_solve` and invests most of its time in line 723. This plot also shows pink points representing a time-based profile of the source code lines within the active routine (where the top refers to the top of the file that contains the routine). The solvers (`*_solve_cell`) show almost a random line progression limited to the half bottom of the plot, which indicates the presence of a loop that covers the bottom half of the file and that spans for the whole execution of the routine. The comparison of the solvers shows that `x_solve_cell` lasts less than the rest and that most of the samples point to one line at the bottom of the source file; while the other solvers have mainly sampled two lines at the bottom of the file. Finally, routines `*_backsubstitute` (depicted in white and manually labeled as A, B and C) invest most of their time on one line, which observing the source code corresponds to a single statement within the five-nested loop that comprises the routines. With respect to the performance plot, the black line represents the MIPS using the right Y-axis, while the remaining lines are plotted on the left Y-axis and show the ratio of cache misses per instruction at different levels of the cache hierarchy. The MIPS rate within the solvers is uniform, being higher on `x_solve` due to less cache miss ratios in L2D and Last-Level Cache (LLC). This

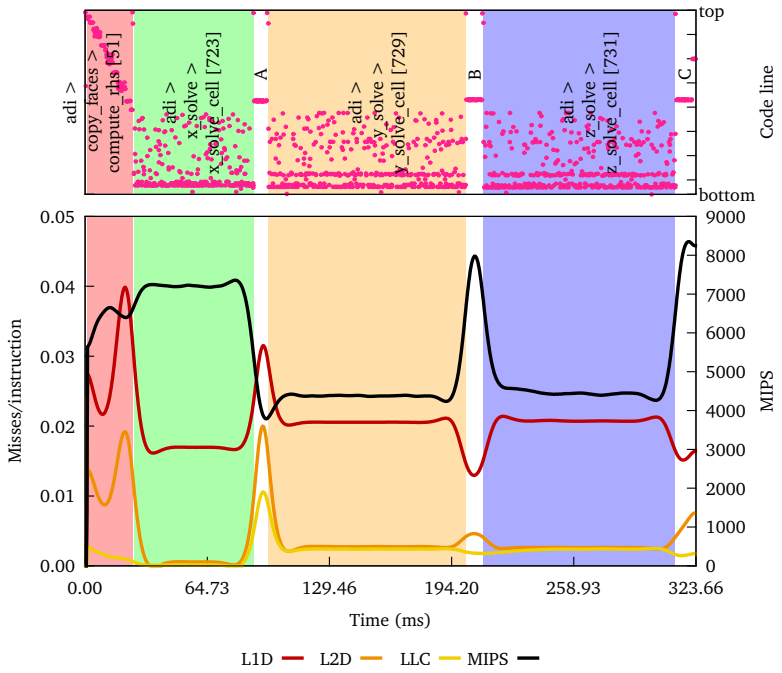


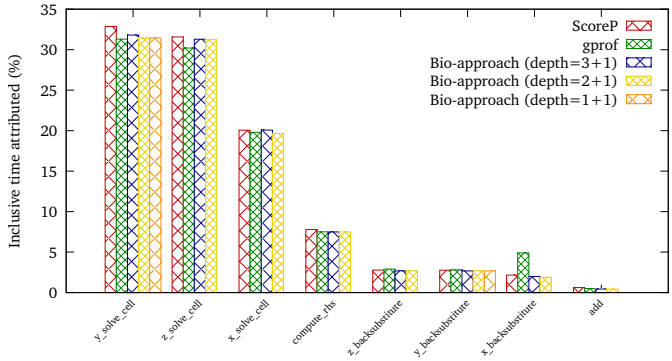
Figure 4.23
Source code-references and performance metrics collocated for the MPI version of the NAS bt.A benchmark.

behavior contrast with that of the `*_backsubstitute` routines, where `x_backsubstitute` (A in the plot) runs at a slower pace due to an increase in the instructions miss ratios in L2D and LLC.

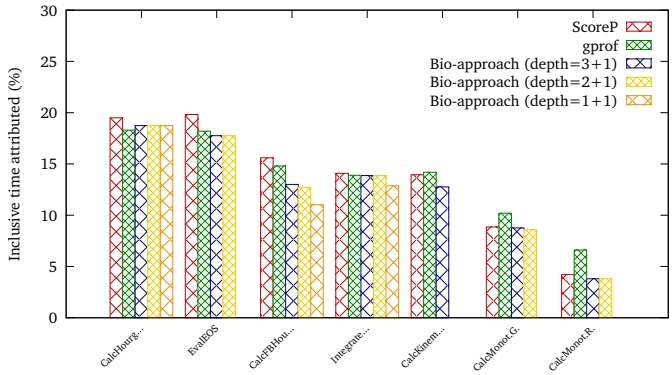
4.5.2.1 Validation

The results provided by this approach are compared to those obtained with other performance tools that rely on different measurement techniques (Score-P [147] using direct instrumentation and gprof using time-based sampling running at 100 Hz). The comparison also helps to detect how the depth of the call-stack unwound affects the results. To conduct this evaluation, the bt.B benchmark from the NAS MPI benchmark suite, Lulesh and the HydroC [175] proxy benchmarks were executed with the three measurement techniques. The binaries were executed with one process to avoid the variability introduced by the network into the different measurement systems. To proceed with the instrumentation-only approach, the gprof results revealed the most representative routines (in terms of percentage of time) later instrumented using ScoreP. To use the bio-inspired approach, the entry and exit points of the application main loop were instrumented and the application then was sampled using a coarse frequency (25 Hz) to generate a trace-file. Such a trace-file was later processed in a way that the instrumented entry and exit points delimit the computing region.

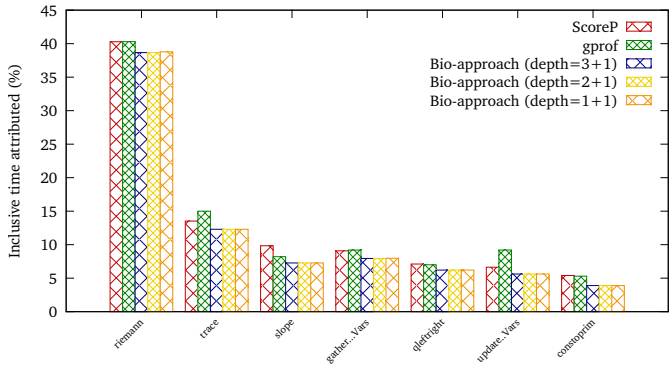
Figure 4.24 shows the results for the applications using gprof, ScoreP and the approach described in this section when collecting different call-stack depths. Note that the call-stack depth is expressed in terms of X+1, meaning that the sampling handler unwound X frames of the call-stack and also emitted the interrupted PC address provided by the sampling handler. In general, it may observed that the results from the proposed approach using $depth$=3+1 are similar to those obtained by ScoreP and gprof. Using a smaller value for $depth$ does not attribute time to some of the routines, though it does provide good approximations for many cases. For instance, it is worth mentioning that in Figure 4.24a the bio-approach only provides measurements for `y_solve_cell` and `y_backsubstitute` when $depth$=1+1. This



(a) NAS bt.B benchmark.



(b) Lulesh mini-app.



(c) HydroC application.

Figure 4.24
Comparison of the code attribution with other performance tools and using different levels of call-stack unwind when applied to different applications.

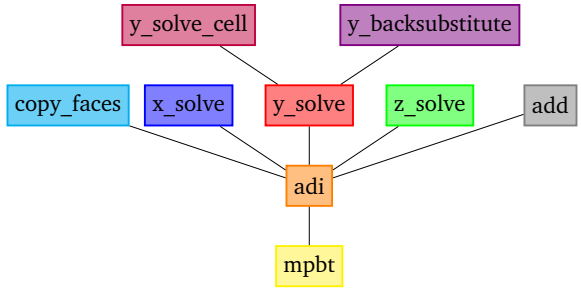


Figure 4.25
Summarized call-graph for the NAS MPI bt benchmark

occurs because the routine `y_solve_cell` becomes the pivot in the algorithm and because of the structure of the application (depicted in Figure 4.25). Since this approach applies to those call-stacks containing the pivot, when $depth$=1+1 the mechanism only finds samples containing invocations from `y_solve`. Then, the second part of the algorithm (which looks for common frames) adds `y_backsubstitute` to the common frames. However, it ignores `adi` because the collecting mechanism did not capture samples at `y_solve` (invoked from `adi`), so the mechanism does not explore the call-stacks that contain this routine. A similar issue occurs in Lulesh (Figure 4.24b) when $depth$=1+1 where only three of the seven most representative routines are disclosed.

4.6 Detecting time evolution of memory access patterns

When it comes to performance analysis, traditional performance analysis tools have naturally associated performance metrics with syntactical application components such as routines, loops and even statements. Though this association has proven valuable and also helpful for understanding and improving applications, the impact of the memory hierarchy makes it necessary to explore the performance from the data perspective, as well. A study from this point of view includes, though it is not limited to, the unveiling of the application variables that are referenced the most and their access costs, the detection of memory streams to assist prefetch mechanisms, the calculation of reuse distances and even identification of the cache organization that may improve the execution behavior. Two mechanisms have accordingly emerged for addressing such studies. On the one hand, there is an instruction-based instrumentation that monitors load/store instructions and decodes them to capture the referenced addresses. While this approach accurately correlates code statements with data references, it comes with a severe cost, overwhelming the analysis with large data collections and/or time-consuming analysis and is not practical for long in-production executions. On the other hand, several processors have enhanced their Performance Monitoring Unit (PMU) to sample instructions based on a user specified period and associate them with data such as the referenced address. For instance, Intel and AMD chips have named their respective extensions Precise Event Based Sampling (PEBS) and Instruction Based Sampling (IBS).

PEBS and IBS work in a similar fashion. These mechanisms periodically choose an instruction from those that enter the processor pipeline. The selected instruction is then tagged and is monitored as it progresses through the pipeline while annotating any event caused by the instruction. When the instruction completes, the processor generates an record containing the instruction address, its associated events and the machine state (without time-stamp), and the record is then written into a previously allocated buffer. Every time the buffer becomes full, the processor invokes an interrupt service routine provided by a profiler responsible for collecting the generated records. Since instructions are reported at the retirement stage, these mechanisms exclude contributions from speculative execution. For the particular case of load instructions,

PEBS collects data such as, but are not limited to: the linear address[3] referenced, the layer of the memory hierarchy that served the reference or how many cycles did it take to reach the processor. Seeing as these monitoring mechanisms report linear addresses from the process address space without providing information on the physical addresses, they are consequently not helpful for understanding issues such as memory migrations.

When exploring the address space, it is convenient to map the address space to the application data structures in order to let the analyst match the generated results with the application code and also explore their pattern access type. For that reason, it is not enough to capture the referenced addresses, but it is also necessary to extend the instrumentation mechanism in order to capture the base address and the respective sizes of the static variables and the dynamically allocated variables. As for the static variables, the instrumentation package needs to explore the data symbols within application binary image using the binutils library [71] in order to acquire their name, starting address and size. Regarding the dynamic variables, the instrumentation has to capture whenever the `malloc` related routines occur and then capture their input parameters and output results to determine the starting address and size. As dynamically allocated variables do not have a name, the tool needs to identify their allocation, which is why the tool need to capture the call-stack reference. Finally, it is worth mentioning that some languages (such as C and C++) allow local (stack) variables to be declared within code blocks that can only be referenced by the inner block statements. While these references are captured by the perf tool, the instrumentation tool does not track their creation; so, their references appear on the resulting plot but do not have an associated variable name.

In order to demonstrate the value of this extension, the folding tool has been applied to a modified version of the Stream benchmark. Since Stream accesses to statically allocated variables through ordered linear accesses, the application code has been modified so that: 1) the `c` array is no longer a static variable but allocated by `malloc` and 2) the `scale` kernel loads data from pseudo-random indices from the `c` array. Due to modification 2), the `scale` executes additional instructions and expose lesser locality of reference, so the loop trip count in this kernel has been reduced to $N/8$ to compensate its duration. The resulting code looks as the one depicted in Listing 4.2. With respect to the measurement, the main loop body was instrumented and the application sampled at 20 Hz and captured memory references every 250k load instructions. Regarding the execution environment, the application was executed on an Intel®Core™i7 2760QM running at 2.40 GHz on a Linux 3.11 operating system. The Core™i7 processor has three levels of cache with a line size of 64 bytes: level 1 are two 8-way 32 Kbyte caches for instructions and data, level 2 consists of a 8-way unified 256 Kbyte cache and level 3 is a 12-way unified 6,144 Kbyte cache.

Listing 4.2
Modified version of the Stream benchmark.

```
1  for (i = 0; i < NTIMES; i++)
2  {
3    for (j = 0; j < N; j++)
4      c[j] = a[j];           /* Copy */
5    for (j = 0; j < N/8; j++)
6      b[j] = s*c[random(j)]; /* Scale */
7    for (j = 0; j < N; j++)
8      c[j] = a[j] + b[j];    /* Add */
9    for (j = 0; j < N; j++)
10     a[j] = b[j] + s*c[j];  /* Triad */
11 }
```

[3]Linear addresses also refer to logical addresses in x86-64 architectures since segmentation is generally disabled thus creating a flat 64-bit space, according to sections 3.3.4 and 3.4.2.1 from *Intel® 64 and IA-32 Architectures Software Developer's Manual* as of April, 2,015.

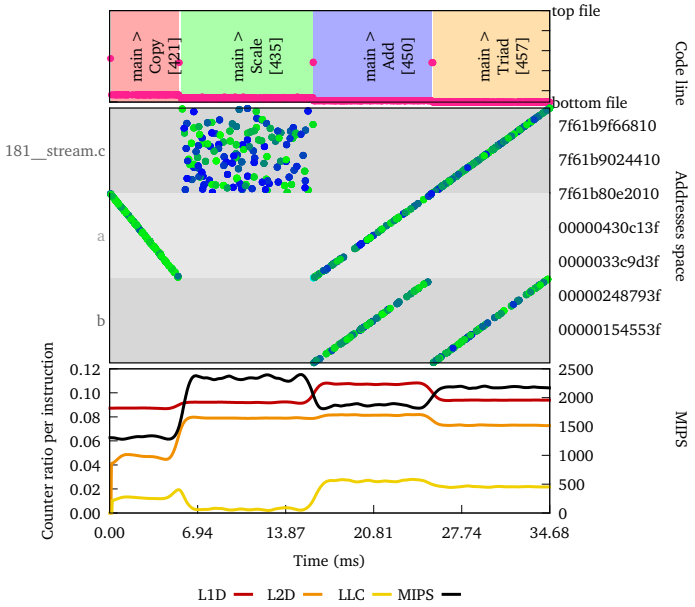


Figure 4.26
Analysis of the modified Stream benchmark. Triple correlation time-lines for the main iteration: source code, addresses referenced and performance.

Figure 4.26 shows the result of the extended framework. The Figure consists of three plots: 1) source code references (top), 2) address space load references (middle) and 3) performance metrics (bottom). In the source code profile each color indicates the active routine (identified by a label of the form X >Y [n], where Y and X refer to the active routine and its ancestor and n indicates the most observed line). In addition, the purple dots represent a time-based profile of the sampled code lines where the top (bottom) of the plot represents the beginning (end) of the container source file. This plot indicates that the application progresses through four routines and that most of the activity observed of each of these routines occurs in a tiny amount of lines. The second plot shows the address space and their references. On this plot, the background color alternates showing the space used by the variables (either static or dynamically allocated) and the left and right Y-axes show the name of the variables referenced and the address space, respectively. The dots show a time-based profile of the addresses referenced through load instructions and their color indicate the time to solve the reference based on a gradient that ranges from green to blue referring to low and high values, respectively. This plot outlines several phenomena. First, as expected, the access pattern in the `Scale` routine to the variable allocated in line 181 of the file `stream.c` (formerly `c`) shows a randomized access pattern with most of the references in blue (meaning high latency). The straight lines formed by the references in the rest of the routines denote that they progressively advance and so expose spatial locality while the greenish color indicates that these references take less time to be served. Second, the `Copy` routine accesses to the array `a` downwards even though the loop is written so that the loop index goes upwards. This effect occurs because the compiler replaced the loop with a call to `memcpy` (from glibc 2.14) that reverses the loop traversal, unrolls the loop body and uses SSSE3 vector instructions. A linear regression analysis indicates that approximately each instruction references five addresses in `Copy` and since SSSE3 vector instructions may load up to 16 bytes, this translates into a 31.25% vector efficiency. Finally, the instructions within routines `Add` and `Triad` reference two addresses per variable in average, the loaded data comes from two independent variables (or streams) simultaneously, and their accesses go from low to high addresses honoring the code. The third plot shows the achieved instruction rate (referenced on the right Y-axis) within the instrumented region as well as the L1D, L2D and LLC cache misses

per instruction (on the left Y-axis). While the reader would expect a large cache miss ratio per instruction in `Scale`, they observe that the kernels behave similarly to the rest of the kernel routines. This occurs because `random()` executes instructions to compute its results without accessing the memory, so reducing the cache miss ratio per instruction.

4.7 Precise evolution of power consumption

Unlike the performance monitoring, power monitoring has proven complicated and intrusive until the inclusion of self-power consumption monitoring capabilities into the processors such as the RAPL infrastructure available on the Intel®processors [43, 182]. RAPL offers a mechanism to limit, control and monitor the power and energy use of a single processor socket. The monitoring capabilities are accessible through the PAPI performance measurement infrastructure[4] [20] allowing performance tools to integrate the power and energy metrics seamlessly. As a result of this integration, performance tools not only simultaneously report performance and power measurements for their correlation, but also allow the folding mechanism to depict the fine evolution of the power metrics.

RAPL is accessible through the processor model specific registers[5,6]. The RAPL interface exposes several domains of power distributed for every processor socket and each can be monitored and limited independently. The interface domains consist of a package domain (i.e. the whole socket), a basic power plane (*i.e.* the cores of a single processor socket), a memory domain (*i.e.* the directly-attached DRAM) and, optionally, additional power planes that are commonly assigned to the integrated GPU. The difference with respect to performance tools and according to the processor manual, the energy consumption information is updated at 1 kHz and by default, the processor socket reports the energy measurements in multiples of 15.2 μJoules. Another difference between performance and energy counters is that performance counters are associated with the thread-level while the energy counters sum up all the energy used by the entire socket.

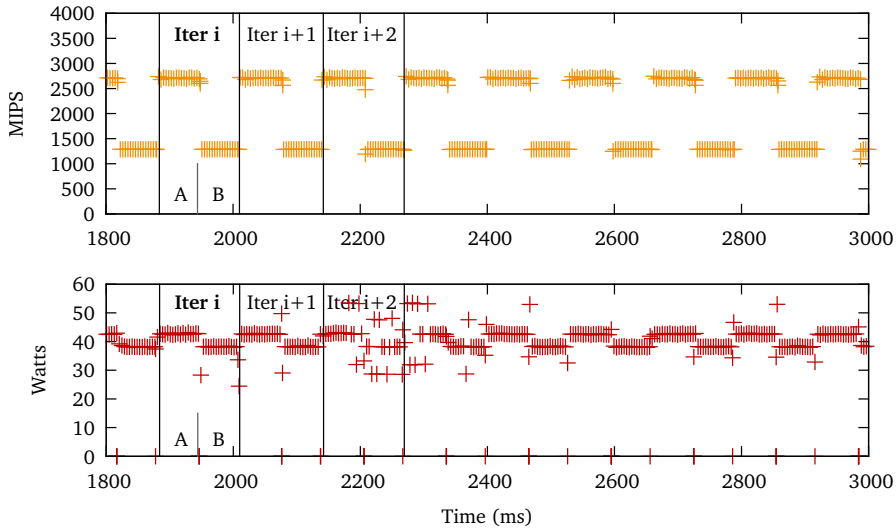
In this thesis a serial benchmark has been designed to execute a loop that invokes two kernels (named A and B) on a RAPL-ready processor to illustrate the differences between the accuracy of the performance and power counters. Each kernel is designed to consume a different amount of energy constantly and to execute at different MIPS rate, though they are designed to last approximately the same time. The benchmark executes in conjunction with an instrumentation package that collects performance and power metrics periodically and also at the start and end points of each kernel. Figure 4.27a shows the achieved MIPS rate and the drained power of the package by this benchmark on a small period of the whole benchmark execution. The results show that the instructions counter is less sensible to noise than the power counter. More precisely the reader may observe that iteration labeled as *Iter i+2* is particularly perturbed. With respect to the noise, the Figure shows that at some kernel changes the readings gather a 0-value, which means that the PCU has not updated the energy measurement from the previous reading. Superimposing the results of each iteration (what the folding process does conceptually), results in the plot depicted in Figure 4.27b. This Figure shows that at start and end points of each kernel (i.e. kernel A starts at 0 and ends at 60 ms and kernel B starts at 60 ms and ends at 130 ms), the power counter presents perturbation while the instruction counter does not. The perturbation correlates with the frequent reads done by the instrumentation package at the enter and exit points of the kernel and the fact that the power measurements are quantified and discrete. This conclusion derives from the fact that it is very unlikely that two consecutive instrumentation points allow the PCU to update the power metering nor consume 15.2 μJoules.

To avoid the issues related with the quantification and discretization, the data gathering and the folding algorithm require modifications in three different directions: reducing the number of counter measurements, improving the outlier removal and reducing the noise.

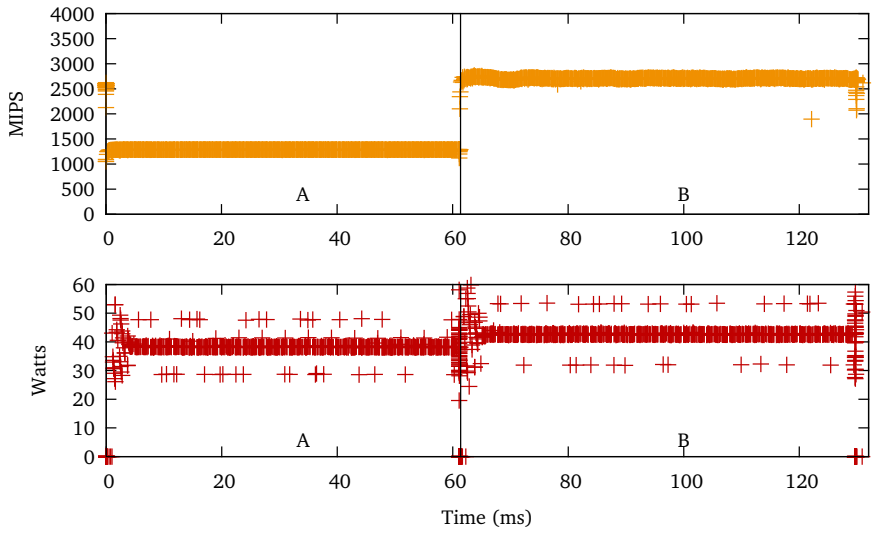
[4] Available since PAPI version 5.0.

[5] For further reference, see section 14.9 from *Intel® 64 and IA-32 Architectures Software Developers' Manual* as of April 2,015.

[6] Although the use of MSR registers is restricted by the kernel, a regular user can read their contents in Linux operating system by setting the appropriate read permissions to the `/dev/cpu/*/msr` files.



(a) Raw evolution of the instruction rate executed (top) and the package power consumption (bottom).



(b) Superimposed MIPS (top) and power metrics (bottom) for the two different kernels.

Figure 4.27
Evolution of the MIPS counter (top) and the package power consumption (bottom) for a benchmark that executes two different kernels.

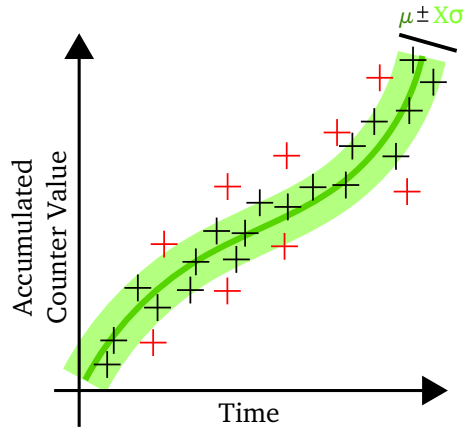


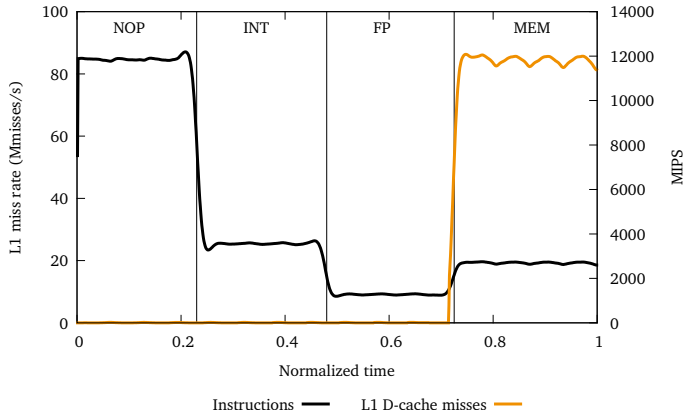
Figure 4.28
Distance-based instance filtering used in the folding mechanism. The green line refers to the curve fitting result when using with the samples (μ) and the gray color shows the area (within $\mu \pm X \times \sigma$). Those samples that have to be taken into account are shown in red whereas the samples that can be ignored are colored in black.

- The data generation process emits the performance counter values at the beginning and end of the instrumented region and the folding mechanism normalizes the performance counters to minimize the perturbation suffered on each instance. The reduction of the measurements is achieved by not capturing the power metrics either at the start or end points of the instrumented region. It is therefore impossible to know exactly the total energy consumed by the region. Suppressing the monitoring at the beginning of the instrumented region requires interpolating the first folded sample in order to approximate the actual consumed energy since start of the region. On the other hand, removing the monitoring at the end of the instrumented region leaves two opportunities: estimating the consumed energy from the last folded sample to the end of the region, or apply the folding to absolute values (not normalized). The work in this thesis has focused on the last alternative, i.e. not capturing the power metrics at the end of the region and apply the folding to absolute values.

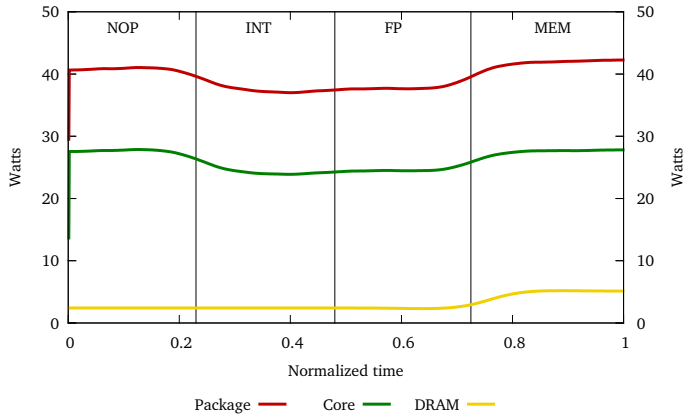
- The second direction moves toward using a stricter filtering step. A pre-filtering step has been added to the folding mechanism before applying the curve fitting of the samples to improve the outlier removal. This step interpolates a subset of the samples and then uses the result as a reference so that those samples that lie within $\mu \pm X \times \sigma$ are finally used in the interpolation, as shown in Figure 4.28. The dark green line depicts the preliminary interpolation (μ) and the light green region shows the area (the $\pm X \times \sigma$ part of the equation) that surrounds the interpolation. The filtering keeps the samples that lie within the light green area (colored in black) and discards the samples that are beyond the designated area (colored in red).

- Finally, it is worth adapting the curve fitting parameter to the counter used. When using the Kriging interpolation, for instance, the nugget parameter determines how strict the interpolation follows the points, therefore it acts as a low-pass filter. So as to mitigate the noise observed in the energy counters, the folding uses a more relaxed value for the nugget variable.

It is possible to test the modifications on the folding process as well as showing the potential of plotting together performance and energy counters by using a hand-crafted benchmark. The benchmark consists of four kernels that take approximately 500 million cycles each, for a total of 76 ms in an Intel®SandyBridge processor running at 2.6 GHz. Each kernel executes at a different instruction rate by executing different types of instructions, including no-operations



(a) Performance progression.



(b) Power consumption progression.

Figure 4.29
Study of the folded results for performance and energy measurements by using a serial benchmark that executes four different types of kernels.

(NOP), integer operations (INT), floating-point operations (FP) and memory operations (MEM). The output of applying the folding mechanism to both performance and energy counters into the benchmark is shown in Figure 4.29. The Figure shows that the high instruction rate of the no-operation kernel translates into high energy consumption at the package- and core-levels. This relation, however, only applies to high instructions rates and not to lower instruction rates as observed in the kernels that execute integer and floating-point instructions. Although the integer operations execute twice as fast as the floating-point kernel; the energy consumption of these two kernels is approximately the same. With respect to the memory bounded kernel, note the expected increase on the L1 data cache misses and the increase in the DRAM energy consumption and also at the core-level.

4.8 Analysis methodology for parallel applications

The folding results are very useful for detailing the performance evolution within delimited code regions. Application developers may find these results valuable because they correlate

performance and source code locations and help to confirm whether the performance expectations are met. These developers easily identify application spots to study because they are familiarized with the application code structure. However, there are two scenarios worth discussing. First, it is becoming frequent that analysts have to report on the application performance without having any preliminary knowledge of the application. Second, applications are becoming more and more complex on every new version, so it is convenient to search for alternatives to analyze applications from scratch, even for developers or external analysts. This section therefore proposes a methodology for finely analyzing a parallel application without the need for analysts to have prior knowledge of the application studied.

The methodology presented is based on Amdahl's law: it is worth studying those code regions that represent most of the execution time because the returns on improving them will be higher where there are node-level optimization opportunities. This methodology takes advantage of a clustering tool for trace-files to categorize the application *computation regions* (i.e. the code comprised between a parallel run-time exit and its subsequent entry) using a Density-Based clustering algorithm [63] (DBSCAN) and enriches the trace-file with the clusters found [85]. The clustering tool assigns to each individual computation region its performance metrics derived from the trace-file. The metrics typically used refer to the execution rate (cycles-per-instruction [CPI], or inversely, instructions-per-cycles [IPC]) and the computational complexity (the number of instructions executed and the ability of the algorithm to distribute the work among processes), but these metrics can be changed at user request. In this plot, points (i.e. computing regions) that appear on the right execute at slower pace than those located on the left and those computing regions that are on the top execute more instructions than those located at the bottom. Then, the DBSCAN algorithm groups the regions according to the selected metrics and according two additional parameters that control the minimum points to form a cluster ($MinPoints$) and the search radius neighborhood (Eps). The resulting clusters obtained are those subsets C_i of the data that fulfill the following conditions:

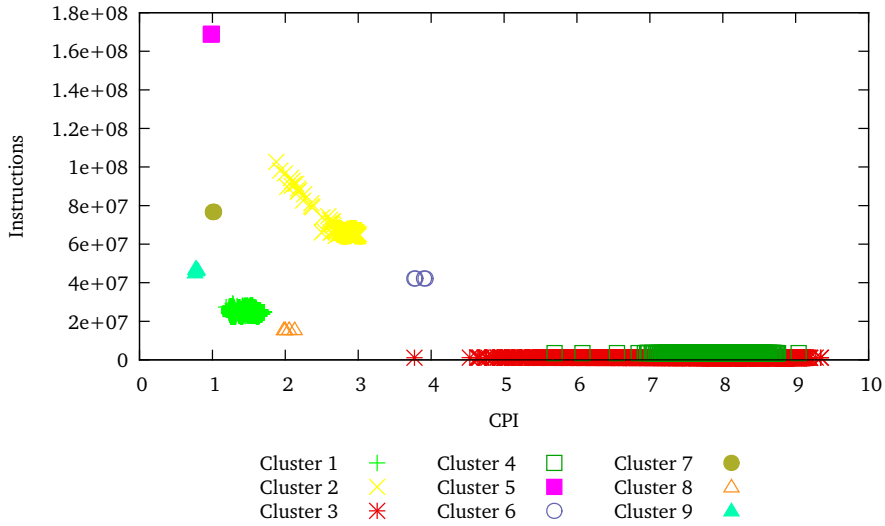
1. For any given pair of points $p \in C_i$ and $q \in C_i$ it is possible to find a set of points $a_1, a_2, ..., a_{n-1}, a_n \in C_i$, being $p = a_1$ and $q = a_n$, where the Euclidean distance for each pair a_i, a_{i+1} is less or equal to Eps. This property is called *density reachability*.

2. $|C_i| \geq MinPoints$. This is the minimum density condition to consider C_i as a cluster.

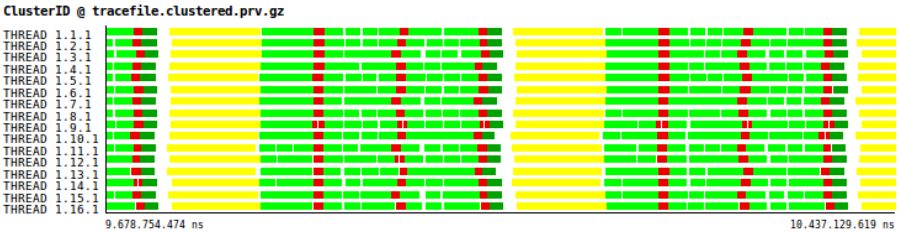
Then, the grouping information is written into the trace-file for an ulterior analysis.

The main objective of the analysis and optimization process would include moving the computing groups to the leftmost part of the plot and possibly, to the bottom part. Since it is unfeasible to optimize the whole application, the methodology proposed focuses on analyzing those regions that take most of the time. To this end, the clustering tool not only groups the computing regions, but also sorts the identified groups according to the fraction of the total time they represent with respect to the total computation. The analyst should therefore focus on the groups with a lower identifier and which are located in the right (or top-right) part of the plot.

Figure 4.30 shows the results for the clustering tool when applied to a parallel application. Each point in the scatter-plot represents the execution of a computation region that is classified by two axes: instructions (on the Y-axis) and CPI (on the X-axis). The DBSCAN algorithm analyzes the whole cloud of points and identifies 16 groups and each group is represented using a unique combination of color and shape in the plot. For instance, Cluster 1 (+ shape in light green) refers to computation regions that execute approximately 2×10^7 instructions with an unbalanced CPI that ranges from 1.1 to 1.7. On the other side, compute regions belonging to Cluster 2 (depicted in yellow) presents variability on both axis (instructions and CPI). Clusters 3 and 4 (red and dark green, respectively) experience a differences in terms of CPI but each instance represents an extremely short computation region in terms of instructions. Other groups, such as Clusters 5, 7 and 9 present a steady behavior across invocations. Variability in instructions means that the computation regions do not always execute the same code and that is most likely to happen due to work unbalance. Variability in CPI indicates that the region does not behave at the same performance; and that effect is not only likely to occur if the work differs from instance to instance, but also because the region faces different bottlenecks at each execution. In this case, it is interesting to note that the more instructions executed, the lower CPI; so at some level, the



(a) Scatter-plot results.



(b) Time-line results focusing on a region that exhibits two repetitive patterns.

Figure 4.30
Clustering results for a MPI application.

faster execution compensates for the extra work to execute. The variability, whatever the type, influences the folding mechanism when applied to clusters because it translates into computing regions that behave differently and typically produces curly performance counter rate results though this effect can be reduced by increasing the outlier removal factor.

With respect to the time-line, it provides further insight of the clusters and their progression along the execution. For instance, the Figure exhibits a region in time with two repetitive regions. Each region consists of a sequence of the execution of a Cluster 2 (yellow) followed by several Cluster 1 (green) compute regions interleaved with Cluster 3 (red) compute regions and ending with Cluster 4 (dark green). The time-line also shows that the application presents a SPMD behavior as the sequence of compute regions does not depend on the process examined.

4.9 Framework for a productive node-level performance analysis

A framework that combines the clustering tool, the folding mechanism and a set of performance counters was created in order to implement the aforementioned methodology. Figure 4.31 shows the data-flow of this framework. The framework detects and categorizes similar computation regions within parallel applications and then depicts the progression of multiple performance counters within the associated source code and optionally, the memory references. Also, and if requested by the user, the framework combines multiple performance counters using analytical

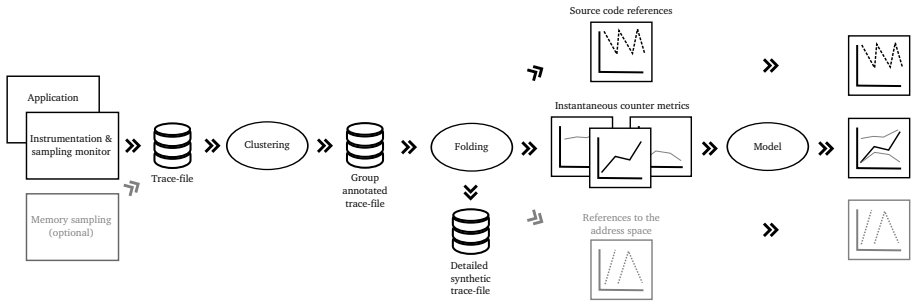


Figure 4.31
Data-flow of the resulting framework.

models based on performance counters to simplify the analysis of the performance metrics. This section describes how the trace-files are generated and how these trace-files are processed by the framework to provide its results.

4.9.1 Trace-generation

The work described in this thesis aims to support any measurement system that supports instrumentation and sampling at the same time using a public API (see the Appendices at the end of this thesis). As this thesis has been framed in the context of the BSC performance tools, the current implementation of the framework receives a Paraver trace-file generated with the Extrae instrumentation package [66]. Extrae automatically instruments the application using the `LD_PRELOAD` environment variable and gathers information using PMPI [70] instrumentation layer or other instrumentation techniques and sampling mechanisms without any further user intervention. Once the application finishes, the instrumentation package generates a trace-file containing performance measurements attached to parallel programming model calls and also to sample points. Note the information collected at instrumentation and sampling points:

Instrumentation points collect time-stamped performance counter metrics that help to determine the cumulative number of events that occurred within the delimited region,

Sampling points capture time-stamped information including performance counter values, a segment of the call-stack references and memory references.

As noted earlier, the framework reports information regarding memory references. While Extrae captures information regarding the performance counters through PAPI, this library does not capture the PEBS generated information[7] so it is necessary to capture this information from another tool. perf [144] is a tool that uses the performance counters subsystem in Linux and since Linux kernel version 3.11 it benefits from PEBS or IBS to collect memory references from either load or store instructions, but not both at the same time. This tool allocates a 1-entry buffer to store the memory references and then samples the application at a user defined period, so each time the processor reaches the period, it generates a memory reference record and then perf captures this record and associates a time-stamp to it. This way, perf is capable of generating time-stamped trace-files containing sampled memory references even though neither PEBS nor IBS capture a time-stamp. Both (Extrae and perf) tools must use the same timing source in order to correlate the data captured, but the perf tool uses low-level kernel timing routines and Extrae uses the Posix compliant high precision clock routines by default. In this thesis, Extrae was modified for use in the same low-level routines from perf by adopting a kernel module that exposes these timing routines[8] to the user-space applications, though there are other possibilities for achieving this goal.

[7] As of PAPI 5.4.0.
[8] https://lkml.org/lkml/2013/3/14/523

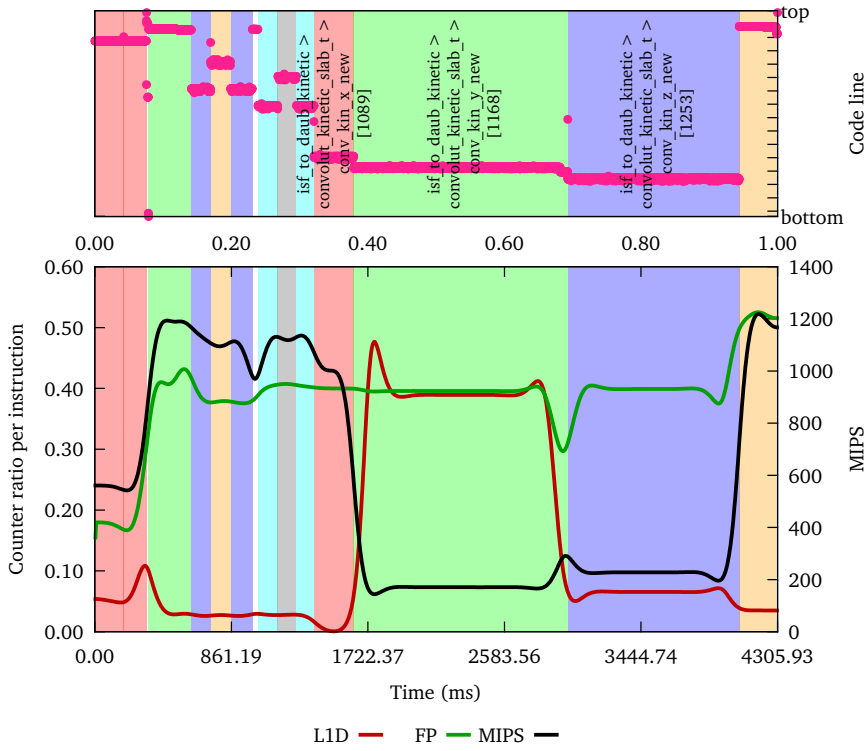


Figure 4.32
Sample of the framework results.

4.9.2 Framework data-flow

Once the user provides a trace-file containing MPI and/or OpenMP delimited regions and sampled performance metrics, the first stage of the framework detects similar computation regions within the application through the collected performance data using the clustering tool. The clustering tool processes the application trace-file to detect similar computation regions and then adds events to each computation region according to its categorization into the trace-file. Once the regions have been delimited within the trace-file, the folding mechanism uses these new events to obtain instantaneous metrics of the hardware counters within each computation region. This step is particularly useful when the analyst does not know the application and chooses the sampling frequency blindly, because it may be the case that the sampling frequency is too coarse to provide any details within the computation region. At the end of this step, the framework produces as many detailed reports as the combination of identified clusters and performance counters. If the processor type on which the application ran has an associated performance model, the framework combines the detailed metrics of a cluster into a single report in order to summarize all the results in the lesser possible reports.

4.9.3 Example of the result

The framework has the ability to report instantaneous performance metrics or apply performance models based on performance counters, in addition to pin-pointing to the responsible source code and referenced addresses. The results of the framework are expressive because they expose the evolution of the performance metrics, associate them with the source code and also express their temporal behavior. Compared to regular profilers, or even trace-based tools, the

framework is able to provide metrics in between instrumented points but does not summarize the performance under a single value, which may mislead the analyst into thinking the region behaves uniformly across the execution when in fact it does not. Figure 4.32 exemplifies the results of the framework when applied to the application BigDFT [79]. The instruction rate is depicted in black and referenced on the right Y-axis and several ratios with respect to performance counters (L1D cache misses and floating-point instructions) are referenced on the left Y-axis. The instruction rate observed is below 200 MIPS for two thirds of the computing region and spans for two source code identified regions (green and blue). The reason for such a low MIPS rate seems to be related to the high number of L1D cache misses, which is approximately 40% and 7% in the green and blue regions, respectively. The source code correlation points to two regions into the code of the routines `conv_kin_y_new` and `conv_kin_z_new`. This type of outcome is very useful for the analyst when it comes to understanding whether there are bottlenecks in the application, and if there are any, what their nature is and where they are located.

In summary, the framework produces rich reports that no other performance tools described in the related work provides. While the clustering algorithm automatically identifies the potential regions to be optimized, the folding finely depicts the performance evolution of these regions and correlates the performance with the source code. Although the folding results provide considerable insight, the use of performance models or ratios between performance counters allow the analyst to avoid the need for digging into the hard semantics of the performance counters and to keep away from them by using high-level categories that are easier to understand.

5 Practical uses of the framework

```
The consciousness of AC encompassed all of what had once been a
Universe and brooded over what was now Chaos.  Step by step, it must
be done.
And AC said, "Let there be light!"
And there was light.
```

— Isaac Asimov, THE LAST QUESTION

Is seeing really believing? This chapter evaluates the performance (and sometimes power) of a first-time seen broad spectrum set of applications executed on a variety of systems to demonstrate the value of the framework and specially the folding mechanism, presented in this thesis. While some of the applications are compiled using aggressive compiler optimization flags, the framework is capable of pointing out the nature of the performance bottlenecks and their location within the source code. Such a correlation allows the analyst to understand the characteristics of the application and the reason why the node-level performance is behind the processor's peak performance. Moreover, this information guides the analyst to apply small modifications to correct the inefficiencies and improve the overall performance, so, demonstrating the productivity of the framework.

5.1 Application analyses

This chapter is a *compendium* of several application analyses that were carried out during the research framed in this thesis and which demonstrate the usefulness of the folding mechanism as well as the methodology and the framework presented. Before moving on into the discussions, note that some parts of the framework evolved during this thesis and that some facts are worth mentioning regarding this evolution. First of all, the clustering tool groups computation regions according to their performance parameters (typically instructions and CPI [or IPC]). However, it may be noted that there are some cases in which the metrics include instructions and duration (or DPI[1]). There are two reasons for this happening. First, at the early stages of the development it was not possible to capture the value of the cycle counter when the sampling signal derives from this counter because the PAPI library forbade it. Consequently, in these first analyses the clustering tool relied on instructions and duration (instead of instructions and CPI [or IPC])

[1] Duration per instruction.

when grouping the computing regions. Second, there are systems, such as the Intel Xeon Phi processor, that only allow capturing two performance counters at a time. So, one counter gathers instruction events always, while the other performance counter swaps periodically among other performance counters for multiplexing purposes. With respect to the interpolation mechanism, it also occurs that the Kriging contouring algorithm was the first available fitting implementation during this research. Consequently, most of the analyses use this fitting algorithm; however, there are analyses in which the fitting rely on piece-wise linear regressions.

Tables 5.1 and 5.2 show, respectively, details regarding the applications and the systems used thorough the analyses described. All the selected applications use the MPI parallel programming model and most of them are compiled using the native compiler (when available). The applications are compiled using flags that enable aggressive optimizations but the exceptions to this rule are BigDFT and CESM because the problem they implement does not converge using these type of optimizations. Moreover, the applications are compiled requesting optimization reports and also, adding debugging information into the application binary in order to translate the call-stack information into code references. It is worth noting the size of the applications which range from a few thousand lines of code (CGPOP) to millions lines of code (CESM). Also, the Arts_CF application was provided by the user in the binary form without providing access to the source code. In addition, serial benchmarks from the NAS and SPEC benchmark suites were used to study the applicability of the combined energy and performance analysis through the folding mechanism. With respect to the systems, the Intel-based clusters are the most dominant (as it occurs in the Top500 list as of writing this thesis, representing more than 80% of the listed systems). *Altamira* is the only system in which RAPL is enabled, so experiments focusing on combining performance/power analysis were executed in this system. The system labeled as *Experimental* served to explore the temporal analysis of memory access patterns. Owing to the requirements of this type of analysis, it was not possible to use in-production systems and the experiments used two computers connected over Ethernet. The first computer (Core i7) has loaded the kernel module to provide the same clock source to perf and Extrae, but as the computer only has four cores the remaining processes are executed on the support computer (Xeon) to prevent the former computer from overloading.

Table 5.3 provides information regarding the execution characteristics of each application (not including the execution of serial Benchmarks). The Table indicates on which system an application was executed, how many processes were used for the analysis and also the sampling frequency. These applications were executed to reproduce an in-production environment using a representative input data-set and number of processes. The processes of the applications have been pinned to a particular core during the experimentation, disallowing the processes from migrating between cores. The process pinning ensures that the performance or energy consumption does not degrade due to the movement among cores because process have always access to the same private cache memory. With respect to the performance collection mechanism, the applications were instrumented at MPI calls and sampled using frequencies that range from 10 to 50 Hz, ensuring negligible overhead and below `gprof`'s sampling frequency (100 Hz).

Regarding the metrics depicted by the framework's results, they depend on the available performance/energy counters to the system used. If the system used has a performance model associated with its processor (such as *MareNostrum2*, which uses the IBM®PowerPC®970MP) then the results honor the model. Otherwise, the results depict several meaningful performance counters along the region to correlate performance counters. When the folding results include the MIPS rate, the instruction execution rate is represented in black and referenced on the right Y-axis; whereas the remaining counters use the left Y-axis. The correlation between performance and source code is achieved using the different approaches stated in this thesis. In order to

[2] https://grid.ifca.es/wiki/Supercomputing/Userguide

[3] http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/Configuration/Configuration_node.html

[4] http://www.bsc.es/marenostrum-support-services/marenostrum-system-architecture

[5] http://www.bsc.es/marenostrum-support-services/mn3

[6] http://www.bsc.es/marenostrum-support-services/other-hpc-facilities/nvidia-gpu-cluster

[7] https://www2.cisl.ucar.edu/resources/yellowstone

[8] Memory sampling references are captured every 10^6 load (or store) instructions.

Table 5.1
Applications' characteristics.

	Arts_CF 17Oct2014	**BigDFT** v1.7r28	**CESM** v1.1.1	**CGPOP**
Number of Klines	338	332	1,390	6
Number of files	564	713	1,940	20
Compiler suite	Intel Compiler v14.0.2	IBM XL v14.1	Intel Compiler v13.0.1	IBM XL v12.1
Compiler flags	-O3 -vec-report2 -opt-report -g	-O2 -qtune=qp -qarch=qp -qreport	-O2 -fp-model precise -g	-O3 -qstrict -qreport -g

	GTC v2	**HydroC** 22Apr2012	**Mr. Genesis**	**Nemo** v3.4
Number of Klines	76	3	14	221
Number of files	24	40	51	484
Compiler suite	Intel Compiler v13.0.1	GNU v4.4.6	IBM XL v12.1	Intel Compiler v13.0.1
Compiler flags	-O3 -xAVX -vec-report2 -g	-O3 -g	-O3 -qstrict -qreport -g	-O3 -vec-report2 -opt-report -g

	Nest v2.2.2	**PEPC** v1.0	**PMEMD** 16Jun2011	**Siesta** v3.1
Number of Klines	100	12	95	124
Number of files	392	48	109	207
Compiler suite	Intel Compiler v13.1.3	GNU v4.3.4	IBM XL v12.1	GNU v4.4.6
Compiler flags	-O3 -vec-report2 -mmic -g	-O3 -g	-O3 -qstrict -qreport -g	-O3 -g

Table 5.2
Systems' characteristics.

System name	**Altamira**[2]	**Juqueen**[3]	**Knights**	**MareNostrum2**[4]
Processor type	Intel®Xeon®E5-2670	IBM®PowerPC®A2	Intel®Xeon Phi™ 7120P	IBM®PowerPC®970MP
Processor frequency	2.60 GHz	1.60 GHz	1.23 GHz (*max: 1.33 GHz*)	2.30 GHz
Peak performance/core (MIPS)	10,400	6,400	1,333	11,500
Chips/node (Cores/chip)	2 (8)	1 (16)	1(61)	2(2)
Memory/node (GBytes)	64	16	8	8

System name	**MareNostrum3**[5]	**Minotauro**[6]	**Tamariu**	**Yellowstone**[7]
Processor type	Intel®Xeon®E5-2670	Intel®Xeon®E5649	Intel®Xeon®E7450	Intel®Xeon®
Processor frequency	2.60 GHz (*max:3.30 GHz*)	2.53 GHz	2.40 GHz	2.60 GHz (*max:3.30 GHz*)
Peak performance/core (MIPS)	13,200	10,120	9,600	13,200
Chips/node (Cores/chip)	2 (8)	2 (6)	4 (6)	2 (8)
Memory/node (GBytes)	32	24	48	32

System name	**Experimental**	
Processor type	Intel®Core™ i7 2760QM	Intel®Xeon®E5-2620v2
Processor frequency	2.40 GHz	2.10 GHz
Peak performance/core (MIPS)	9,600	8,400
Chips/node (Cores/chip)	1 (4)	4 (6)
Memory/node (GBytes)	8	128

Table 5.3
Executions' characteristics. Summary of which applications have been executed on which systems, the number of processes and the sampling frequency.

Application	System	# processes	Sampling frequency
Arts_CF	MareNostrum3	512	20 Hz
BigDFT	Juqueen	1,024	20 Hz
	Experimental	21	50 Hz[8]
CESM	Yellowstone	570	50 Hz
CGPOP	MareNostrum2	24, 96, 192, 384	10 Hz
	Minotauro	24	10 Hz
	Experimental	24	50 Hz[8]
GTC	MareNostrum3	256	20 Hz
HydroC	Altamira	8	50 Hz
Mr. Genesis	MareNostrum2	8, 128, 256, 512	10 Hz
	Altamira	8	50 Hz
Nemo	MareNostrum3	128	20 Hz
NEST	Knights	120	20 Hz
PEPC	Tamariu	16	10 Hz
PMEMD	MareNostrum2	4, 64, 128, 256	10 Hz
Siesta	Altamira	128	50 Hz

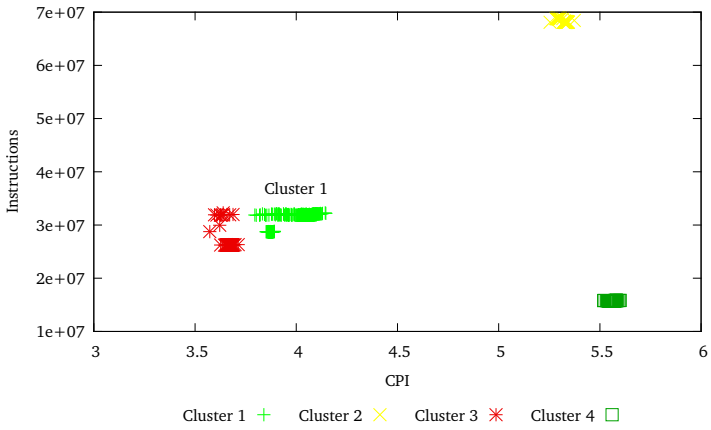
ease the reading of the plots, some of the analyses were manually changed, with the addition of labels and arrows to present the associated source code. Finally, in the event of proposing modifications to the source code, the framework was applied to the altered version in order to compare the analysis between the two versions of the code. So as to ease the comparison, the temporal dimension (X-axis) and the different metrics shown (on the Y-axes) were kept the same within an experiment.

5.1.1 CGPOP

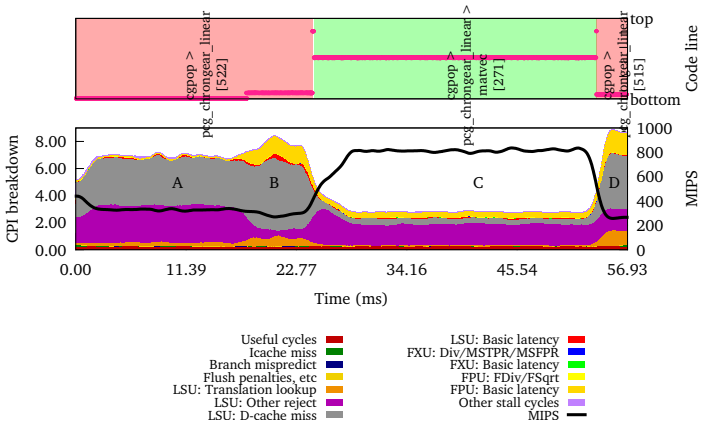
CGPOP [204, 203] is a proxy application of the Parallel Ocean Program [198] application. POP is a three-dimensional ocean circulation model designed primarily for studying the ocean climate system and a component within the Community Earth System Model (CESM) [98]. This application was analyzed in three directions using the same input. First, to understand its performance behavior and study whether there are optimization opportunities. Second, to examine how the application behaves on systems in which the processor shares resources such as the Last-Level Cache (LLC). Finally, the last experiment unveils the memory access patterns for the application data objects. The first analysis was ran in *MareNostrum2*, the second on *MinoTauro*, and the third on the *Experimental* system. The two first analyses focused on executions in which the application was using 24 MPI ranks and the instrumentation package configured to sample the application at 10 Hz in addition to capture MPI activity. With respect to the memory access pattern study, the application was executed twice to capture information regarding the load and store references, with the resulting plots shown side-by-side for comparison. Regarding the sampling periods in this last analysis, the one for collecting performance was coarser than the gprof sampling frequency (10 *vs* 20 ms) whereas memory sampling references were captured every 10^6 load (or store) instructions.

5.1.1.1 Analysis in MareNostrum2

Figure 5.1 shows the clustering results and the folding results for the region identified as Cluster 1. The clustering results reveal variability in terms of CPI for Cluster 1 and variability in terms of instructions for Cluster 3. Focusing on the most time-consuming region (Cluster 1), each invocation to this computing region executes approximately 3.3×10^7 instructions at 3.5 CPI. Also, an analysis of the trace-file indicates that Cluster 1 represents up to 84.8% of the total computation time.



(a) Clustering results.



(b) Detailed performance progression of Cluster 1.

Figure 5.1
Analysis of CGPOP in MareNostrum2.

With respect to the folding results, they include a profile of the source code references at the top diagram and the evolution of the MIPS rate and the CPIStack model in the plot below. In these results, the MIPS rate clearly identifies three phases: from the beginning to 23 milliseconds [ms], from 23 to 52 ms and from 52 ms to the end. However, the results of the performance model identifies four phases, which are more likely to exist according to the source code profile. The plot includes manually added labels (A - D) into the bottom plot to easily reference each of the phases in the text. The instruction pace ranges from 200 (in phases A, B and D) to 800 MIPS (in phase C), representing from 2%, up to 7% of the chip peak performance. In general, the primary cause for such low MIPS rate is due to the stalls caused in the load-store unit (LSU) and more precisely because of the data-cache misses as well as load/store instruction rejection. Still, the nature of the bottlenecks along phases varies. For instance, phases B and D expose an increase in the data-cache misses, in the FPU basic latency and also in terms of TLB misses.

The folding results show that the region occurs in the routine `pcg_chrongear_linear`, more precisely in lines 504-526 of file `solvers.F90`. The left column of Table 5.4 shows the code related to the delimited phases. Phase A involves the loop (lines 519-526) that is responsible for computing the solution and the residual for the next step. Phase B correlates with the loops

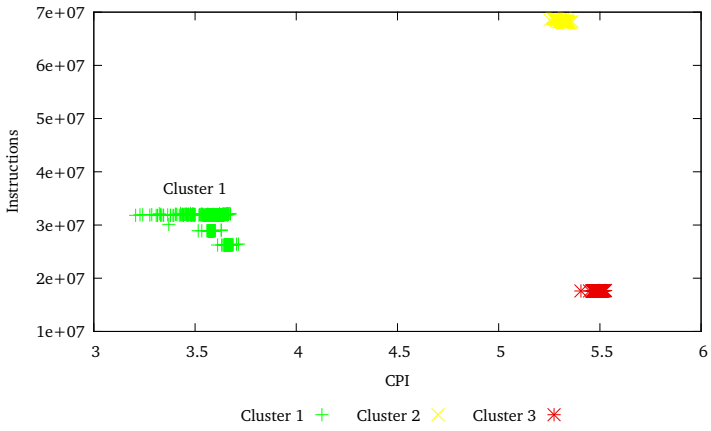
Table 5.4
Side-by-side comparison of both original and optimized CGPOP files. The highlighted regions refer to the main differences between the two.

Original solvers.F90

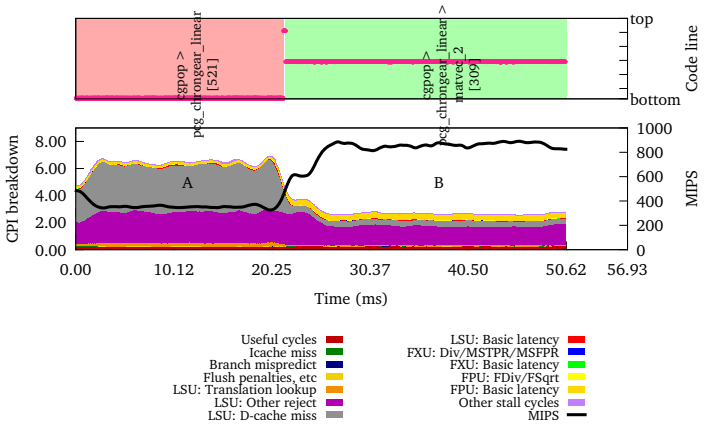
```
iter_loop: do m = 1, solv_max_iters
  sumN1=c0
  sumN3=c0
  do i=1,nActive
    Z(i) = Minv2(i)*R(i)
    sumN1 = sumN1 + R(i)*Z(i)
    sumN3 = sumN3 + R(i)*R(i)
  enddo
  do i=iptrHalo,n
    Z(i) = Minv2(i)*R(i)
  enddo
  call matvec(n,A,AZ,Z)
  sumN2=c0
  do i=1,nActive
    sumN2 = sumN2 + AZ(i)*Z(i)
  enddo
  call update_halo(AZ)
  ...
  do i=1,n
    stmp = Z(i) + cg_beta*S(i)
    qtmp = AZ(i) + cg_beta*Q(i)
    X(i) = X(i) + cg_alpha*stmp
    R(i) = R(i) - cg_alpha*qtmp
    S(i) = stmp
    Q(i) = qtmp
  enddo
end do iter_loop
```

Optimized solvers.F90

```
sumN1=c0
sumN3=c0
do i=1,nActive              ! Code replication
  Z(i) = Minv2(i)*R(i)
  sumN1 = sumN1 + R(i)*Z(i)
  sumN3 = sumN3 + R(i)*R(i)
enddo
do i=iptrHalo,n             ! Code replication
  Z(i) = Minv2(i)*R(i)
enddo
iter_loop: do m = 1, solv_max_iters
  sumN2=c0
  call matvec_2(n,A,AZ,Z,nActive,sumN2)  ! Code fusion
  call update_halo(AZ)
  ...
  sumN1=c0
  sumN3=c0
  do i=1,n
    stmp = Z(i) + cg_beta*S(i)
    qtmp = AZ(i) + cg_beta*Q(i)
    X(i) = X(i) + cg_alpha*stmp
    R(i) = R(i) - cg_alpha*qtmp
    S(i) = stmp
    Q(i) = qtmp
    Z(i) = Minv2(i)*R(i)          ! Loop
    if (i <= nActive) then        ! fusion
      sumN1 = sumN1 + R(i)*Z(i)
      sumN3 = sumN3 + R(i)*R(i)
    endif
  enddo
end do iter_loop
```



(a) Clustering results.



(b) Detailed performance progression of Cluster 1.

Figure 5.2
Analysis of the modified version of CGPOP in MareNostrum2.

in lines 504-511, where the first loop computes two reductions (on `sumN1` and `sumN3`) and a vector multiplication and the second loop computes a vector multiplication. Phase C calculates the matrix-vector product where the matrix is stored in Compressed-Sparse-Row format through the `matvec` call in line 512. Afterwards, the application reduces the result of the matrix-vector product in phase D (on the variable `sumN2`) in the loop in lines 514-516. Also, the compiler optimization report shows that the compiler has unrolled four times the loops in lines 504-508 (phase B) and 514-516 (phase D). The compiler also states that these loops cannot take advantage of vectorization because the operands, which are 64-bits long, are not suitable for using the Altivec instruction set. Finally, dependencies do not allow improvements to either the performance of the calculation of the solution and residual (phase A) or the matrix-vector product (phase C). As a result, the reductions can be computed as the data are being generated.

In order to shorten the execution time, the code is altered so that the `matvec` subroutine also computes the reduction, as described in the original loop at lines 514-516 by merging the two loops. The idea behind this modification is to combine two regions with different types of bottlenecks to reduce the pressure on the affected units so that they have more time to respond. Further inspection of the code shows that vector `Z` (which ranges from 1 to n) is divided into

Table 5.5
Execution time for the CGPOP application using larger data sets.

# processes	Time w/o opt (s)	Time w/ opt (s)	Improvement
96	686	636	7.2%
192	417	387	7.2%
384	263	262	<0.1%

two parts and each is treated differently. The first part is kept within indices 1 to `nActive`, whereas the second part is stored within indices `nActive+1` and *n*. This allows the loop in lines 504-511 to be embedded into the body of the loop found in lines 519-526 but requires adding the appropriate conditionals and a prologue before the `iter_loop` loop executes, so as to maintain the semantics of the original code. Again, this change leads to intersperse the floating-point instructions in between the loop traversal so that the pressure on the FPU decreases as there are other instructions to execute before using its results. Table 5.4 summarizes all these changes applied to the source code by comparing the original source code (on the left) and the modified source code (on the right).

Figure 5.2 shows the results for the clustering and folding tools after applying the aforementioned modifications. The clustering results reveal that one of the computation groups merged with others. Since the changes were applied to Cluster 1, it is very likely that the original Clusters 1 and 2 merged together as a result of closer performance metrics. The analysis of the folding results show only two phases (A and B) that last 23 and 27 ms and run at 350 and 860 MIPS (which corresponds to 3% and 7.5% of the peak performance), respectively. These results indicate that the original phases B and D were integrated into the original phases A and C. With all these improvements, the overall duration of the region decreased by 6 ms and the application ran 10% faster that in its original version.

The aforementioned optimizations have shown benefits on CGPOP by improving the serial node performance when using a reduced number of MPI processes, but it is worth checking whether these modifications are valuable for larger core count executions. Table 5.5 shows a comparison of the wall-clock time for the best of three runs of the original and modified code using a number of processes that represent a typical production workload in *MareNostrum2*. The Table outlines that the optimized version of CGPOP outperforms the original version in 7.2% when using 96 and 192 processes but does not improve it when executing on 384 tasks. Although the parallel scalability is outside the topic of this thesis, a preliminary analysis of the execution using 384 processes that the application invests about 110 seconds in the initialization phase. Also, approximately 18% and 22% of the total time is devoted to `MPI_Waitall` and `MPI_Allreduce` in the main computation phase.

5.1.1.2 Analysis on the impact of shared resources

Nowadays, it is very common for a single processor to contain several computation units (namely, cores) that share some of the chip resources. The most notable example is the LLC, which is the last-standing cache in the path to the main memory. Sharing resources poses some difficulties when analyzing applications because it is unclear during execution how resources distribute and interact among the processes that run on the system. This section serves as a little appendix to show how this type of analysis could be conducted applying the folding mechanism on CGPOP.

While CGPOP is an SPMD application because each MPI process shares the same application code, the input data does not uniformly distribute among processes. This results in an unbalanced run in which some processes finish earlier and need to wait for the remaining processes. The unbalance occurs systematically during the whole application execution and remains constant along time because CGPOP does not implement any adaptive work balance mechanism. Therefore, the clustering tool groups unbalanced computation regions so that each unbalanced group represents a single MPI process (or core).

In order to proceed with this experiment, the original version of CGPOP was executed using 24 MPI processes on four nodes of *Minotauro*. Each node of *Minotauro* consists of two hexa-core chips and each chip has three levels of cache and access to main memory. The CGPOP processes

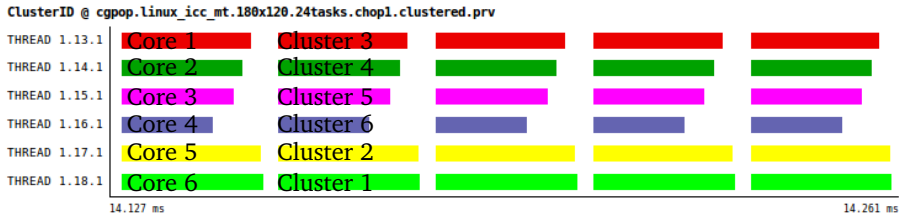


Figure 5.3
Periodic work unbalance in CGPOP. The time-line shows in color the clusters along time and has been extended with labels that identify clusters and the cores where the activity run.

were pinned to the cores of one of the two chips within the node. This way of executing the application not only keeps all the processes accessing to the same private cache memory but also ensures that the other chip does not compete to access to the memory bus. With respect to the cache structures, the first level consists of independent 32 Kbyte instruction and data caches per core. The second level is a 256 Kbyte unified cache per core. The Last-Level Cache (LLC, L3) is a unified cache of 12,288 Kbyte shared among all the cores and the maximum memory bandwidth is rated at 32 GB/s[9]. Figure 5.3 shows resulting trace-file obtained for this experiment. The Figure consists of a time-line showing the work unbalance, the resulting clusters, as well as the core pinned to a particular MPI rank (and therefore, cluster).

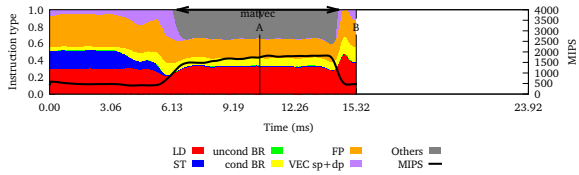
Figures 5.4 and 5.5 shows the folding results of the six cores of one node. More specifically, Figure 5.4 represents the detailed evolution of the instruction rate and the instruction break-down along the region. On the other hand, Figure 5.5 shows the progression of the MIPS rate and the ratio of cache misses per instruction at different levels of the memory hierarchy. In the Figures, the plots are sorted according to their required time to execute. The plot at the top (bottom) shows the core/cluster that takes the less (most) time to execute.

An exploration of the trace-file indicates that these clusters occur immediately after an `MPI_Allreduce` invoked from the routine `global_sum`. This MPI operation involves all the processes, so these clusters are likely to start at the same time after the MPI call finishes because the way this MPI call is implemented. All the plots use the time-scale of the longest computation region (Cluster 1, executed in core 6) so as to ease the comparison. The plots are marked manually with labels (A to E) to denote five points of interest regarding the interaction between cores. The plots also include a label that represents the execution of the `matvec` routine, which in turn, exposes a uniform behavior with respect to the instruction decomposition.

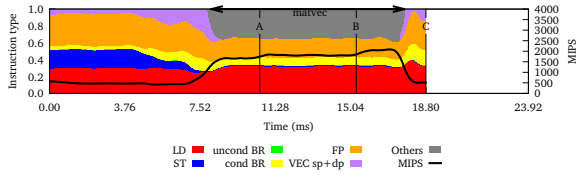
Point A in Clusters 4, 5 and 6 (which execute on cores 2, 3, 4, respectively) experience a subtle increase on the instruction rate according to Figures 5.4a, 5.4b and 5.4c. This increase occurs within the `matvec` routine, which has a constant instruction break-down along its execution so the performance improvement is unlikely to occur because a change in the application. Notice that point A also represents the time when Cluster 1 finishes executing the code with a portion of store instructions (Figure 5.4f) actively using all the cache levels (as seen in Figure 5.5f). Therefore, the stores executed in core 6 alter (or pollute) the shared-cache among all cores.

It is worth remembering that the computing regions finish when the core invokes an MPI call. In this case, CGPOP establishes communication to other processes by invoking `MPI_Irecv`, `MPI_Isend` and waiting for the completion of these messages through `MPI_Waitall`. Points B, C, D and E represent the times when cores four, five, two and one finish their computing region, respectively. At each of these points, one core starts waiting and stalls the processor until data arrives from the network; so freeing up the resources it had allocated. For instance, when Cluster 6 finishes (point B), the instruction rate on the remaining cores increase without any representative change on their work. The very same behavior is repeated at points C, D and E. When the two last executing cores (five and six) run alone (at point E), their instruction rate in `matvec` double the instruction rate observed when the routine begins. In fact, the misses in LLC

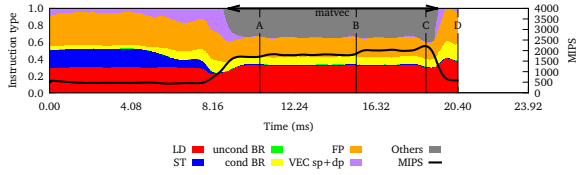
[9]According to the specifications of the processor indicated in: `http://ark.intel.com/products/52581/Intel-Xeon-Processor-E5649-12M-Cache-2_53-GHz-5_86-GTs-Intel-QPI` - Last accessed March, 2015.



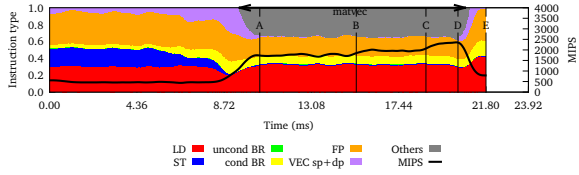
(a) Core 4 (Cluster 6).



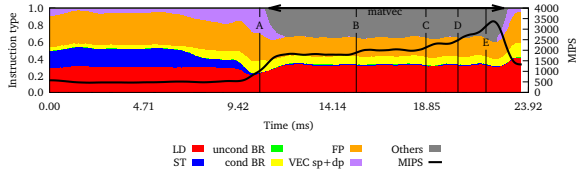
(b) Core 3 (Cluster 5).



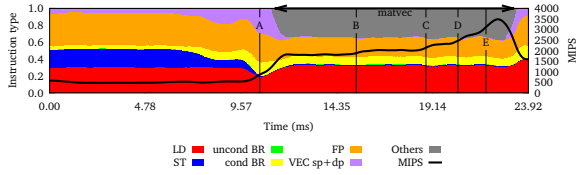
(c) Core 2 (Cluster 4).



(d) Core 1 (Cluster 3).

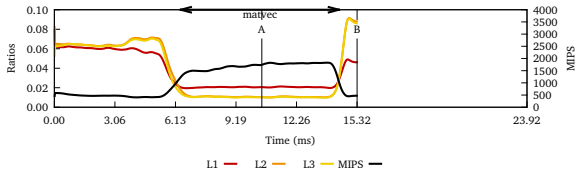


(e) Core 5 (Cluster 2).

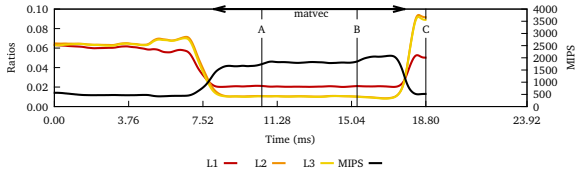


(f) Core 6 (Cluster 1).

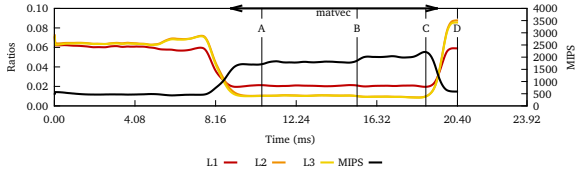
Figure 5.4
MIPS rate and architecture impact model on each core of the hexa-core chip. The clusters/cores are sorted according to their duration (shortest at the top).



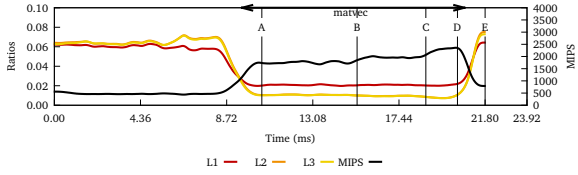
(a) Core 4 (Cluster 6).



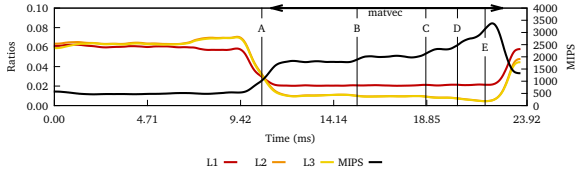
(b) Core 3 (Cluster 5).



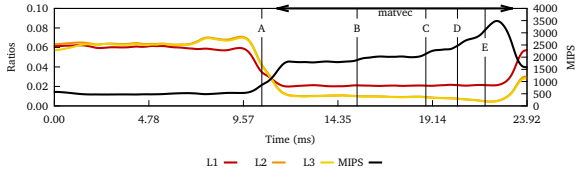
(c) Core 2 (Cluster 4).



(d) Core 1 (Cluster 3).



(e) Core 5 (Cluster 2).



(f) Core 6 (Cluster 1).

Figure 5.5
MIPS rate and architecture impact model on each core of the hexa-core chip. The clusters/cores are sorted according to their duration (shortest at the top).

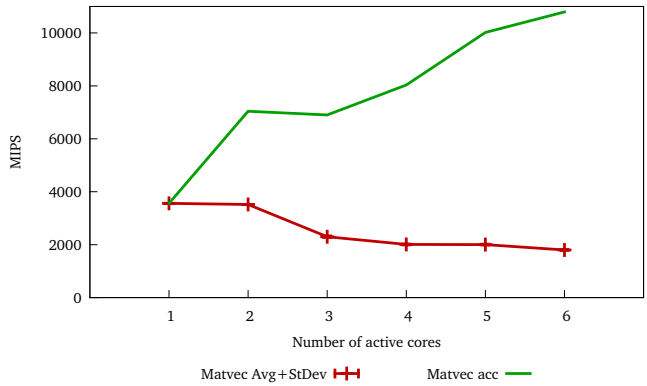


Figure 5.6
Aggregated and average MIPS achieved depending on the active cores executing `matvec`.

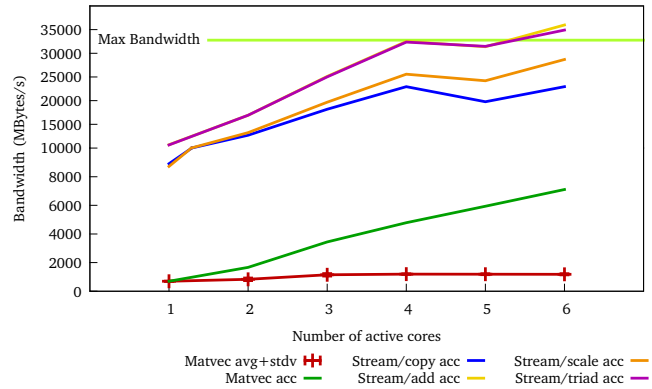


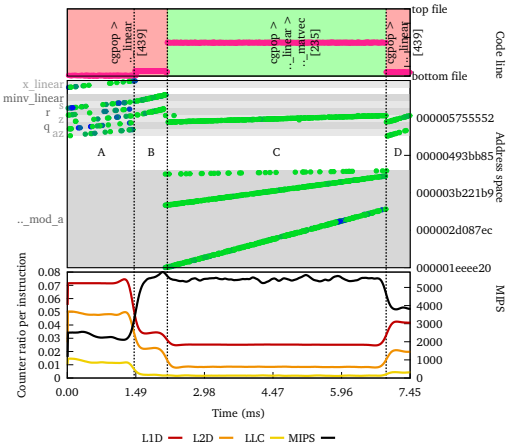
Figure 5.7
Aggregated and average MIPS achieved depending on the active cores executing `matvec`.

(but not in L1D or L2D) per instruction ratio start decreasing from point C and reach a significant reduction at point E, where only two cores share the LLC.

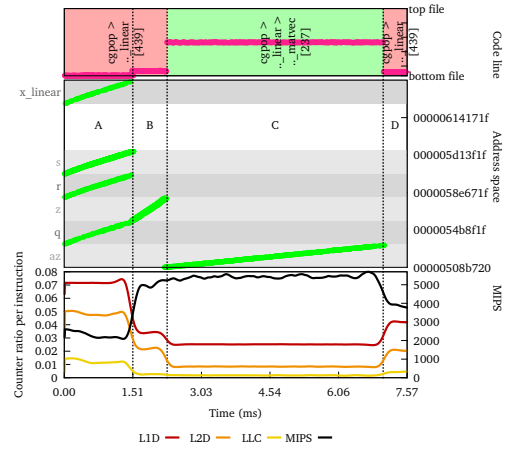
A comparison between cores allows further analyses point out the interactions between the processes. Figure 5.6 shows both the accumulated and average (plus standard deviation) MIPS according to the active cores when executing `matvec` simultaneously. According to the plot, the more cores that execute the routine, the less the average MIPS rate achieved per core, even though the accumulated instruction rate increases. The most significant change occurs when increasing the active cores from two to three. The reader may wonder why the performance achieved when one core a single core does not increase further. The case studied here shows that the two processes with more work take approximately the same amount of time to execute; so, the most unbalanced process does not have enough time to take benefit from all the resources when running alone.

On the other hand, Figure 5.7 shows the memory bandwidth used. The plot shows the accumulated and average (plus standard deviation) bandwidth to memory according to the active cores when executing `matvec` concurrently. The used bandwidth at a given time t is calculated according to:

$$BW(t) = \sum_{c=1}^{activecores} LLC_{MissRate}(t,c) \times CacheLine \tag{5.1}$$



(a) Load references.



(b) Store references.

Figure 5.8
Source code, performance and memory references analysis for the CGPOP application.

where $LLC_{MissRate}(t, c)$ refers to the LLC miss rate at given t observed in core c and CacheLine is 64 bytes. For comparison purposes, the plot additionally shows the memory bandwidth reported by the best of five runs of the Stream benchmark [141] using from one to six threads on a chip and also the theoretical maximum memory bandwidth. The results of CGPOP show an increase of the memory bandwidth used with a significant increase when increasing the active cores from two to three, as it inversely occurred in the MIPS rate. The plot also shows that the system is capable of delivering data from memory to the processor at a much higher rate. In conclusion, the analysis presented shows that sharing the cache (or in other words, reducing the cache per core) penalizes the application performance while sharing the bandwidth to main memory does not interfere with the application performance.

5.1.1.3 Analysis of memory access patterns

Figure 5.8 shows the obtained plots depicting the load and store references for the most time-consuming region of this execution. Each plot has its own address space depending on the accessed variables. Note too that the store memory references are shown in green because

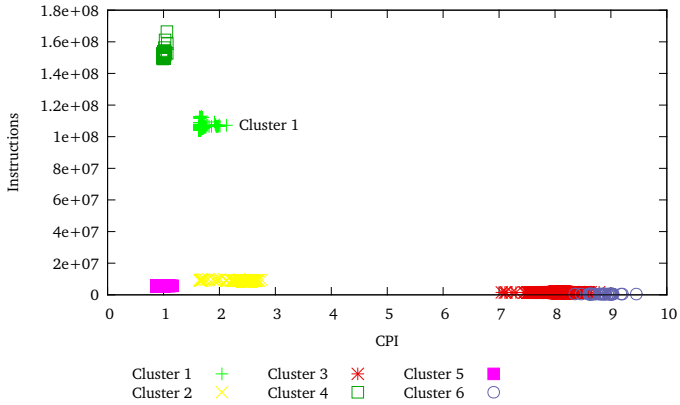
in this architecture the store instructions are inserted into a store buffer and when the store instructions go to memory they are no longer under control of PEBS. The Figure indicates that the region faces two routines: `pcg_chrongear_linear` (in red) and `matvec` (from the matrix module, in green), but the plots have been manually extended by adding labels (A-D) to ease the referencing. With respect to the load instructions within the data structures, it may be observed that phase C accesses to variables `z` and `a` (from the matrix module). The plot shows that the load references to variable `a` are partitioned into three disjoint portions that are accessed linearly and simultaneously by the processor. The analysis of the source code shows that this variable represents a sparse row matrix that includes three arrays (one for double precision values and two for integer indices). Phase A shows that the references require more time to be served (blue colored) and this is also related to the highest ratio of cache misses (1 out of every 14 instructions miss at L1D). The code in this phase loads data from six arrays (`x_linear`, `s`, `r`, `z`, `q` and `az`) and stores data to four arrays (`x_linear`, `s`, `r` and `q`). The code of the application was modified for the purposes of using an array of structures (AoS) in order to check whether this improved the performance. However, after creating a small test case that resembles the application behavior the timings indicate that using AoS does not offer performance improvements because the LLC miss ratio and the number of instructions doubles. With respect to the stores (Figure 5.8b), the results show several effects. First, phase B generates the data for the array `z` and is used immediately afterwards in phase C. Second, the `a` variable keeps unchanged during this region. Finally, phase D does not expose stores because it reduces a vector into a scalar (`sumN2`).

5.1.2 PMEMD

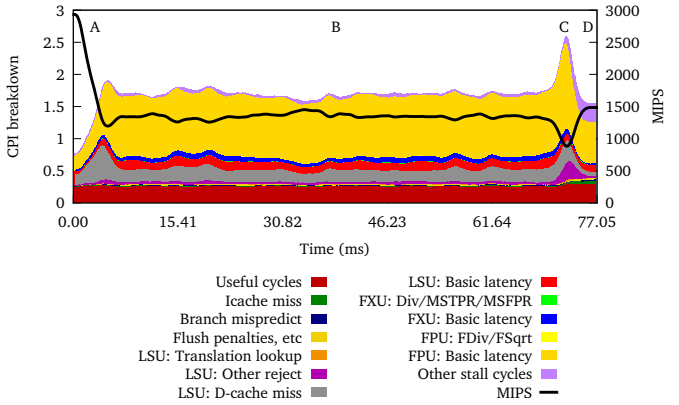
PMEMD is an application that belongs to the AMBER suite [214, 184]. AMBER is a set of molecular mechanical force fields for the simulation of bio-molecules and a package of molecular simulation programs. The scatter-plot shown in Figure 5.9a depicts the grouping of the computation regions of the application when executed in *MareNostrum2* using 4 MPI processes. The obtained results indicate that all clusters present variability in terms of either instructions or CPI. In average, each execution of Cluster 1 commits 1.1×10^8 instructions running at 1.75 CPI. The analysis of the trace-file extended with clusters information indicates that Cluster 1 takes up to 71.3% of the execution time.

Figure 5.9b shows the folded results for Cluster 1 combining the MIPS rate and breaking down the CPI according to the CPIstack model for the IBM PowerPC 970MP processor. The result in the Figure shows four phases (manually labeled as A-D in the plot) according to the resulting CPI break-down. The most dominant phase in the computation region is phase B, which runs at 1,400 MIPS (about 12.1% of the maximum rate) and lasts 58 ms approximately. The CPI break-down of this phase is uniform across time and the FPU is the execution unit that accounts most part of the CPI. According to the results, multi-cycle floating-point operations (labeled as FPU: FDiv/FSqrt) and the latency of the FPU units (labeled as FPU: Basic latency) are responsible for one of the total CPI (1.7) achieved within this phase.

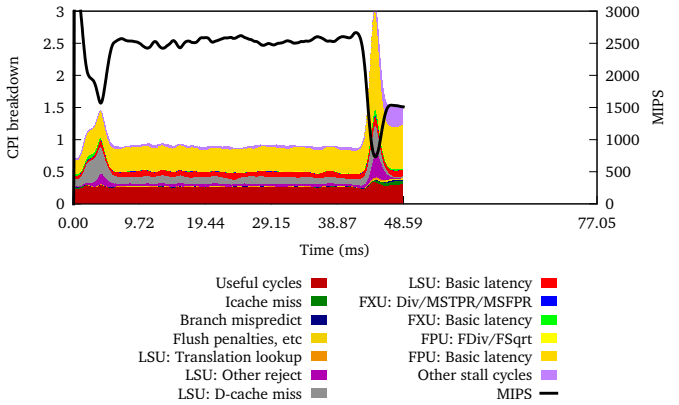
The framework identifies that the code executed in this phase refers to the subroutine `pairs_calc_efv` and more precisely by the loop contained in lines 691-748 of file `pme_direct.f90`. The report produced by the compiler shows that the loop contains non-vectorizable reductions but the compiler unrolled it four times. An analysis of the code of the loop shows that it contains dependencies on a square root and its inverse at the very beginning of the loop body as shown in the left column of Table 5.6. According to the results in Figure 5.9b, the analyst has to address the latency on the FPU to reduce the total CPI in this phase to increase the instruction rate. In order to reduce the pressure on the FPU, the memoization technique is suggested so as to increase the distance between the data generated by the FPU and its use. To apply this technique it is necessary to create temporal lookup tables to store the pre-computed values of the long latency operations and calculate these values before executing the conflicting loop. More precisely, two lookup tables called `sqrt_delr2` and `inv_delr2` of `vec_max` size are created. These tables are accessed in the loop in lines 685-690 to store the pre-computed square root and the inverse of the square root of the element being accessed. The right column of Table 5.6 shows the summarized changes applied the source code. Although this solution increases the distance between the floating-point operations, it has three drawbacks *a priori*: an increase in



(a) Clustering results.



(b) Detailed performance progression of Cluster 1.



(c) Detailed performance progression of Cluster 1 after modifying the source code.

Figure 5.9
Analysis of PMEMD.

Table 5.6
Side-by-side comparison of both original and optimized PMEMD files.

Original pme_direct.f90

```
nxt_cnt = 0
do vec_idx = 1, vec_max
  if (delr2_vec(vec_idx) .lt. max_nb_cut2) then
    nxt_cnt = nxt_cnt + 1
    nxt(nxt_cnt) = vec_idx
  end if
end do




do nxt_idx = 1, nxt_cnt
  vec_idx = nxt(nxt_idx)
  delr2 = delr2_vec(vec_idx)
  img_j = img_j_vec(vec_idx)
  cgi_cgj = cgi * img_qterm(img_j)
    delr = sqrt(delr2)
    delrinv = 1.d0 / delr
    x = dxdr * delr
    ind = int(eedtbdns_stk * x)
    dx = x - dble(ind) * del
    ind = ishft(ind, 2)
    ...
```

Optimized pme_direct.f90

```
nxt_cnt = 0
do vec_idx = 1, vec_max
  if (delr2_vec(vec_idx) .lt. max_nb_cut2) then
    nxt_cnt = nxt_cnt + 1
    nxt(nxt_cnt) = vec_idx
    sqrt_delr2(nxt_cnt) = sqrt(delr2_vec(vec_idx))
    inv_delr2(nxt_cnt) = 1.d0/sqrt_delr2(nxt_cnt)
  end if
end do

do nxt_idx = 1, nxt_cnt
  vec_idx = nxt(nxt_idx)
  delr2 = delr2_vec(vec_idx)
  img_j = img_j_vec(vec_idx)
  cgi_cgj = cgi * img_qterm(img_j)
    delr = sqrt_delr2(nxt_idx)      ! Scalar
    delrinv = inv_delr2(nxt_idx)    ! expansion
    x = dxdr * delr
    ind = int(eedtbdns_stk * x)
    dx = x - dble(ind) * del
    ind = ishft(ind, 2)
    ...
```

Table 5.7
Execution time for the PMEMD application using larger data sets.

# processes	Time w/o opt (s)	Time w/ opt (s)	Improvement
64	929	824	11.3%
128	635	563	11.3%
256	631	560	11.2%

the total executed instructions, an increase in the number of accesses to the memory hierarchy and a higher memory consumption. Regarding the memory accesses, the folding results shows that the original application do not have issues regarding the LSU; so, there is room to stress this functional unit. With respect to memory consumption, creating the buffers does not pose a problem because the new structures are small in size (the value of `vec_max` is 128).

The plot depicted in Figure 5.9c shows the result when applying the stated transformation. The results indicate a reduction of the overall CPI to a value close 1 when it was about 1.7 for the analyzed phase. According to the results, there is a reduction in the FPU that results into a higher MIPS rate (from 1,400 to 2,600), but still far (22.6%) from the peak rate. As expected, the modification applied increases the number of instructions executed by 8%. However, the duration of the computation region is 36.7% shorter and the overall gain is approximately 26%.

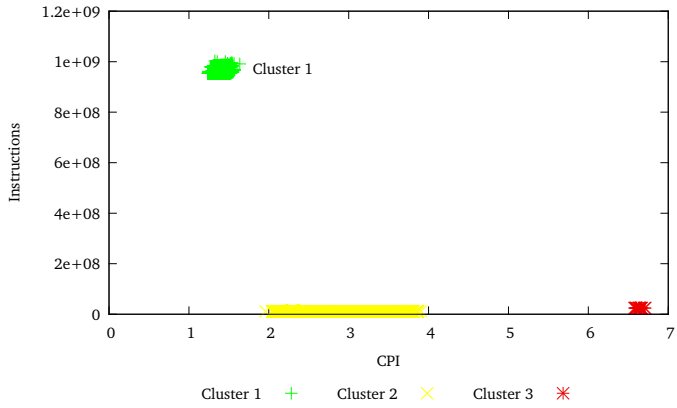
Although the analysis of the application shows benefits at small scale, it is interesting to study whether the suggested modifications impact executions with larger core-count. Table 5.7 tabulate the best wall-clock timing measurements obtained from three runs of PMEMD on *MareNostrum2* using a representative large input. The results show two effects. First, the changes proposed result in a faster version with a constant increase (11.3%), regardless of the number of MPI tasks used during the execution. And second, despite the improvement on the serial-node performance, the application does not scale at 256 processes.

5.1.3 Mr. Genesis

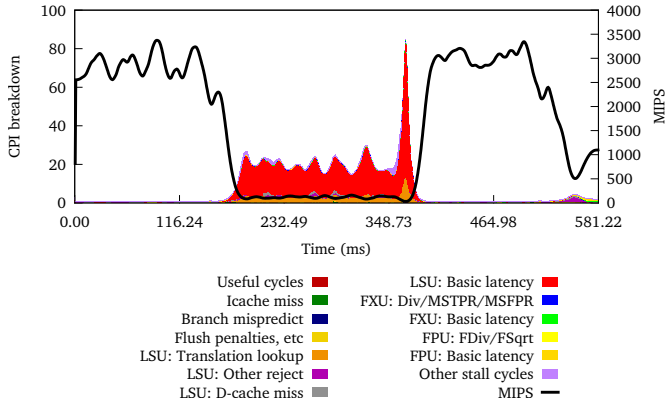
Mr. Genesis [150] employs a finite volume approach in order to evolve the Relativistic Euler equations combined with a Constrained Transport scheme to account for the divergence free evolution of the dynamically included magnetic field. The application was executed in *MareNostrum2* and the clustering tool reported that the application had three types of computing regions according to their performance, as depicted in the resulting scatter-plot in Figure 5.10a. Cluster 1 represents up to 87% of the application execution time and each instance of this computing region executes an average of 9.7×10^8 instructions running at 1.4 CPI.

The plot shown in Figure 5.10b illustrates the CPI break-down evolution for Cluster 1, as well as the MIPS rate. These results obtained show four distinguishable phases (manually labeled A to D) according to the instruction rate achieved. Phases A and C show the best instruction rate and run approximately at 3,000 MIPS (about 26% of the peak rate), whereas phases B and D run close to 100 and 500 MIPS, respectively. The CPI achieved in phase B is typically above 20 with two peaks, the slowest of which achieves a CPI value of 90. The CPI break-down in this phase is clearly dominated by TLB misses (labeled as LSU: Translation lookup) and by the latency of the memory hierarchy (labeled as LSU: Basic latency). With respect to the source code, phase A refers to the `sweepx` routine, phases B and C correlate with routine `sweepy` and phase D points to the routine `step`.

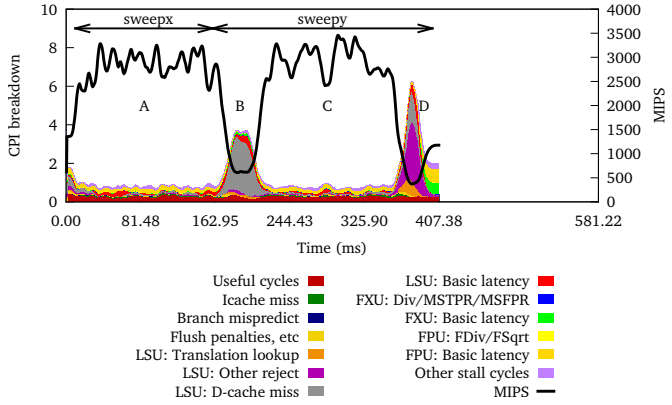
Since phase B is the longest and presents the worst performance within the region, the analyst should focus it to improve the application performance. The folding results correlate phase B with the routine `sweepy`, more precisely with the 2D loop in lines 410-421 of file `sweep.f`. This loop, as shown in the left column of Table 5.8, performs matrix transpositions in the loop in which the data are read from two 3D matrices named `primit` and `conser` and stored into multiple 2D matrices. In addition, it includes Fortran90 intrinsic functions that transpose data from `tracer` and `trflux` into `tracerT` and `trfluxT`. Each cell of the 3D matrices points to five-field structures, so the sequential accesses to these structures ensure spatial locality. However, the store instructions do not exploit spatial locality because of the construction of the loop.



(a) Clustering results.



(b) Detailed performance progression of Cluster 1.



(c) Detailed performance progression of Cluster 1 after modifying the source code.

Figure 5.10
Analysis of Mr. Genesis.

Table 5.8
Side-by-side comparison of both original and optimized Mr. Genesis files.

Original sweep.f90

```
410  do jj = fzn, nzn
411    do ii = fzntt, nzntt
412      denstyT(jj,ii) = primit(ii, jj, j)%densty
413      presT(jj,ii)   = primit(ii, jj, j)%pres
414      velxT(jj,ii)   = primit(ii, jj, j)%velx
415      tracerT(1:nspec, jj, ii) = tracer(1:nspec, ii, jj, j)
416      trfluxT(1:nspec, jj, ii) = trflux(1:nspec, ii, jj, j)
417      momenyT(jj,ii) = conser(ii, jj, j)%momeny
418      momenzT(jj,ii) = conser(ii, jj, j)%momenz
419      energyT(jj,ii) = conser(ii, jj, j)%energy
420    enddo
421  enddo
```

Optimized sweep.f90

```
410  do ii = fzntt, nzntt       ! Loop
411    do jj = fzn, nzn         ! interchange
412      denstyT(jj,ii) = primit(ii, jj, j)%densty
413      presT(jj,ii)   = primit(ii, jj, j)%pres
414      velxT(jj,ii)   = primit(ii, jj, j)%velx
415      tracerT(1:nspec, jj, ii) = tracer(1:nspec, ii, jj, j)
416      trfluxT(1:nspec, jj, ii) = trflux(1:nspec, ii, jj, j)
417      momenyT(jj,ii) = conser(ii, jj, j)%momeny
418      momenzT(jj,ii) = conser(ii, jj, j)%momenz
419      energyT(jj,ii) = conser(ii, jj, j)%energy
420    enddo
421  enddo
```

Table 5.9
Execution time for the Mr. Genesis application using larger data sets.

# processes	Time w/o opt (s)	Time w/ opt (s)	Improvement
128	3,036	2,001	34.6%
256	1,332	815	38.8%
512	526	353	32.8%

The compiler reports that lines 415 and 416 cannot use vector instructions for two reasons. First and most important, the elements accessed by the statements on these lines are 64-bits long (i.e. they are double precision floating-point). This type of data is unsupported by the vector instruction set of the processor, which only supports 32-bit floating-point arithmetic. Second, the intrinsic functions reference the memory with non-vectorizable alignments, probably because these buffers are marked in the source code as `allocatable`. According to the results shown in the plot and the observed application source code, the loop interchange technique is used to improve the data locality on the store instructions. It is worth mentioning that the optimization option `-O3` on the IBM XL compiler automatically turns on the `-qhot=level=0` option. This optimization flag enables high-order transformations such as early distribution, loop interchange and loop tiling to the optimizations applied; however, there is no reference in the optimization report regarding the application of the loop interchange technique. The loop interchange optimization cannot be applied indiscriminately and requires an analysis of the data dependencies. For the case presented here, it is safe to apply this optimization technique, though. Applying loop interchange in the 2D loop, as depicted in the right column of Table 5.8 only reduces the locality on the load instructions in the outer loop. However, this modification does not improve the locality at the inner loop because the access to the each field of the structure preserves its spatial locality.

Figure 5.10c shows the results of the framework for the transformed version of the code. Notice that the duration of the region decreases from 592.69 ms to 386.09 ms, which is approximately 34% of reduction and the whole application runs 30% faster than the baseline. The decrease on the execution time comes from the improvement on phase B, which now runs at 500 MIPS (i.e. a CPI close to 4). Still the main stall in this region is caused by the load/store unit though it is significantly less after applying the modifications.

As in earlier analyses, it is worth evaluating whether these changes still improve executions using a larger number of processes. Table 5.9 compares the elapsed time using a fixed input with and without applying the aforesaid such changes to the code. As presented in the Table, the optimized version of Mr. Genesis shows a improvement compared to the original version and shows improvements that range from 32% to 38%. It is also worth noting that executing Mr. Genesis with a fixed input shows a super-linear speed-up when increasing the number of processes using either the optimized or the unmodified versions. This effect is likely to indicate that the more processors that execute the application, the more application working set fits in the cache memory.

5.1.4 BigDFT

BigDFT [79] is a massively parallel electronic structure code based on density functional theories (DFT) using a wavelet basis set. Wavelets form a real space basis set distributed on an adaptive mesh (two levels of resolution in the BigDFT implementation). BigDFT was analyzed in two directions: by exploring its performance behavior and study performance optimizations possibilities and by studying the memory access patterns for the application data objects. The first analysis was made in *Juqueen* using 1,024 MPI processes and the instrumentation package was configured for the application at 10 Hz in addition to capture the MPI activity. The second analysis was executed in the *Experimental* system using 21 MPI processes and used the combined Extrae and perf monitoring system to collect information from the MPI activity and the memory references. Because of the difference in the system scale size, the application input was different

for each execution. Regarding the sampling periods in the last analysis, the sampling period is set to 50 Hz and memory sampling references were captured every 10^6 load (or store) instructions.

5.1.4.1 Analysis in Juqueen

Figure 5.11a shows the clustering scatter-plot that groups the most time-consuming computing regions of this execution. Each invocation to the region labeled as Cluster 1 executes approximately 3.2×10^9 instructions and executes at 3.8 CPI (that translates into 7,588 ms per invocation), for a 47% of the total execution time. The folding mechanism depicts the internal evolution of the instruction rate and the ratio between the instruction counter and the L1D miss rates within Cluster 1. The result, illustrated in Figure 5.11c, indicates that the user code region faces seven phases in terms of MIPS (labeled as Phase 1-7 within the plot). Phases 2-7 represent more than the 90% of the whole region and run between 370 and 430 MIPS, which is approximately 7% of the peak core performance. The ratio between the MIPS rate and the L1D miss rate reveals that approximately 7% of the instructions on phases 3, 5 and 7 miss in L1D while *circa* 4% of the instructions on phases 2, 4 and 6 miss in L1D.

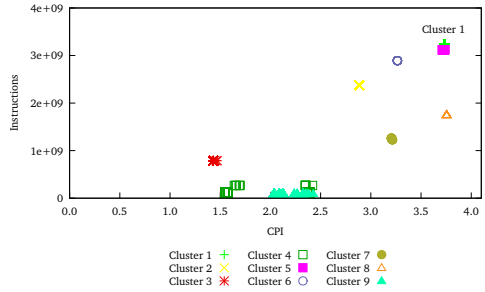
With respect to the source code responsible for such bad performance, the CUBE visualization points out the source code locations of these bottlenecks as shown in Figure 5.11b. On the one hand, the annotations on the tool pinpoint that phases 2, 4 and, 6 correlate with the loop in lines 387-397 of file `convolut_common_slab.f90` within the `convolut_kinetic_slab_sdc` routine three times. On the other hand, phases 3, 5 and 7, refer to the loop in lines 427-437 of the aforementioned file and routine. The source code shows that the code blamed is a nested loop. These loops are written in such a way that they are executed consecutively, which indicates that there is an external loop (or sequence or instructions) that invokes the routine `convolut_kinetic_slab_sdc`.

An analysis of the loops pointed out by the CUBE visualization indicates that the loop in lines 427-437 does not exploit the data locality because it does not access to contiguous elements of matrix `x`. To improve the memory hierarchy efficiency and therefore the performance, the loops indices are swapped to exploit the cache locality better. As a consequence of the loop reordering, the resulting code requires additional statements to maintain the original semantics of the application. More precisely, the original order of the loops was written as `{i2, i1, i3, l}` and the changed loop order becomes `{i3, l, i2, i1}`. Since the loop in lines 387-397 shows a similar code with the same diagnostic, an analogous solution applies.

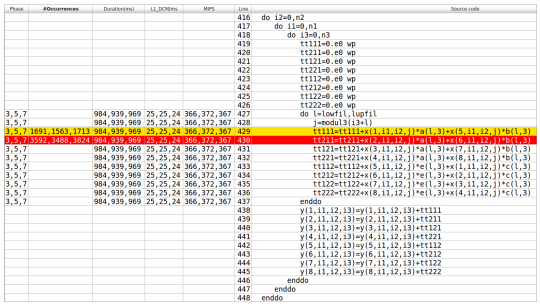
So as to proceed with a comparative analysis, the same analysis was applied to the application after applying the preceding changes. The results of the new analysis reveal that the duration of this computing region had decreased from 7,588 to 4,945 ms despite the increase in the number of instructions executed (from 3.2×10^9 to 3.7×10^9). The computing region was therefore executed at a faster instruction rate. In terms of the clustering scatter-plot (not shown), the computing region has moved a bit upwards and significantly to the left. Figure 5.11d shows the result of the folding mechanism when applied to the modified binary and using the same time-scale as in the original results. The Figure reveals that phases 2, 4 and 6, improved their L1D use because less than 2% of the instructions miss at this level of the memory hierarchy. The improved use of the memory hierarchy resulted in a performance boost that increased the instruction rate from 430 to 880 MIPS. Similarly, phases 3, 5 and 7, also experienced a performance increase running approximately at 660 MIPS and where 5% of the instructions miss in L1D. Overall, the application execution time was cut by 14.6% when applying the stated modifications.

5.1.4.2 Analysis of memory access patterns

Figure 5.12 shows the collocated plots depicting the source code information observed, the addresses referenced (either by loads or stores) and the performance achieved for a region of the application that corresponds to approximately 16% of the application execution time. The time-based results indicate that the region consists of two iterations at all levels (source code, references and performance) and the plots have been manually modified to identify the iterations as well as the phases (routines) within the iterations. The results indicate that the load references

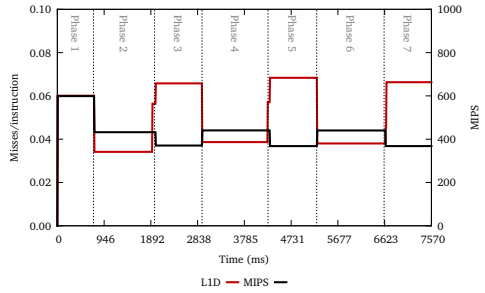


(a) Clustering results.

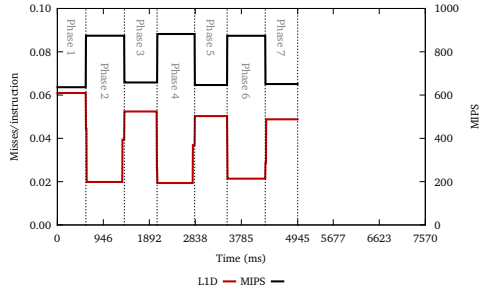


Phase	#Occurrences	Duration(ms)	L1_DCM/ms	MIPS	Line	Source code
					416	do i2=0,n2
					417	do i1=0,n1
					418	do i3=0,n3
					419	tt111=0.e0 wp
					420	tt211=0.e0 wp
					421	tt121=0.e0 wp
					422	tt221=0.e0 wp
					423	tt112=0.e0 wp
					424	tt212=0.e0 wp
					425	tt122=0.e0 wp
					426	tt222=0.e0 wp
3,5,7		984,939,969	25,25,24	366,372,367	427	do l=lowfil,lupfil
3,5,7		984,939,969	25,25,24	366,372,367	428	j=modul3(i3+l)
3,5,7	1891,1563,1713	984,939,969	25,25,24	366,372,367	429	tt111=tt111+x(i1,i1,i2,j)*a(l,3)+x(5,i1,i2,j)*b(l,3)
3,5,7	5592,4163,5224	984,939,969	25,25,24	366,372,367	430	tt211=tt211+x(7,i1,i2,j)*a(l,3)+x(1,i1,i2,j)*b(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	431	tt121=tt121+x(3,i1,i2,j)*a(l,3)+x(7,i1,i2,j)*b(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	432	tt221=tt221+x(4,i1,i2,j)*a(l,3)+x(8,i1,i2,j)*b(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	433	tt112=tt112+x(5,i1,i2,j)*e(l,3)+x(1,i1,i2,j)*c(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	434	tt212=tt212+x(6,i1,i2,j)*e(l,3)+x(2,i1,i2,j)*c(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	435	tt122=tt122+x(7,i1,i2,j)*e(l,3)+x(3,i1,i2,j)*c(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	436	tt222=tt222+x(8,i1,i2,j)*e(l,3)+x(4,i1,i2,j)*c(l,3)
3,5,7		984,939,969	25,25,24	366,372,367	437	enddo
					438	y(1,i1,i2,i3)=y(1,i1,i2,i3)+tt111
					439	y(2,i1,i2,i3)=y(2,i1,i2,i3)+tt211
					440	y(3,i1,i2,i3)=y(3,i1,i2,i3)+tt121
					441	y(4,i1,i2,i3)=y(4,i1,i2,i3)+tt221
					442	y(5,i1,i2,i3)=y(5,i1,i2,i3)+tt112
					443	y(6,i1,i2,i3)=y(6,i1,i2,i3)+tt212
					444	y(7,i1,i2,i3)=y(7,i1,i2,i3)+tt122
					445	y(8,i1,i2,i3)=y(8,i1,i2,i3)+tt222
					446	enddo
					447	enddo
					448	enddo

(b) Correlation between source code and performance metrics.

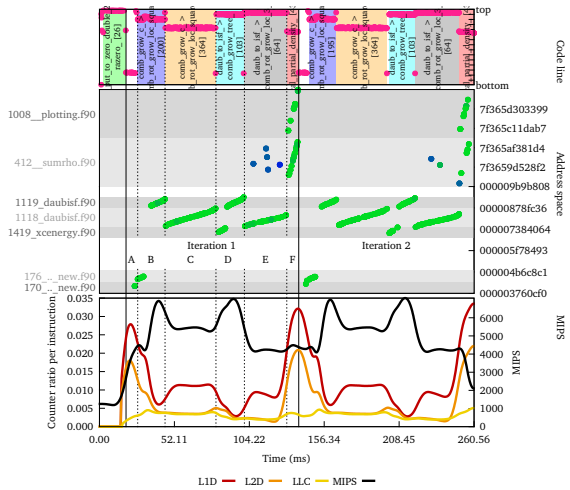


(c) Detailed performance progression of Cluster 1.

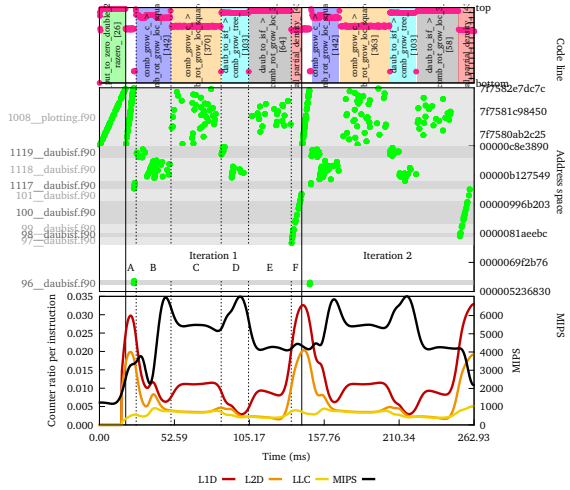


(d) Detailed performance progression of Cluster 1 after modifying the source code.

Figure 5.11
Analysis of BigDFT.



(a) Load references.



(b) Store references.

Figure 5.12
Source code, performance and memory references analysis for the BigDFT application.

expose better spatial locality than the store references and that phase E shows a random access behavior in the load references and these references take more time to be served. Also, phase A traverses completely the array allocated in `plotting.f90` (line 1,008) to store values on it and that happens immediately after executing the `razero` routine (depicted in green) which may be redundant because there are not so many in between.

This report also shows some insight into the chances of making this region parallel using a task-based programming model. For instance, phases B and D store data in the data allocated in `daubis.f90` (line 1,118) and these data are used in phases C and E, begetting true (RAW) dependencies between these pair of phases. Also, phases B and D load and store data from the region allocated in `daubis.f90` (line 1,119) causing true (RAW) and output (WAW) dependencies between these phases. Finally, phase F mainly depends on the data located by `plotting.f90` (line 1,008) which is written by phases C and E. Due to the described dependencies, only phases A and B might safely run in parallel.

5.1.5 GTC

GTC [129] is a parallel, particle-in-cell code for turbulence simulation in support of the burning plasma experiment[10]. The application was monitored on *MareNostrum3* using a 20 Hz sampling frequency when running on 256 MPI processes. Figure 5.13a illustrates the computing regions grouped by their performance metrics. Note that the computing regions labeled as Cluster 1 account for a 28% of the total time. The region experiments some variability with respect to their CPI (ranging from 2.2 to 2.35) but the number of instructions executed is almost constant.

The results of the folding when applied to this region, shown in Figure 5.13c, revealed the existence of four different phases in terms of MIPS. Phase 2 is the longest and takes approximately 61 ms to execute at 800 MIPS, which is 6% of the peak core performance. It may be observed that there is a sudden increase on the miss rates at all cache levels when the application starts this phase. For instance, the ratio between the MIPS and the LLC miss rates points out that one out of twenty instructions (5%) misses at LLC.

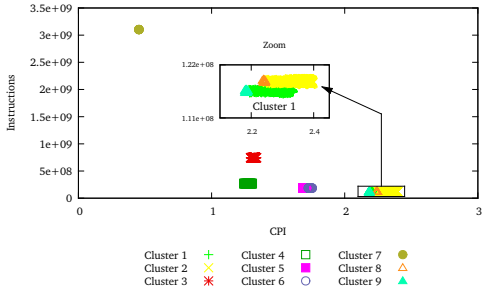
The performance results are complemented by correlating the source code through CUBE visualizer in a screen-shot depicted in Figure 5.13b displaying a partial view of the loop responsible for the performance observed in Phase 2. This loop ranges from lines 131 to 173 within file `pushe.F90` in the routine `pushe`. The Figure indicates that lines 142, 158 and, 159 are the most observed. These lines contain statements that access to the `gradphi` matrix through indirections (`jtgc0` for line 142 and `jtgc1` for lines 158 and 159). The compiler reported that the loop could not take advantage of vector instructions because the subscript was too complex in several of the statements of the loop. Still, there was an opportunity to reduce the cache misses in this region of code by prefetching data from memory before its use (through the `mm_prefetch` instruction). The code was modified to include two prefetch instructions to bring data into ache data referenced by `gradphi` in the beginning of the aforesaid loop two prefetch instructions.

Figure 5.13d illustrates the results of the execution of the tuned application using the same time-scale as in the original Cluster 1. The results of this new analysis show the addition of the prefetch instructions resulted in a reduction of the loop duration by 18% (reduced from 61 to 50 ms). The improvement resulted from the increase in the instruction rate to 955 MIPS, though the LLC miss ratio over instructions did not change substantially. As a result of implementing these changes, the duration of the region was reduced by 11.7% compared to the original execution and the overall execution took 6.2% less time to finish.

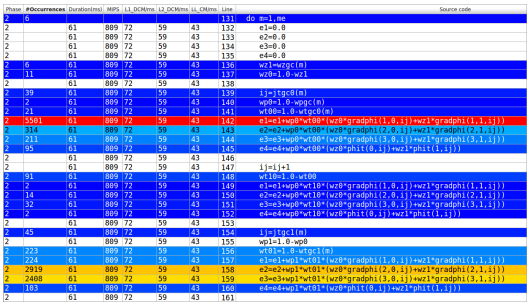
5.1.6 Arts_CF

Arts_CF implements a variable density, conservative and arbitrarily high order finite difference method to simulate flows in complex geometries with cylindrical or cartesian non-uniform meshes [46]. Although the application was provided by the user in binary form in conjunction with an input, the instrumentation package allows the the necessary information to be captured for its use by the framework. This application ran using 512 processes in *MareNostrum3* and the

[10] http://www.iter.org

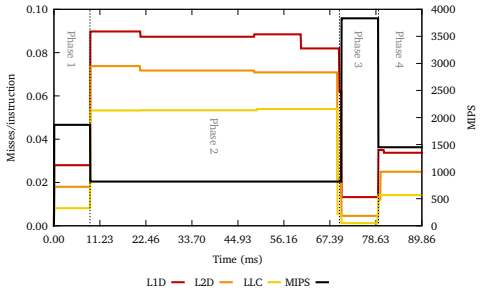


(a) Clustering results.

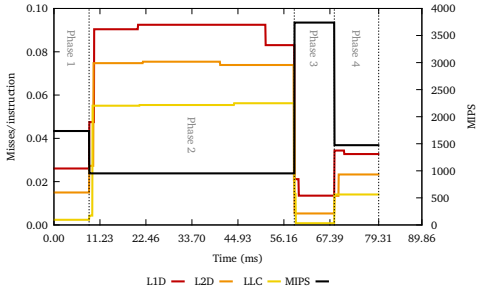


Phase	#Occurrences	Duration(ms)	MIPS	L1_DCM/ms	L2_DCM/ms	LL_CM/ms	Line	Source code
2	6						131	do m=1,me
2		61	809	72	59	43	132	e1=0.0
2		61	809	72	59	43	133	e2=0.0
2		61	809	72	59	43	134	e3=0.0
2		61	809	72	59	43	135	e4=0.0
2	6	61	809	72	59	43	136	wz1=wzgc(m)
2	11	61	809	72	59	43	137	wz0=1.0-wz1
2		61	809	72	59	43	138	
2	39	61	809	72	59	43	139	ij=jtgc0(m)
2	2	61	809	72	59	43	140	wp0=1.0-wpgc(m)
2	21	61	809	72	59	43	141	wt00=1.0-wtgc0(m)
2	5501	61	809	72	59	43	142	e1=e1+wp0*wt00*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
2	314	61	809	72	59	43	143	e2=e2+wp0*wt00*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
2	211	61	809	72	59	43	144	e3=e3+wp0*wt00*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
2	95	61	809	72	59	43	145	e4=e4+wp0*wt00*(wz0*phit(0,ij)+wz1*phit(1,ij))
2		61	809	72	59	43	146	
2		61	809	72	59	43	147	ij=ij+1
2	91	61	809	72	59	43	148	wt10=1.0-wt00
2	2	61	809	72	59	43	149	e1=e1+wp0*wt10*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
2	14	61	809	72	59	43	150	e2=e2+wp0*wt10*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
2	32	61	809	72	59	43	151	e3=e3+wp0*wt10*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
2	2	61	809	72	59	43	152	e4=e4+wp0*wt10*(wz0*phit(0,ij)+wz1*phit(1,ij))
2		61	809	72	59	43	153	
2	45	61	809	72	59	43	154	ij=jtgc1(m)
2		61	809	72	59	43	155	wp1=1.0-wp0
2	223	61	809	72	59	43	156	wt01=1.0-wtgc1(m)
2	224	61	809	72	59	43	157	e1=e1+wp1*wt01*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
2	2919	61	809	72	59	43	158	e2=e2+wp1*wt01*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
2	2408	61	809	72	59	43	159	e3=e3+wp1*wt01*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
2	103	61	809	72	59	43	160	e4=e4+wp1*wt01*(wz0*phit(0,ij)+wz1*phit(1,ij))
2		61	809	72	59	43	161	

(b) Correlation between source code and performance metrics.



(c) Detailed performance progression of Cluster 1.



(d) Detailed performance progression of Cluster 1 after modifying the source code.

Figure 5.13
Analysis of GTC.

clustering tool detected up to 12 computing regions, as depicted in Figure 5.14a. Several of the identified clusters elongate through the X-axis indicating variability in terms of CPI.

The most time-consuming region (Cluster 1), which characterizes approximately 19.5% of the total execution time, ran at an average 7,500 MIPS. The folding results of this region (not shown) indicate a long phase running at 8,000 MIPS (more than 60% of the processor peak performance). Although there is a gap to reach the processor's peak performance, practice shows that the achieved performance is good enough and it is unlikely to offer optimization opportunities by applying simple code transformations. So this analysis focus on the two following most time-consuming computing regions (i.e. Clusters 2 and 3).

For instance, Cluster 3 represents up to 17.6% of the application execution time. Figure 5.14b shows the correlation of the call-stack information and several counter ratios (including cache misses at several levels, branch mispredictions and proportion of floating-point instructions). The Figure points out that the computing region executes three phases during its execution and the code line profile indicates that a small number of lines are responsible for the performance behavior observed. The first phase, colored in red, is mainly devoted to line 667 within the routine `scalar_weno5_coeff`, lasts approximately 30 ms and runs at 2,600 MIPS (less than 20% of the peak performance). The second phase, depicted in green, runs for 26 ms at 7,300 MIPS and correlates with line 722 in the routine `scalar_weno5_residual` and then code progresses to the line 773 where the performance increases to 8,900 MIPS. The last phase, colored in yellow, executes for 3 ms while running at 4,800 MIPS and it is associated with the routine `update_div`.

The analysis of the performance metrics in the first phase shows that neither the cache hierarchy nor the branch predictor is limiting the performance. Less than 2% of the instructions miss at L1D cache and less than 1% of the instructions fail on the branch predictor. However, the results point out that most of the instructions are floating-point instructions. An analysis of the ratio of stalled cycles (shown in Figure 5.14c) points out that up to 75% of the clock cycles are stalled, from which 40% are related to scarce re-order buffer entries. This phase is mostly correlated to an innermost sentence within a four-nested loop and this sentence included a floating-point division on which the divisor is invariant in the inner-most loop. Since divisions take longer to complete than multiplications, the code was changed to multiply by the inverse of the divisor instead.

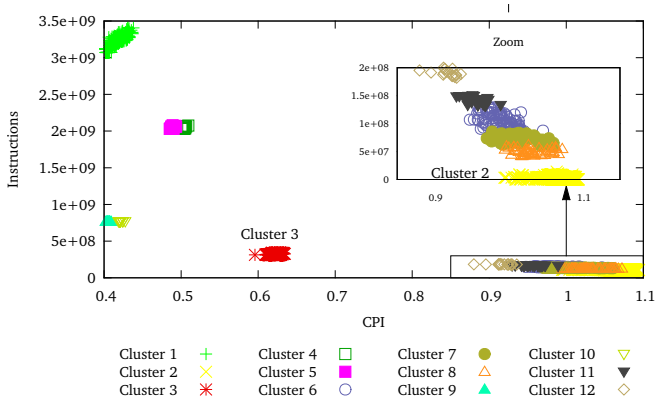
A similar performance analysis (not shown) was applied to Cluster 2, representing an additional 18.3% of the computation time. Approximately 80% of this computing region ran uniformly at 2,600 MIPS and the source code pointed to the inner-most statement of the loops in lines 539, 545, 600 and 606 within the routine `scalar_weno_coeff`. Since the performance symptoms were similar to those detailed in the first phase of the previous computing region, similar code transformation were applied to the loops pointed in this region.

After implementing these changes and re-analyzing the application performance, the duration of Clusters 2 and 3 decreased by 28.3% and 17.4%, respectively. With respect to the performance achieved in Cluster 3 (depicted in Figures 5.14d and 5.14e), results show that the instruction rate in the red phase increased to 4,200 MIPS approximately and the stalled cycles decreased to less than 60%. In overall, the application execution time was reduced by 9.0%.

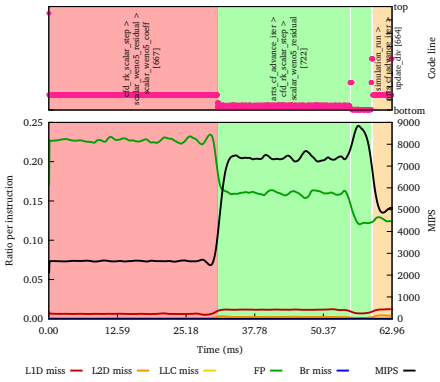
5.1.7 Nemo

Nemo [136] is an ocean model that includes several components besides the ocean circulation, including sea-ice and bio-geochemistry. The application was executed on 128 processors of *MareNostrum3* and presented nine computing regions (as shown in Figure 5.15a), from which the most time-consuming cluster took approximately 14% of the total execution time. Each computing region instance belonging to Cluster 1 ran at approximately 1.5 CPI and executed 3.3×10^8 instructions.

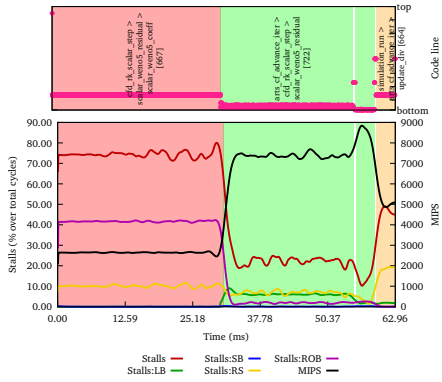
Figure 5.15b shows the combined performance and source code reference results generated by the mechanism for this computation region. The results indicate that most of the time is invested in the routines `tra_ldf_iso` and `tra_zdf_imp` (colored in green and blue) being lines 212 and 204 are the most observed, respectively. The code line profile displays a pattern of two iterations on both routines. With respect to `tra_ldf_iso`, it mostly executes a loop within lines 173-313. This loop contains a nested loop (in lines 207-253) in which 7% of the instructions miss



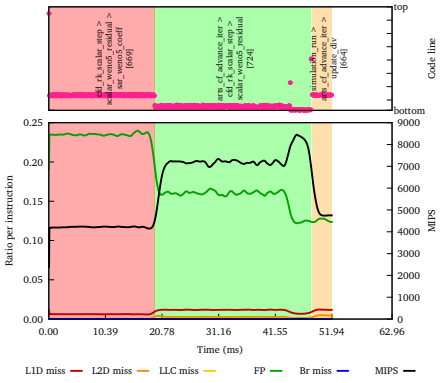
(a) Clustering results.



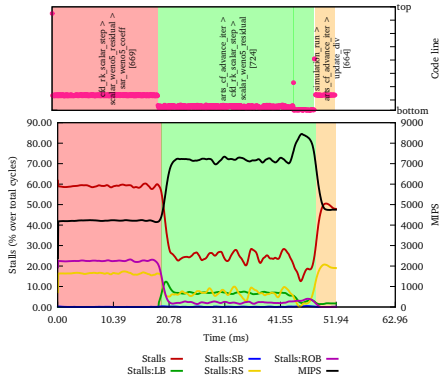
(b) Detailed performance progression of Cluster 3.



(c) Detailed performance progression of Cluster 3 (stalls).

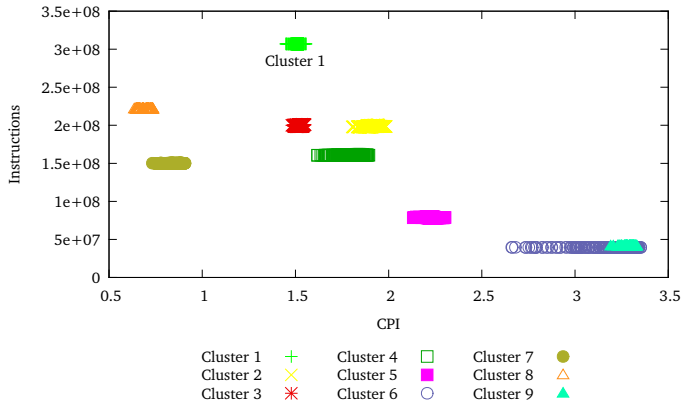


(d) Detailed performance progression of Cluster 3 after modifying the source code.

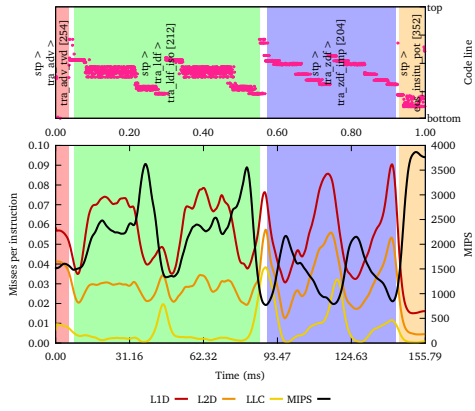


(e) Detailed performance progression of Cluster 3 (stalls) after modifying the source code.

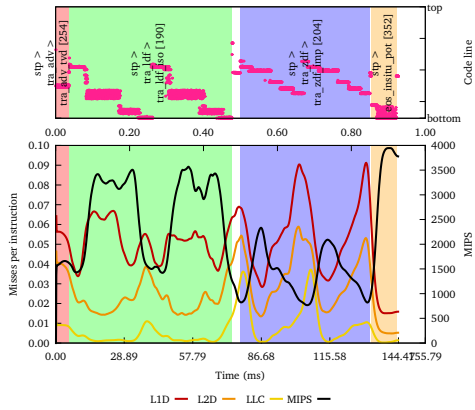
Figure 5.14
Analysis of Arts_CF.



(a) Clustering results.



(b) Detailed performance progression of Cluster 1.



(c) Detailed performance progression of Cluster 1 after the modifications.

Figure 5.15
Analysis of Nemo.

at L1D cache due to the access to one 4D matrix, eleven 3D matrices and several 2D matrices. With respect to `tra_zdf_imp`, the source code line profile indicates that the computation invests most of its time in the loop delimited by lines 161-280. Within this loop there is a nested loop in lines 271-277 that achieves the worst performance (less than 1,000 MIPS) because of the high L1D cache miss ratio (8%). The code within this nested updates a 4D matrix where each element is calculated by accessing to four matrices and one of these matrices is accessed in two different planes.

An analysis of the source code of the loops in `tra_ldf_iso` shows that the loops in lines 245-251 and 303-311 perform a very similar task: they calculate the divergence of fluxes. To perform such calculations they access to the same data structures, perform similar operations and update the same 4D matrix. Since these structures and matrix remain untouched from one loop to the next, the two loops were fused to reduce the number of control instructions executed and to improve the temporal locality of the data. The loop in lines 207-253 calculates two 2D matrices (`zdkt` and `zdklt`) before entering into the nested loops but the nested loops only access to the elements (i, j), $(i+1, j)$ and $(i, j+1)$. Therefore, to reduce the pressure on the cache, the modified code calculated these three elements and stored the results in scalar variables instead of calculating the entire set of elements. Figure 5.15c shows the results of the framework after analyzing the modified version of the source code. The results indicate that the region took 7.2% less time to execute, increasing its average instruction rate from 1,950 to 2,200 MIPS.

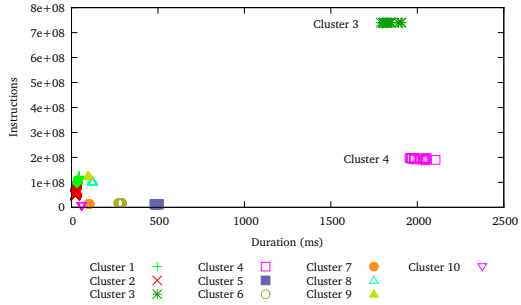
5.1.8 PEPC

PEPC [84] is a parallel MPI-based tree-code for rapid computation of long-range Coulomb forces in N-body particle systems based on the original Barnes-Hut algorithm. The application was executed in *Tamariu* using 16 processes and the clustering results for the execution lead to the plot depicted in Figure 5.16a[11]. The folding results (not shown) for the two leading compute regions, i.e. Cluster 1 and 2, execute at high and uniform performance (3,100 and 2,500 MIPS) when traversing the tree following indirections stored in a linked-list. Because of the code they represent, it is difficult to improve the code without changing the application data-structures, which is beyond the scope of this thesis. Still, an analysis of the time-line (shown in Figure 5.16b) indicates that Clusters 3 and 4 approximately represent a combined 20% of the execution time and each invocation last 1,833 and 2,014 ms, respectively. In addition, the time-line reports that Cluster 4 is executed immediately after Cluster 3 and there were multiple invocations to Cluster 1 and 2 before executing Cluster 4 again.

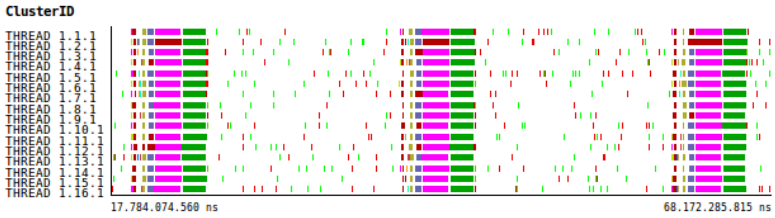
Figures 5.16c and 5.16d show the folding results for the compute regions identified as Cluster 3 and 4, respectively. The black line refers to the MIPS achieved and it is referenced on the right Y-axis, whereas the L2 data-cache and TLB misses use the left Y-axis. According to the instruction rate, it is possible to divide each of the compute regions into four phases. The plots include manually added labels to represent phases in Cluster 3 (named from A to D) and phases in Cluster 4 (named from E to H). The MIPS rate shown in the computation phases of these regions depict several phases where the instruction rates were very low. More precisely, phases A, C and F did not go beyond 100 MIPS, which was less than 1% of the processor peak MIPS rate.

A detailed analysis of the Cluster 3 shows that, in phase A, five out of every 100 instructions miss in L2. The source code references report that two lines in the source code were the most time-consuming of this phase. These lines executed the Fortran90 intrinsic `pack` operation that packs an array to a vector with the control of a mask. Phase C that runs at 80 MIPS shows high L2 and TLB miss ratios per instruction which indicate that they were the limiting factor of this phase. The code associated with this phase involves two loops. The first loop resets several large structured arrays accessed through hash indices. On the other hand, the second loop contains 27 multiplications, nine exponential operations, three divisions and one square root with read-after-write dependencies among them. Note the ratio of TLB misses in this phase. Such an amount of misses may occur if the address space of the hash function is not large enough to store the complete working set. In contrast, L2 cache does not suffer from this issue

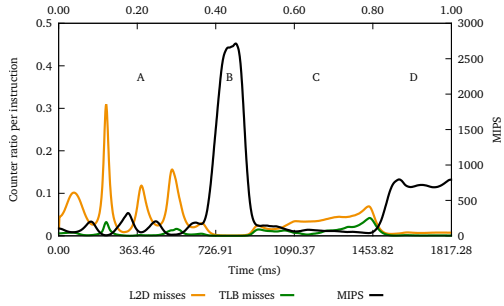
[11] The computing regions in this experiment are classified according to their number of executed instructions as well as their duration due to the limitations of the measurement system.



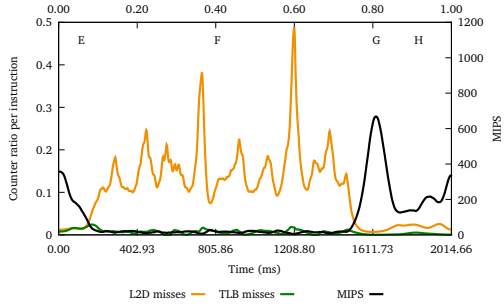
(a) Clustering results.



(b) Time-line results.

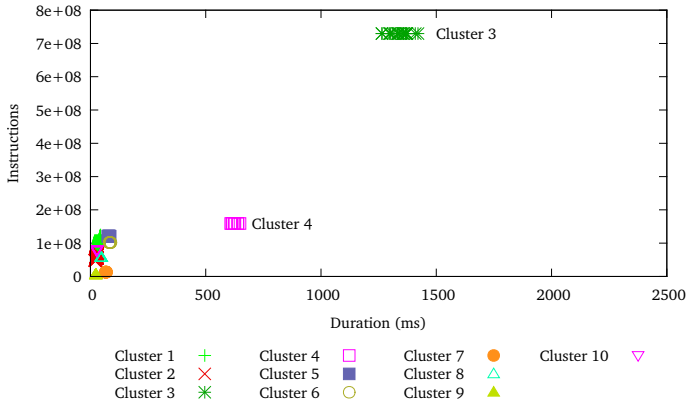


(c) Detailed performance progression of Cluster 3.

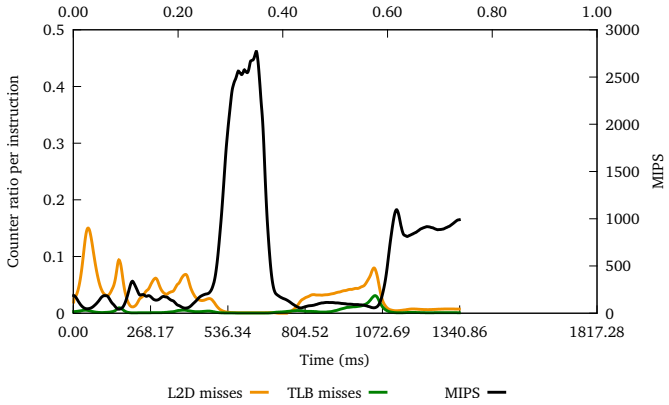


(d) Detailed performance progression of Cluster 4.

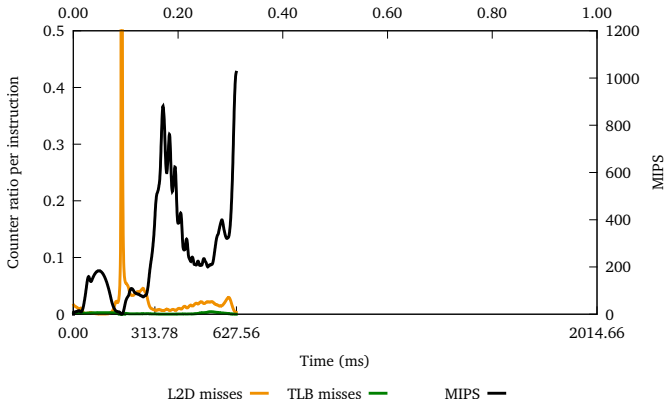
Figure 5.16
Analysis of PEPC.



(a) Clustering results.



(b) Detailed performance progression of Cluster 3.



(c) Detailed performance progression of Cluster 4.

Figure 5.17
Analysis of the modified version of PEPC.

Table 5.10
Side-by-side comparison of both original and optimized PEPC files.

Original tree_allocate.f90

```
120
121 htable%node = 0
122 htable%key = 0
123 htable%link = -1
124 htable%leaves = 0
125 htable%childcode = 0
126
```

Optimized tree_allocate.f90

```
120 do i = 1,maxaddress    ! Loop
121   htable(i)%node = 0
122   htable(i)%key = 0
123   htable(i)%link = -1
124   htable(i)%leaves = 0
125   htable(i)%childcode = 0
126 enddo
```

as its size and associativity are larger than the TLB (TLB is 4-way with 32 entries[10], whereas L2 cache is 12-way with 49152 entries[10]). Improving the performance on this phase would include using large memory pages or building a more efficient hash function. Finally, phase D is mostly devoted to a loop that accumulates multi-pole moments by traversing a tree structure. This loop contains a nested loop that operates for every node child and executes 33 multiplications and three exponential operations. There is a large pressure on the register-file because of the large amount of operations and the small number of registers in the Intel 32-bit architecture (eight). It is therefore unlikely that loop unrolling techniques offer additional performance since they require additional registers. However, the nested loop may benefit from vectorization due to the lesser number of control instructions executed and the availability of additional registers.

With respect to Cluster 4, its MIPS rate is extremely low given that phase F runs at approximately 16 MIPS (approximately 0.16% of the node-level peak performance). This phase commits fewer than 10% of instructions of the region in 60% of the computation time. The analysis of the correlated code shows that a small number of lines are responsible for such a bad performance, mostly in lines 120-124 and 58-62 in the files `tree_allocate.f90` and `tree_build.f90`, respectively. This code initializes structures using intrinsic Fortran90 vector instructions over large arrays pointing to large structures which result in large stride accesses to memory. A suggested change to improve the memory hierarchy accesses includes rewriting these vector instructions using a single loop to initialize the required fields of every array position as shown in Table 5.10.

After implementing the modifications for phase B of Cluster 4, the analysis of the application reports the results shown in Figure 5.17. The plots in this Figure show different aftereffects of the application performance behavior. The clustering results (depicted in Subfigure 5.17a) shows that Cluster 4 moved to the left when compared with the original results, indicating that the region required less time to compute. The internal evolution of Cluster 4, as illustrated in Subfigure 5.17c, shows that the duration of the region decreased to from 2,014 to 627 ms. Comparing these results with the original, such improvements translate into a reduction of the 68% of the execution time and an increase on the average instruction rate from 96 to 253 MIPS. Even though the results show a significant peak on L2D miss ratio, the improvement came from the reduced number of TLB misses and executed instructions in this computation regions by 78% and 18%, respectively. Another interesting consequence is that Cluster 3 also moved a bit to the left on these results, also meaning that the region required less time to execute. In general terms, the execution time of this region decreased by 26% and its instruction rate increased from 403 to 544 MIPS. Since the code associated with Cluster 3 had not changed, the only reason for its improved performance is that it was executed immediately after Cluster 4. The modifications to Cluster 4 improved the memory utilization at both cache and TLB structures, the execution of Cluster 3 was likely to execute faster in such a cleaner state thanks to better cache use. The folding results show a reduction in the number of L2D and TLB misses by 10% and 34%, respectively, as well as a reduced variability across time. Overall, the running time of the application was reduced by 14%.

[10] As reported by the `papi_mem_info` utility on the execution machine.

5.1.9 Nest

Nest [81] is a simulator used in The Human Brain Project[13] for spiking neural network models that focus on the dynamics, size and structure of neural systems rather than the morphology of individual neurons. NEST was executed on two Intel Xeon Phi processors using the coprocessor-only model; so all processes (120) were run on the coprocessors. Figure 5.18a illustrates that Cluster 1 suffered variability in both instruction and duration per instruction[1]. On average, each invocation to a computing region that belongs to Cluster 1 executed approximately 5×10^6 instructions where instructions took approximately 3.3 ns to execute. An analysis of the trace-file shows that Cluster 1 represented 33% of the total time.

Figure 5.18c shows the folding results for Cluster 1. These results clearly indicate that Cluster 1 exhibits four phases. Phase 2 was the longest and took approximately 50% of the duration running at 280 MIPS (about 20% of the core peak performance). The performance counters show that the application was memory bound, more precisely due to L1D and TLB misses because the processor missed in the L1D and TLB every 30 and 40 instructions respectively, during this phase.

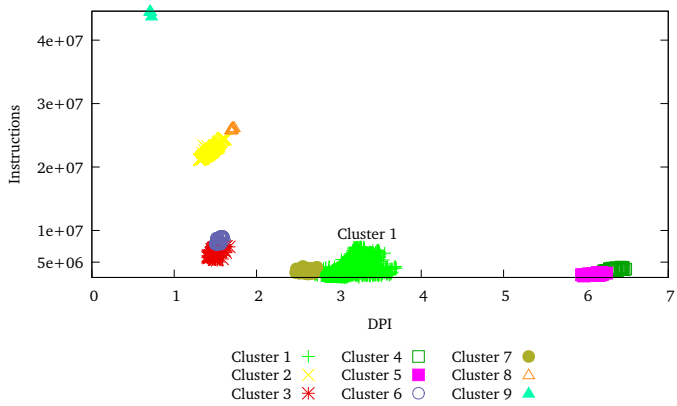
Figure 5.18b displays the attribution of the performance metrics to the application source code for Phase 2. The code correlated with this phase is a single-line in-lined method named `add_value` found in `ring_buffer.h`. This method accesses the `buffer_` attribute through indirections on the `get_index` method, so leaving little room for improvements at this level. The CUBE call-tree (not shown) reports that this method is called by a method named `Connection-Manager::send` after a chain of method invocations. There is a loop in lines 652-653 within the file `connection_manager.cpp` that traverses a list of objects and ends up calling `add_value`. According to the compiler report, one of the routines invoked within this loop was promoted to ignore the inline attribute and the vectorization was also disabled due to the complexity of the loop. While this architecture enjoys the maximum performance from the vector codes, the application would probably benefit from improving the memory referencing, in addition to, using vectorization. It may be though that on way of addressing this issue would be by prefetching the data contained in `buffer_`, for example. However, since each object instance exists in a different memory region, their respective buffers are unlikely to be consecutive; therefore, accessing them may not expose spatial locality. In other words, the application organizes the data in array of structs (typically known as AoS) that contain the `buffer_` member. An approach to increase the spatial locality would involve changing the organization of the data as a struct of arrays (commonly referred as SoA). By using SoA, the buffers of every object instance are allocated together in consecutive regions of memory and so increase the effectiveness of both L1D cache and TLB. However, implementing this modification is far beyond the scope of this work.

5.1.10 CESM

CESM [98] is a fully-coupled, global climate model that provides state-of-the-art computer simulations of the Earth's past, present and future climate states. The application was executed by an external user in the *Yellowstone* system using 570 processes. Figure 5.19a shows the identified regions for this execution and the scatter-plot indicates that the three identified regions show variations in terms of CPI. Cluster 1 also experienced variations of CPI with the result that the more instructions executed, the higher the CPI. According to the obtained results, the most time-consuming region of this application represented approximately 12.7% of the execution time.

Figure 5.19b shows the progression of the source code references and the performance within this region. The results indicate that the region calls several subroutines and for the sake of simplifying the results, the Figure has been manually adapted to display labels (from A to D) in the short routines. With respect to the source code, the region mainly exhibits a pattern that repeats three times in which periods include calls to the routine `tphysbc`. This routine invokes several procedures of differing duration. Among these procedures, there is `radiation_tend` (shown as B) which is sufficiently long and contains an invocation to the routine `rrtmg_sw` (shown as C). The second and third invocations to `tphysbc` present a slightly different source code profile from

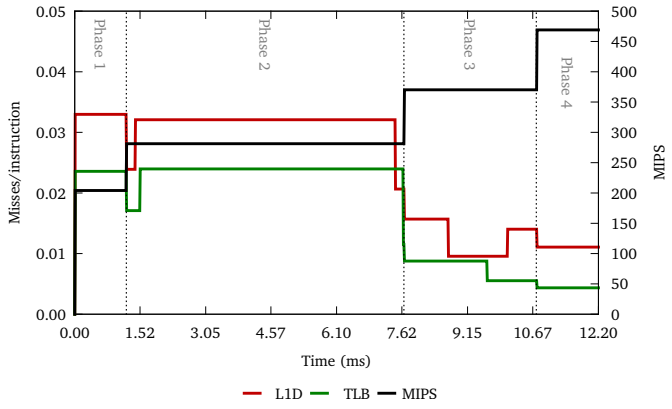
[13] https://www.humanbrainproject.eu



(a) Clustering results.

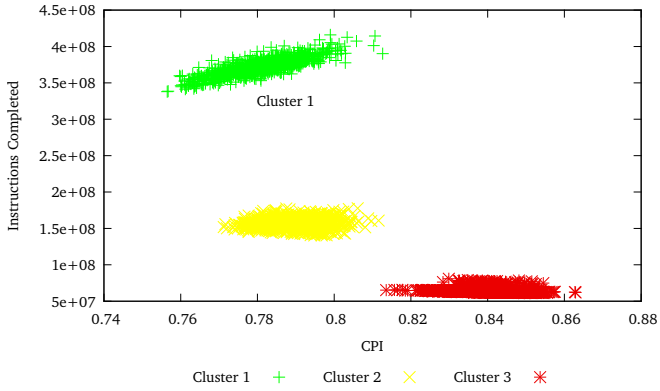
Phase	#Occurrences	Duration(ms)	**MIPS**	L1_DCM/ms	TLB_DM/ms	Line	Source code
						135	
						136	inline
						137	void RingBuffer::add value(const long t offs, const double t
						138	{
2	1419	6	281	9	7	139	buffer [get index (offs)] += v;
						140	}
						141	
						142	inline
						143	void RingBuffer::set value(const long t offs, const double t
						144	{
						145	buffer [get index (offs)] = v;
						146	}
						147	
						148	inline
						149	double RingBuffer::get value(const long t offs)
						150	{
3	34	3	370	4	3	151	assert(0 <= offs && (size t)offs < buffer .size());
3	39	3	370	4	3	152	assert((delay)offs < Scheduler::get min delay());
						153	
						154	// offs == 0 is beginning of slice, but we have to
						155	// take modulo into account when indexing
						156	long t idx = get index (offs);
3	592	3	370	4	3	157	double t val = buffer [idx];
						158	buffer [idx] = 0.0; // clear buffer after reading
						159	return val;
						160	}
						161	

(b) Correlation between source code and performance metrics.

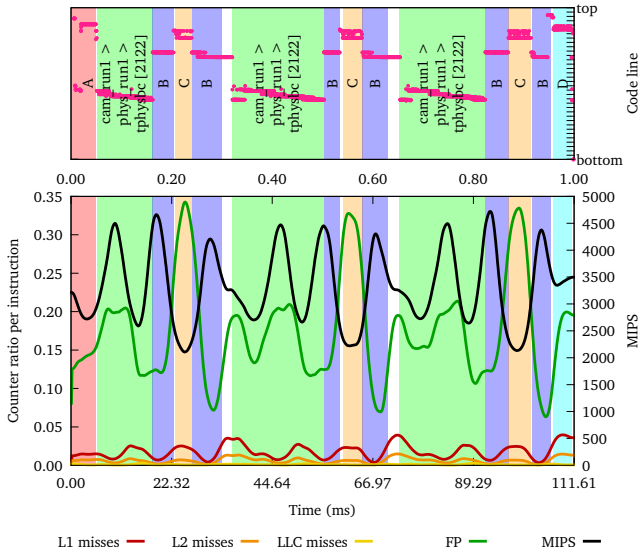


(c) Detailed performance progression of Cluster 1.

Figure 5.18
Analysis of Nest.



(a) Clustering results.



(b) Detailed performance progression of Cluster 1.

Figure 5.19
Analysis of CESM.

Table 5.11
Benchmarks used for the performance and power experiments and the location of the begin and end points to delimit the iterative part of the application.

Benchmark suite	Name	Time-step code region
SPEC CPU 2006	434.zeusmp	`src/zeusmp.F` 675-709
	435.gromacs	`src/md.c` 413-820
	436.cactusADM	`src/PUGH/Evolve.c` 96-147
	437.leslie3d	`src/tml.f` 330-435
	444.namd	`src/spec_namd.C` 184-225
	465.tonto	`src/mol.F90` 13617-13629
	470.lbm	`src/main.c` 44-58
	481.wrf	`src/module_integrate.F90` 274-288
NPB 3.3	bt.B	`BT/adi.f` 8-20
	ft.B	`FT/appft.f` 62-72
	is.C	`IS/is.c` 446-641
	lu.B	`LU/ssor.f` 102-232
	mg.B	`MG/mg.f` 255-264
Stream	stream	`stream.c` 220-262
Lulesh	lulesh	`full/lulesh.cc` 2,893-2,904

the first since they include the execution of the loop in line 644 in the beginning. Regarding the overall performance, the results point out that less than 2% of the instructions missed at L2 and that more than 16% of the instructions were floating-point instructions. The lowest MIPS observed, which was also the most intensive floating-point phase along the region, occurred in `rrtmg_sw`. This routine invoked a set of routines (named `taumol*`) sequentially, where each contained nested loops with many (14, in most of the cases) floating-point multiplications to generate a single value stored in a 2D matrix. This large quantity of floating-point activity, in conjunction to the dependencies generating a single value, resulted in a high value of cycles stalled (not shown, but approximately 70%). Also, the MIPS rate decreased within the `tphysbc` routine due to similar reasons when invoking `get_snow_optics_sw` and `get_ice_optics_sw` resulting in half of the processor cycles stalled. In this case none of these multiplications could be replaced with cheaper operations (as in Arts_CF), so a major effort was required for improving these portions.

5.2 Simultaneous performance & power analysis

In addition to the previous tests, there has been further experimentation with respect to simultaneous analyses of performance and power metrics. The experiments reported within this section were executed in *Altamira*, which runs at a nominal frequency of 2.60 GHz with a Thermal Design Power (TDP) of 115 Watts. This system runs Linux kernel 2.6.32 and allows changing the processor frequency from 1.2 up to 2.6 GHz in 0.2 GHz steps. While the processor can increase its frequency up to 3.3 GHz using the Intel Turbo Boost capability, this functionality was manually disabled in order to perform all the tests at a uniform frequency.

The analyses focus on two set of applications. The first set involves serial benchmarks that were executed sequentially one after another and pinned to the first core of the socket. On the other hand, the second set includes MPI-based parallel applications. The pinning for the parallel applications depended on the execution configuration applied in terms of processes per socket, while ensuring that each core executed only one process at the most. The applications were compiled using the GNU compiler suite version 4.4.6 with `-O3 -g` as compile flags and OpenMPI version 1.6 for the parallel applications.

5.2.1 Serial benchmarks

The selected serial benchmarks include a subset from the SPEC CPU2006 benchmark suite [92], from the NAS benchmark suite version 3.3 [10] as well as Stream [141] and Lulesh [99] benchmarks and are listed in Table 5.11. Since these applications do not employ any parallel programming model, it is not possible to benefit from the framework. It is therefore necessary to instrument the start and end points of the time-stepper routine in order to delimit the synthetic region for the folding mechanism. The Table therefore shows the location within the source where the time-step begins and ends within the application source code.

The results of the folding for all the benchmarks are shown in Figure 5.20 and describe the time-step routine of the benchmarks by combining performance and power metrics. In each plot, the instruction rate is shown in black and referenced on the right Y-axis. On the other hand, the power metrics are plotted on the left Y-axis and colored in blue, purple, red for the DRAM, the core and the total package (socket), respectively. The first observation derived from these plots is that the Stream benchmark is the application that drains more power along its execution. This benchmark dissipates more than 36 Watts, from which 20 and 14 Watts are drained by the package and the memory subsystems, respectively. These plots also outline that the cores consumed essentially the same amount (between 16 and 18 Watts) whatever the performance achieved by the application and irrespective of the application activity. More specifically, benchmarks 437.leslie3d (5.20d), 481.wrf (5.20h) and ft.B (5.20j) experienced the largest range on the instruction rate within the time-stepper region (ranging from 1,000 to 7,500 MIPS, from 3,000 to 9,000 MIPS and from 2,500 to 7,000 MIPS, respectively) with a small variation on the core power consumption. Note too that at the end of lu.B (5.20l) and ft.B (5.20j), the more MIPS achieved, the less power drained. It may also be observed that the power consumption of the DRAM was mostly uncorrelated with the performance in all executions except for 437.leslie3d (5.20d), 481.wrf (5.20h) and is.C (5.20k) where the high peak performance resulted into higher DRAM consumption. With respect to the power consumption, it is interesting to note that sum of the DRAM and core power consumption did not count for the total consumption of the package. The results show that though the processors did not have any integrated GPU on their power plane, there was some uncategorized power consumed that explains the remaining power to reach the reported total. Finally, although the shapes in the package and core results were similar, the results shown in lu.B (5.20l), ft.B (5.20j), 434.zeusmp (5.20a) and 437.leslie3d (5.20d) suggest that the power consumed by the DRAM was under-weighted by the PCU when summing up the package consumption.

5.2.1.1 Application of the DVFS techniques

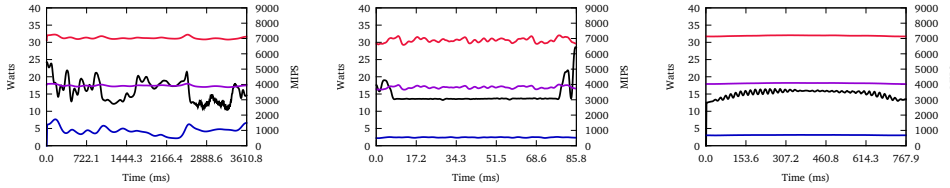
The power dissipated by an electronic circuit using the current CMOS technology is divided into two parts: static and dynamic, the latter being claimed the main source of power consumption. Dynamic power is the rate at which electric energy is transferred by a circuit to commute the transistors of the circuit and it is equal to:

$$P_{dynamic} = \alpha C V^2 f \tag{5.2}$$

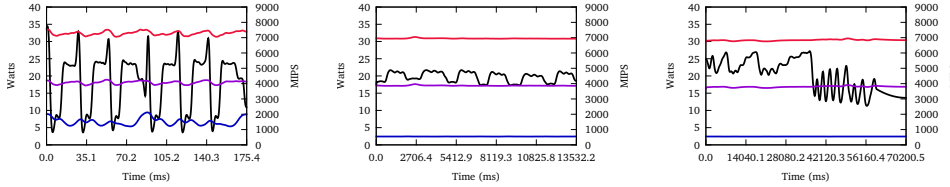
where α is the circuit activity ratio (so, directly related to the application executed), C refers to the chip capacitance, V indicates the operational voltage and f specifies the processor frequency. According to this formula, there are two ways to reduce the energy consumption considering a fixed application activity: by reducing either the processor voltage or the processor frequency. While the former cannot usually be tackled by the user in a production environment because it may either require privileges or physical access to the node, the latter is still applicable. In fact, since the voltage required to operate the CPU at given frequency is, roughly speaking, proportional to the frequency, the relationship between frequency and power consumption is sometimes rewritten as:

$$P_{dynamic} = \alpha C f^3 \tag{5.3}$$

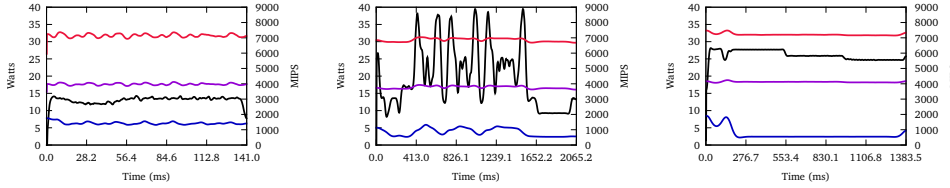
which suggests a better alternative for reducing the dissipated power without accessing the system physically by altering the processor frequency, though changing it alters the time-to-solution.



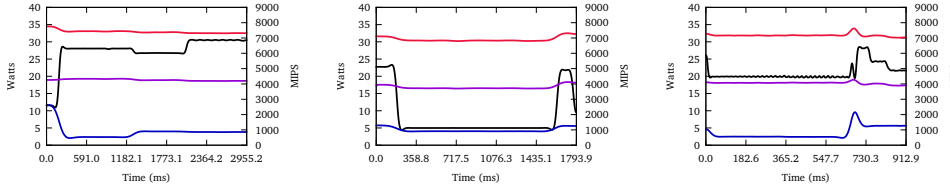
(a) 434.zeusmp **(b)** 435.gromacs **(c)** 436.cactusADM



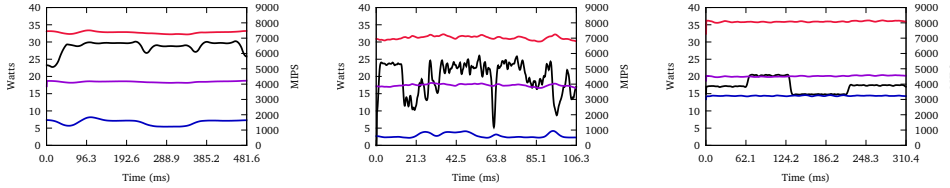
(d) 437.leslie3d **(e)** 444.namd **(f)** 465.tonto



(g) 470.lbm **(h)** 481.wrf **(i)** bt.B

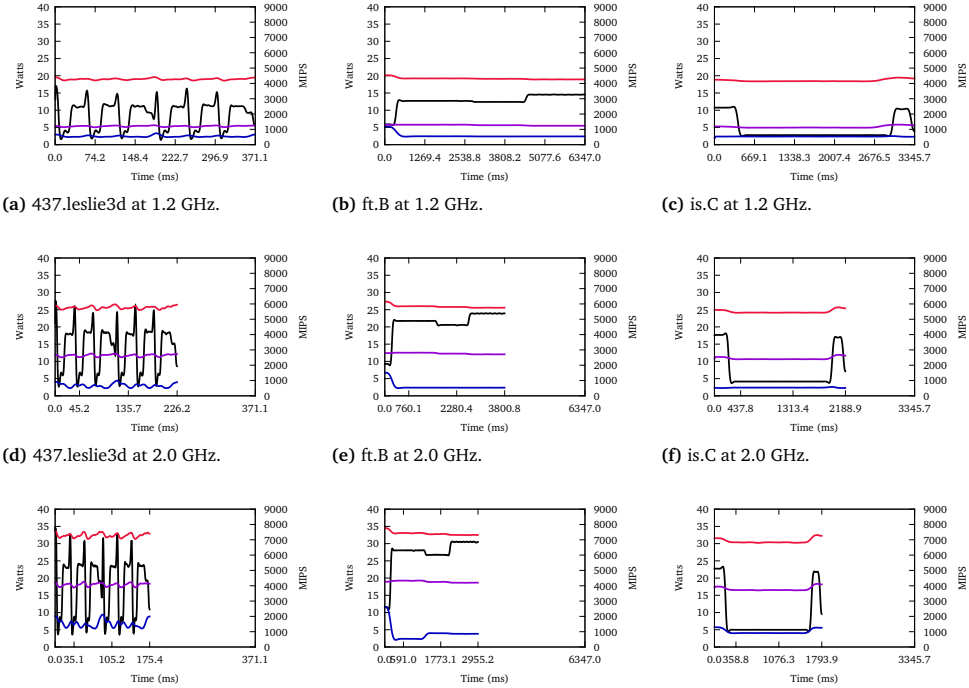


(j) ft.B **(k)** is.C **(l)** lu.B



(m) mg.B **(n)** Lulesh **(o)** Stream

Figure 5.20
Comparison of the performance and power consumption on the main iteration of several benchmarks running at 2.6 GHz. The black line refers to the instruction rate. The blue, purple and red refer to the power drained by the DRAM, the core and the package, respectively.



(a) 437.leslie3d at 1.2 GHz. **(b)** ft.B at 1.2 GHz. **(c)** is.C at 1.2 GHz.

(d) 437.leslie3d at 2.0 GHz. **(e)** ft.B at 2.0 GHz. **(f)** is.C at 2.0 GHz.

(g) 437.leslie3d at 2.6 GHz. **(h)** ft.B at 2.6 GHz. **(i)** is.C at 2.6 GHz.

Figure 5.21
Performance and power progression of three benchmarks when run at three core frequencies.

Current operating systems increase or reduce the processor frequency (f) depending on the load of the system using a technique called Dynamic Voltage and Frequency Scaling (DVFS). In order to reduce energy consumption, the operating system lowers the frequency when the system is idle and reestablishes the processor frequency when the system becomes loaded. Previous researches, such as [64] and [183], have already used this technique to reduce the energy consumption on parallel applications that present load unbalance by decreasing the processor frequency on the processors with less work.

In order to explore the impact on the performance and the power consumption of this technique, a subset of the previous benchmarks was run using three core frequencies (1.2, 2.0 and 2.6 GHz). The plots in Figure 5.21 show the results for these executions. Again, the black line refers to the instantaneous MIPS rate and is plotted on the right Y-axis, whereas the energy consumption is shown red for package, blue for DRAM and purple for core and use the left y-axis. The first thing to notice is that neither the power nor the performance shape of the benchmarks changed when modifying the core frequency, except for the amplitude of the signal. For instance, the highest and lowest peaks in the is.C benchmark (depicted in Figures 5.21c, 5.21f and 5.21i) range from 600-2,400, 900-4,000 and 1,100-5,000 MIPS and, 4.9-5.8, 10.6-11.9 and 16.4-18.3 Watts in terms of performance and power drained by the core, respectively. These plots outline that the higher the core frequency, the higher amplitude exists in both performance and power metrics. More room for energy and performance improvements can therefore be reached at higher frequencies, as it would be expected from Equation 5.2. As noted earlier, although the power consumption may be directly related to the processor performance, as it occurs in 437.leslie3d and is.C, sometimes the power consumption is independent of the performance achieved, as observed in ft.B.

The plots also show that the reduction factor observed in the power use when decreasing the

core frequency was higher than the reduction factor of the clock rate. For instance, the cores consume about 18 Watts when running ft.B at 2.6 GHz. According to Equation 5.2 the power consumed running at 1.2 GHz would be approximately 9 Watts, however, the results indicate that power consumption did not reach 6 Watts. The plot shown in Figure 5.22a shows a similar effect when varying the processor frequency. The slope between frequencies 1.2 and 2.0 GHz is smaller than the slope between frequencies 2.0 and 2.6 GHz. These observations, considering that the chip capacitance (C) and the activity rate (α) are constant, lead to the conclusion that the processor lowers its voltage (V) when lowering its frequency (f), according the rewritten Equation 5.3.

Even though the focus of this thesis involves depicting the detailed behavior of computation regions, the recent increase of interest in power related topics, makes worthwhile summarizing the findings of the selected benchmarks. The tabulated results appear in Tables 5.12 and 5.13. The first Table shows the average duration, the core consumption and the whole-socket consumption of the main iteration of the benchmark when run at one of the selected processor frequencies (2.6, 2.0 and 1.2 GHz). The Table shows that the energy consumed by the cores decreases as the frequency decreases, but to a lesser scale. That is, although reducing the processor frequency from 2.6 to 1.2 GHz makes the application run more than two times slower, the energy consumed by the cores does not reduce accordingly because of the static component of the power dissipated. It is also interesting to note that the total energy consumed by the whole package increases as the processor frequency decreases because the energy dissipated is a function of time and power. So the longer the application is run, the more energy it drains. These results agree with the results of a group of experiments carried out by Le Sueur and Heise [124]. The results suggest that to reduce the energy consumed by the system when running an application, it should be run at the maximum processor frequency and halt the processor once the application finishes.

On the other hand, Table 5.13 shows the average number of instructions and the average number of L1D, L2 and LLC cache misses of the main iteration of the benchmark as well as additional performance and power metrics derived from the aforementioned data. The Table presents the average MIPS and the average MIPJ per core ($MIPJ_C$) and per package ($MIPJ_P$) (analogously to MIPS, MIPJ stands for Millions of Instructions per Joule) achieved by each benchmark. The results indicate that considering $MIPJ_C$ as the power efficiency metric, the lowest frequency provides the best results in terms of instructions per Joule achieved. However, the most fruitful frequencies range from 2.0 to 2.6 GHz when recognizing $MIPJ_P$ as the power efficiency metric because using the highest frequencies allows the application to finish earlier. More specifically, four of the slowest benchmarks (434.zeusmp, 470.lbm, is.C and Stream) achieve the best $MIPJ_P$ (114.51, 92.45, 61.26 and 108.69) running at 2.0 GHz, while the rest of the benchmarks show better $MIPJ_P$ when running at 2.6 GHz. These four benchmarks show high L1D, L2 and LLC miss rates, indicating that these applications are memory bounded. An obvious approach to select the most effective frequency in terms of $MIPJ_P$ depends on the ratio between the executed instructions and the LLC misses. In overall, the benefit in terms of $MIPJ_P$ is minimal when moving from 2.0 to 2.6 GHz as depicted in Figure 5.22b.

5.2.2 Mr. Genesis

This application was executed spawning eight MPI processes into the eight cores of a single processor. Figure 5.23a shows the performance evolution of two metrics (MIPS and LLC miss rate) for the main computing region of the application after applying the modifications stated in section 5.1.3. As seen before, concerning the performance, the application has two important routines according to their time invested, `sweepx` and `sweepy`. On the one hand, the `sweepx` routine shows a uniform performance instruction rate running at 4,000 MIPS and a uniform

[10] Time refers to the average duration of the time-stepper function in milliseconds. Core and Chip refers to the energy used in Joules by all the cores and the whole socket, respectively.

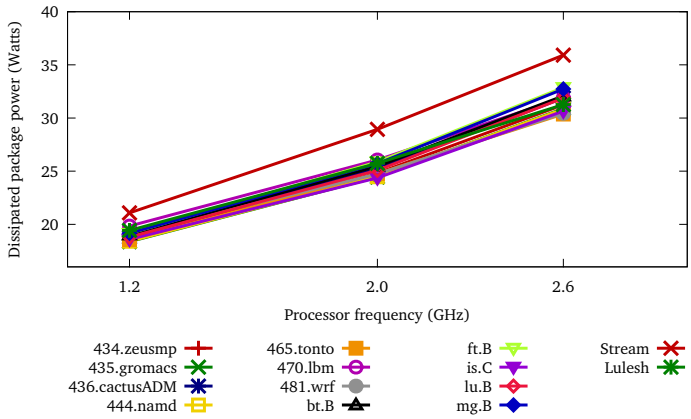
[11] Instructions is shown in millions. L1D, L2 and LLC refer to the ratio of L1, L2 and LLC with respect to the instructions executed. MIPS, $MIPJ_C$ and $MIPJ_P$ stand for, Millions of Instructions per Second, Millions of Instructions per Joule (using the core or the package energy counter, respectively).

Table 5.12
Time and energy consumption for the selected benchmarks executing at different frequencies.

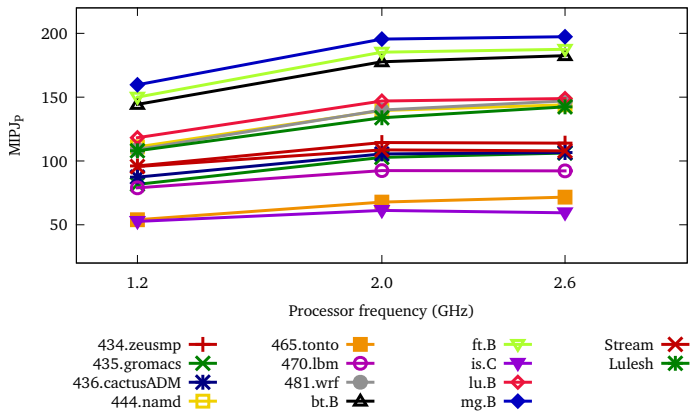
	2.6 GHz			*2.0 GHz*			*1.2 GHz*		
Benchmark	**Time**[10]	**Core**	**Package**	**Time**	**Core**	**Package**	**Time**	**Core**	**Package**
434.zeusmp	3,647	63,405	113,983	4,571	52,371	113,458	7,253	37,537	135,152
435.gromacs	85	1,455	2,626	111	1,221	2,714	186	932	3,417
436.cactusADM	766	13,857	24,426	977	11,482	24,670	1,586	8,542	29,810
437.leslie3d	175	3,168	5,671	226	2,646	5,782	370	1,985	7,043
444.namd	13,528	232,393	417,330	17,600	194,785	430,706	29,414	148,442	541,220
465.tonto	70,151	1,182,319	2,130,323	90,937	1,033,436	2,252,233	152,462	793,420	2,828,812
470.lbm	140	2,484	4,468	171	2,070	4,457	263	1,609	5,218
481.wrf	2,039	34,147	62,090	2,638	29,518	65,262	4,383	24,845	83,751
bt.B	1,383	25,322	44,361	1,789	21,440	45,560	2,953	16,458	56,119
ft.B	2,954	55,938	97,097	3,798	46,666	98,251	6,338	35,884	121,350
is.C	1,817	30,417	55,683	2,215	23,935	53,971	3,379	17,270	62,810
lu.B	912	16,443	29,073	1,174	13,600	29,451	1,946	10,453	36,606
mg.B	481	8,866	15,742	618	7,328	15,891	1,015	5,597	19,455
Stream	310	6,226	11,136	382	5,224	11,056	596	3,861	12,563
Lulesh	106	1,853	3,315	137	1,657	3,526	225	1,350	4,371

Table 5.13
Performance and energy metrics for the selected benchmarks using different processor frequencies.

					2.6 GHz			*2.0 GHz*			*1.2 GHz*		
Benchmark	**Instrs**[11]	**L1D**	**L2**	**LLC**	**MIPS**	**MIPJ$_C$**	**MIPJ$_P$**	**MIPS**	**MIPJ$_C$**	**MIPJ$_P$**	**MIPS**	**MIPJ$_C$**	**MIPJ$_P$**
434.zeusmp	12,988,177	2.2%	0.4%	0.2%	3,561.33	204.84	113.95	2,841.43	248.00	114.48	1,790.73	346.01	96.10
435.gromacs	278,947	1.2%	<0.1%	<0.01%	3,271.73	191.72	106.23	2,513.04	228.46	102.78	1,499.72	299.30	81.64
436.cactusADM	2,603,516	0.9%	0.4%	0.1%	3,398.85	187.88	106.59	2,664.81	226.75	105.53	1,641.56	304.79	87.34
437.leslie3d	716,501	3.0%	0.8%	<0.01%	4,094.29	226.17	126.34	3,170.36	270.79	123.92	1,936.49	360.96	101.73
444.namd	60,102,000	1.1%	<0.1%	<0.01%	4,442.79	258.62	144.02	3,414.89	308.56	139.54	2,043.31	404.89	111.05
465.tonto	152,627,913	1.3%	0.4%	<0.01%	2,175.71	129.09	71.65	1,678.39	147.69	67.77	1,001.09	192.37	53.95
470.lbm	411,983	5.2%	0.9%	0.2%	2,942.74	165.85	92.21	2,409.26	199.03	92.44	1,566.48	256.05	78.95
481.wrf	9,132,508	1.0%	0.2%	<0.01%	4,478.92	267.45	147.09	3,461.91	309.39	139.94	2,083.62	367.58	109.04
bt.B	8,099,479	1.9%	0.2%	<0.01%	5,856.46	319.86	182.58	4,527.38	377.77	177.78	2,742.80	491.32	144.33
ft.B	18,206,181	3.6%	0.9%	<0.01%	6,163.23	325.47	187.51	4,793.62	390.14	185.30	2,872.54	507.36	150.03
is.C	3,304,812	3.3%	0.5%	0.4%	1,818.83	108.65	59.35	1,492.01	138.07	61.23	978.04	191.36	52.62
lu.B	4,329,609	2.1%	0.4%	0.1%	4,747.38	263.31	148.92	3,687.91	318.35	147.01	2,224.88	414.20	118.28
mg.B	3,106,833	1.7%	0.4%	0.1%	6,459.11	350.42	197.36	5,027.24	423.97	195.51	3,060.92	555.09	159.69
Stream	1,201,408	5.2%	4.2%	0.9%	3,875.51	192.97	107.89	3,145.05	229.98	108.67	2,015.79	311.16	95.63
Lulesh	472,003	1.4%	0.3%	0.2%	4,452.86	254.72	142.38	3,445.28	284.85	133.86	2,097.79	349.63	107.99



(a) Average dissipated package power.



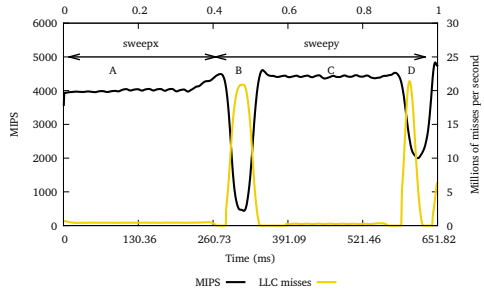
(b) Average MIP_P achieved.

Figure 5.22
Performance and power related metrics when executing multiple benchmarks on several processor frequencies.

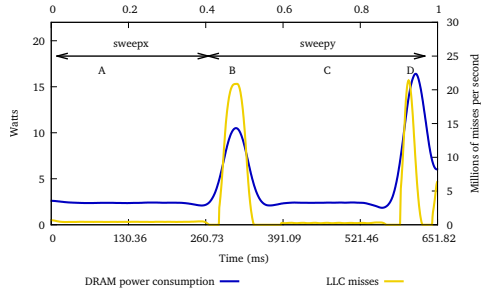
LLC miss rate (less than 1 million per second), except at the end, where the routine experiences a slight performance increase from 4,000 to 4,500 MIPS. On the other hand, `sweepy` presents different phases in terms of performance that are related to two parts of the code labeled as B, C and D that achieve 500, 4,500 and 1,800 MIPS, respectively.

Focusing on the sections that achieve lowest performance, the results show that phase B refers to the loop in lines 410-421 of file `sweep.f` as described earlier. This phase reached up to 20 millions of LLC misses per second per core (which translates into one miss every 25 instructions) and at that moment the DRAM consumption per socket reaches 12 Watts. Phase D involves the execution of the loop in lines 556-564 of the file `sweep.f` which partially undoes the work performed in Phase B by applying another matrix transposition. In this phase, the LLC miss rate increased again to 20 millions of misses per second (i.e. there was one miss every 90 instructions) and the dissipated power increased to 17 Watts.

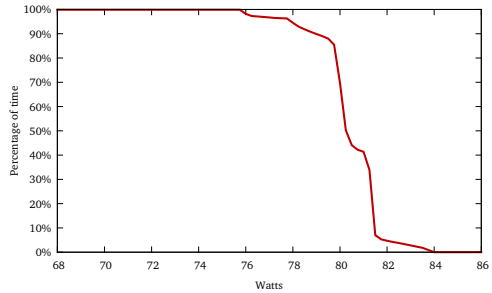
Since the processor contains the RAPL infrastructure that automatically increases the frequency based on several factors, including thermal and power conditions, it is also valuable to analyze the energy footprint of the application. For instance the plot in Figure 5.23c shows that during the 60% of the total execution time of Mr. Genesis, the socket consumes up to 80 Watts.



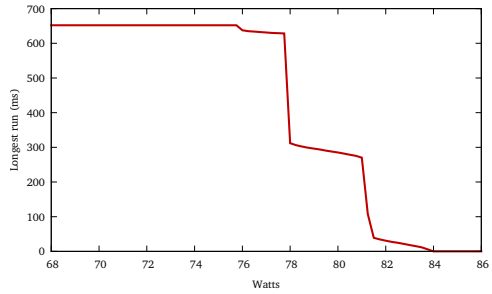
(a) Clustering results.



(b) Detailed performance progression of Cluster 1.

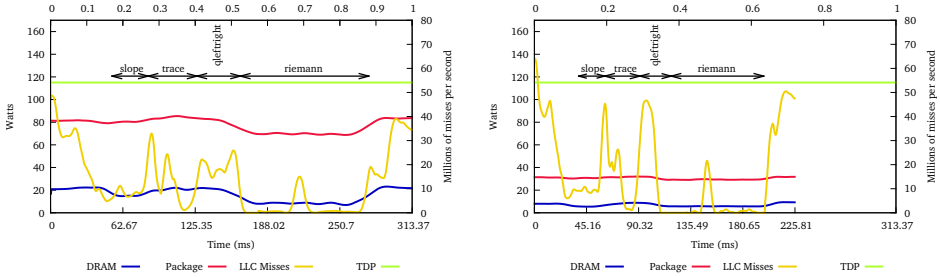


(c) Percentage of time above a power limit.



(d) Longest duration of the application above a power limit.

Figure 5.23
Performance and power consumption analysis of MR. Genesis in Altamira.



(a) 8 MPI processes per socket / *shared execution*.

(b) 1 MPI process per socket / *exclusive execution*.

Figure 5.24
Temporal evolution of the HydroC main computation region using a different number of MPI processes per socket.

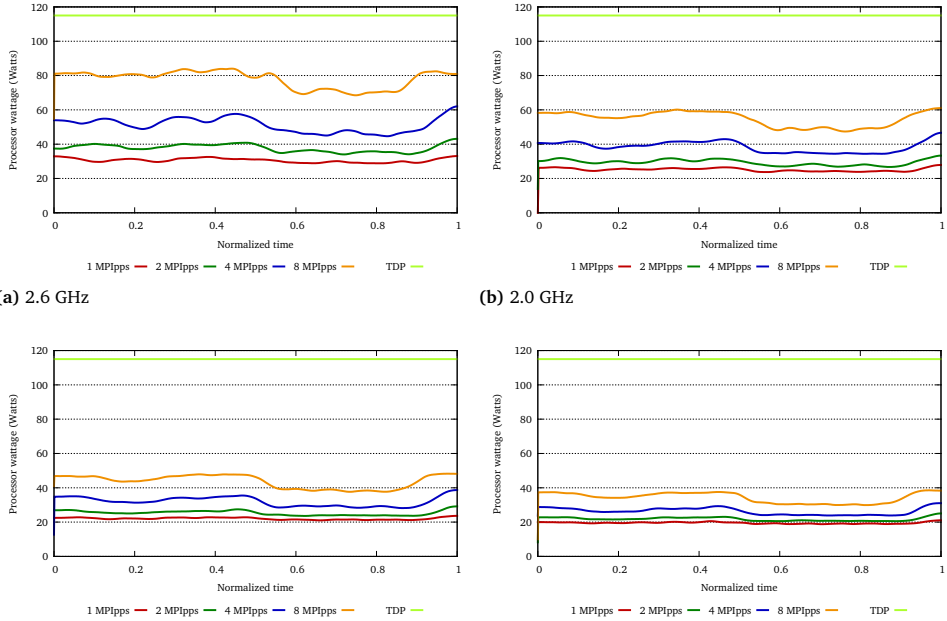
Then the consumption rapidly and in under 10% of the time the processor drains more than 81.5 Watts. It is interesting to note that the longest duration sustained in 81.5 Watts by the whole socket lasts under 40 ms as shown in Figure 5.23d. These values can be useful for the acceleration mechanism that speeds up the processor frequency in several directions. First, the processor could increase the frequency in the regions with lesser power consumption in a safe manner without surpassing the TDP. Second, these results also mean that limiting the power consumption to 81.5 Watts would only affect less than 10% of the application. Finally, if the acceleration mechanism has a duration limit of longer than 40 ms, it would be able to boost the application when entering the most power consuming regions of code.

5.2.3 HydroC

HydroC is a proxy benchmark of the RAMSES [175] application. This application solves a large-scale structure and galaxy formation problem using a rectangular 2D space domain split in blocks. It was instrumented and sampled in executions using eight MPI processes in two configurations to evaluate the performance and power when sharing the computational resources. One of the executions has placed all the processes into a single octa-core chip while the other has run one process on eight processors.

Figure 5.24 shows the temporal evolution of the power consumption of the DRAM at socket level and the LLC cache misses per core within the time-stepper routine when using one and eight MPI processes per socket (MPIpps) running at 2.6 GHz. The results obtained show a tight correlation between the rate of LLC misses and the power consumed by the DRAM. This seems natural and expected because the higher the miss rate the more data movement from/to memory, so increasing its energy consumption. This effect is most noticeable when using all the cores (as shown in Subfigure 5.24a) because the power consumption counter reflects the accumulated energy consumption by the whole socket and the signal presents a wider amplitude.

The use of the source code referencing capabilities helps to delimit the routines on the plots according to the routines executed across time. The comparison of the two subfigures (5.24b and 5.24a), shows an increase of the DRAM power consumption in the routines `trace` and `qleftright`. The consumed energy moves from 10 to 20 Watts when using one or eight MPI processes per socket, respectively. The `riemann` routine shows better power scalability when moving from one to eight processes per socket. In this routine, while DRAM consumes approximately 5 Watts on the exclusive execution and increases up to 8 Watts when executing in shared mode. These results point out that, even multiplying by eight the number of cores accessing to the memory, the power consumption due to DRAM is only multiplied by a factor ranging from 1.6 to 3. Since the power metering reports measurements for the whole chip, it becomes evident that the shared execution becomes more power efficient; so placing all the processes in the lowest number of chips reduces the aggregate power consumption.



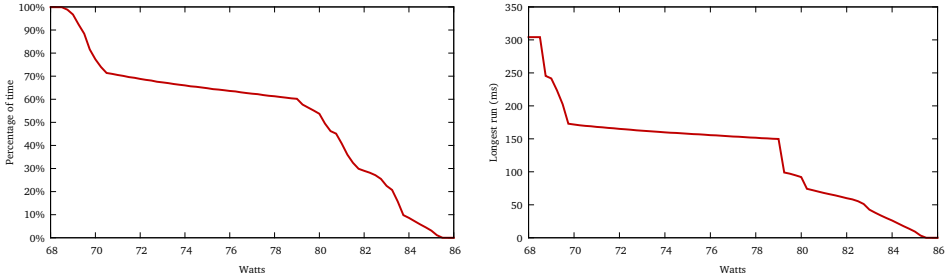
(a) 2.6 GHz **(b)** 2.0 GHz

(c) 1.6 GHz **(d)** 1.2 GHz

Figure 5.25
Progress of the power consumption in the main computation region using combinations of MPI processes per socket (MPIpps) and processor frequencies.

On the other hand, the performance when all cores of the processor execute the application the execution time takes longer than employing one core per socket (313.37 *vs* 225.81 ms). For instance, the results for one MPI process indicate that the LLC miss rate per core decreases in the shared execution on `qleftright` compared to the exclusive execution and consequently, the duration of the routine increases. The observed increase occurred because the shared execution involved seven additional MPI processes running and therefore, competing for the shared resources as the LLC cache. As a consequence of sharing these resources, the duration of the routine `qleftright` increased from 8% to 12% (from 18 to 37.5 ms) when moving from the exclusive to the shared execution. In addition, the `riemann` routine also increased its duration, from 95 to 113 ms, although its proportional duration decreased from 42% to 36%. According to the performance data gathered, the total number of LLC misses per core increased by 12.5% when moving from the exclusive to the shared execution. This effect is reasonable because the LLC is a shared resource among the cores within the socket and cannot sustain all the requests from the cores with the increased number of LLC misses experienced by the application.

As with the sequential benchmark executions, HydroC was executed varying the processor frequency as well as the number of processes per socket. The results of the progression of the power consumption within the computation region are depicted in Figure 5.25. Each of these plots has the X-axis (representing time) normalized, so each experiment takes the visually the same to complete. The shape of the different plots is almost the same except for the amplitude, which varies according to the processor frequency. When the system is fully occupied and the processor clock rates at 2.6 GHz, the power dissipates by HydroC presents two modes: 70 and 80 Watts. When comparing the processor specifications and the results obtained, the application consumes about 70% of the processor maximum TDP. The same Figure indicates that the power consumed does not increase linearly with respect to the number of executing cores. This observation can be explained because in the executions with one, two and four MPIpps the idle cores are not halted and still consume energy.



(a) Percentage of time above a power limit. **(b)** Longest duration of the application above a power limit.

Figure 5.26
Energy footprint for the execution of HydroC when using eight MPI processes per socket running at 2.6 GHz.

Table 5.14
Scalability of SIESTA. Energy is shown in KJoules and Duration is shown in seconds.

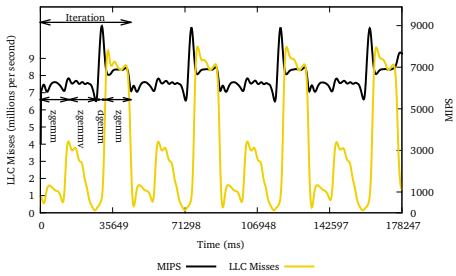
# processes	Energy	Duration	Speedup	Parallel Efficiency
16	8205	51824	1	-
32	8296	27375	1.89	0.96
64	9885	16252	3.19	0.80
128	13373	10883	4.76	0.60
256	25610	11001	4.71	0.29

Finally, the last analysis of the application explores the amount of time that the processor is above a power consumption rate. This type of analysis may prove helpful on how the RAPL infrastructure of the Intel SandyBridge processors benefit from the folded results. The plot in Figure 5.26 shows the energy footprint of the application depicting the percentage of time and also the longest time-frame above a power limit when using all the cores from a processor. For instance, the results in Subfigure 5.26a indicate that the application drains up to 80 Watts for about half of the execution and only 10% of the whole execution needed more than 84 Watts. These results show that there is room for power consumption and the processor may decide to overclock itself to run faster. Subfigure 5.26b depicts the duration of the computation according to its consumed power. The plot reveals that those regions that consume more than 84 Watts last less than 25 ms. With such information, the processor may determine to increase the clock rate for the periods where the power consumption is far from TDP. Then, the processor should restore the clock rate for the periods in which power consumption is high (which are typically short).

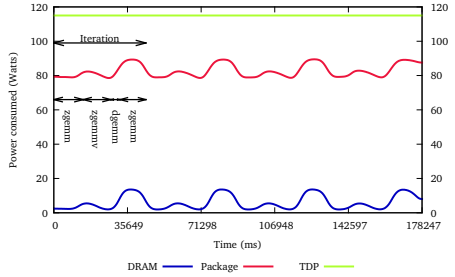
5.2.4 SIESTA

SIESTA [200] is a software implementation for performing electronic structure calculations and *ab-initio* molecular dynamics simulations of molecules and solids. It implements a self-consistent density functional method using standard norm-conserving pseudopotentials and a flexible, numerical linear combination of atomic orbitals basis set. The application was executed varying the number of processes from 16 to 256 using the two available sockets of each node. Table 5.14 shows several application properties when using the selected number of MPI processes and the tabulated results show that the application does not scale linearly in terms of either performance or energy consumption. The analysis discussed here focuses on the execution that employs 128 MPI processes because it achieves the best performance, even though its efficiency could be considered moderately low (0.60).

The performance of the main computation of SIESTA is depicted in Figure 5.27a and it shows a pattern repeated four times. An inspection of the data gathered shows that the iteration mainly invokes the routines `zgemm`, `zgemmv`, `dgemm` and again `zgemm` which are provided by the Linear

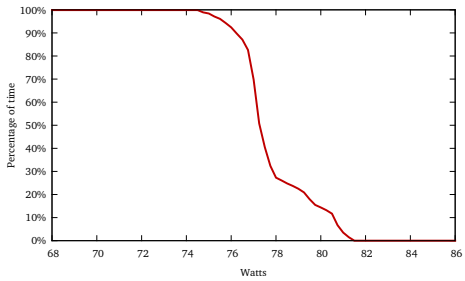


(a) Performance characterization.

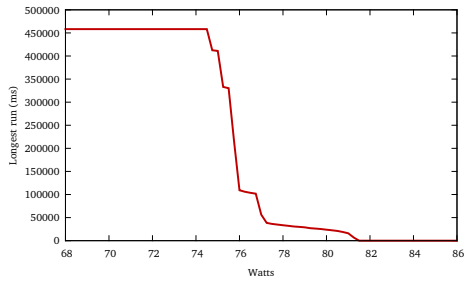


(b) Power characterization.

Figure 5.27
Performance and power characterization of the SIESTA application.



(a) Percentage of time above a power limit.



(b) Longest duration of the application above a power limit.

Figure 5.28
Energy footprint for the execution of SIESTA on one (1) socket running at 2.6 GHz.

Algebra Package (LAPACK[16]). With respect to performance, the application reaches a range from 5,500 to 9,000 MIPS. The results show that the most performing routine is `dgemm` and also that `zgemm` presents two forms of behavior in MIPS and LLC miss rates. In terms of energy, Figure 5.27b shows that the power drained is about 80 Watts, increasing when the code executes the `zgemmv` and the second call to `zgemm` up to 90 Watts. Such an increase corresponds to the highest LLC miss rates observed in the application.

Regarding the energy footprint, the processor does not typically consume more than 80 Watts (about 70% of the TDP) when running the application, as depicted in Figure 5.28. In this case, these results show that the regions of code that consume more than 80 Watts do not last more than 25 seconds. So, the processor typically drains a uniform amount of power and requires more power for certain short periods. More precisely, considering the same 81.5 Watts as in the previous example, the results show that SIESTA would exceed this limit for 7 seconds approximately. Consequently, it is reasonable to assume that these analyses may help to boost the processor frequency whenever the application runs in a scenario where the power drained is below the TDP.

5.3 Final remarks

This chapter has focused on analyzing several first-time seen HPC applications of a broad research areas by employing the framework described in this thesis. The reports generated by the framework were turned into relatively simple and small modifications that improved

[16] `http://www.netlib.org/lapack`

Table 5.15
Summary of the applications' modifications and their achieved overall improvement.

Application	Symptoms	Modification applied	# processes	Improvement
Arts_CF	RoB full due to long FP instructions	Replace FP divs with FP mults	512	9.0%
BigDFT	Poor temporal data locality (cache)	Loop interchange	1,024	14.6%
CGPOP	Poor temporal data locality (cache)	Loop fusion & code overlap/inline	24	10.0%
			96	7.2%
			192	7.2%
			384	<0.1%
GTC	Poor temporal data locality (cache)	Data prefetch	256	6.2%
Mr. Genesis	Poor spatial/temporal data locality (cache, TLB)	Loop interchange	16	30.0%
			128	34.6%
			256	38.8%
			512	32.8%
Nemo	Code replication and matrix usage	Code removal & convert matrix into scalar values	128	1.0%
PEPC	Poor spatial/temporal data locality (cache, TLB)	Loop fusion	16	14.0%
PMEMD	Stalled cycles due to long FP instructions	Memoization	4	26%
			64	11.3%
			128	11.3%
			256	11.2%

the node-level performance on eight applications, so demonstrating the productivity of the framework. Table 5.15 summarizes for each application, the nature of the symptoms observed, the modifications applied as well as the overall application improvement.

In addition to such a summary, the findings observed through the different applications allow the authors to extract additional and more general comments. First of all, programs, compilers, systems and performance tools evolve along time and become more complex. It is therefore relevant to invest some time to evaluate the application performance to find out whether the performance achieved is appropriate. These applications have reached an in-production status and despite having been used for a long time there are still opportunities for improving their performance. Even upgrading to newer compilers and using aggressive compiler optimizations, the results of the framework described here have led to effortless changes to the source code that impact on the application performance.

In fact, compilers include numerous optimization rules and heuristics that allow them to tune the resulting binary code when translating the source code into binary code. However, compilers have to honor the application semantics; so, they cannot change the code wildly and they have to stick to the application developer's requests. While the user can allow the compiler to relax some of the application semantics and use even more aggressive optimization rules, there are still other reasons that prevent the processor's peak performance from being reached. For instance, since the processor has a limited number of logical registers, the compiler needs to use spill instructions to handle all the variables in the code region. Moreover, it is difficult for the compiler to anticipate how many cycles an instruction will require to complete and also the application may not expose sufficient instruction-level parallelism to overlap those stalled cycles. Considering the large exploration space, the optimization process becomes a daunting time-consuming task that may not offer the best possible code.

Performance analysis tools, either in the form of profile or a trace-based, help the analyst to locate the bottlenecks, but the framework discussed here excels in locating and identifying the nature of the node-level performance issues. Regarding the performance issues, it is widely accepted that the memory gap is the main performance limiting factor. This factor is truly the nature for the performance drop on several applications analyzed (e.g. CGPOP, Mr. Genesis, PEPC, GTC and BigDFT, among others). However, there are other bottlenecks still to be considered such as long floating-point dependency chains (as observed in Arts_CF, CESM and PMEMD, for instance). Each of these applications have been optimized using different approaches according to the source code that exposes the problem. The transformation applied in Arts_CF is simpler because it focuses on swapping multi-cycle floating-point instructions for shorter instructions. CGPOP exposes four phases stressing different processor units and the modification mitigates such intensive use by combining consecutive phases to increase the distance between dependencies. With respect to PMEMD, the application does not suffer from the memory gap and the application semantics allows pre-calculating intermediate values; so new memory structures effectively increase the floating-point dependency distances. Consequently, overlapping computation regions with different bottleneck natures may increase the overall performance in applications.

Finally, the most important fraction of the power dissipated by a processor relates to the static part. Consequently, to effectively reduce the energy consumed by a system it is appropriate to follow the *haste and halt* approach. This approach means executing the application at the maximum speed and then stopping the processor until more work is ready. Of course, performance analysis tools also help to reduce the energy consumed by a processor as they help to reduce the time-to-solution and so the energy consumed by the system.

6

Conclusions and future research directions

```
‘‘You’re a monster.
Thanks.  Does this mean I get a raise?
No, just a medal.  The budget isn’t inexhaustible.’’
```

— Orson Scott Card, ENDER'S GAME

Previous chapters of this thesis have presented and demonstrated the usefulness of the main contribution of this thesis: the folding mechanism. This mechanism depicts instantaneous metrics within repetitive code regions ensuring minimum overhead during the application run. While most of the metrics are heavily tailored to performance analysis, it should be noted that it is possible to apply this mechanism to other concerns such as power analysis and call-stack and memory references. The mechanism has become a component of a framework, in addition to an already existing tool and it has motivated the proposition of an analysis methodology.

6.1 Conclusions

This thesis has been framed in the application performance analysis area and its main contribution is a process named folding. The folding mechanism provides instantaneous performance metrics using minimum instrumentation and coarse grain sampling and consequently, ensures very low overhead (less than 5%). This mechanism takes advantage of the repetitive behavior found in many applications, especially within the HPC environment given that most of these applications are written as a repetitive sequence to calls that contain loops due to the nature of the problems they solve. The folding creates a synthetic representation of these regions to depict their instantaneous and accurate performance evolution. The mechanism smartly combines performance data captured using instrumentation and sampling methods to report the progression of several metrics accurately, including performance and energy hardware counters, source code and memory references. This thesis refers to a previous work that identifies the application structure according to performance metrics. The combination of both results in a framework that provides instantaneous metrics of the most dominant computing regions begetting a methodology of use. This framework excels in evaluating the serial node performance of several parallel applications. These evaluations have helped to improve the performance in some of these applications by manually applying small and well-known optimization techniques (such as loop reorder, split and fusion, data prefetch and memoization).

6.1.1 Performance metrics

This thesis has presented the instantaneous progression of the hardware performance counters by using the associated counters from the folded samples and the delimiting instrumentation points. To this end, the folding constructs a continuous function based on these counters and then applies a contouring algorithm. This thesis has explored two different contouring algorithms: a Gaussian process named Kriging and piece-wise linear regressions. The results obtained using either approach are satisfactory in most situations, but they exhibit particular problems in very specific situations. The Kriging results expose variability that resemble noise when the behavior of the regions present variations, while the piece-wise linear regressions require a proportion of consecutive samples to identify performance phases. Still, the results of these approaches provide metrics that resemble measurements obtained using higher sampling frequencies.

With respect to the performance metrics, the folding depicts the instantaneous value of the performance counters. The increasing quantity and semantics complexity of the performance counters respond to the growing intricacy of the processors and pose a problem when interpreting their results and by extension, the folding results. The folding mechanism has adopted analytical performance models built on top of performance counters to facilitate the understanding of the application performance behavior. That way, the folding generates simplified reports that depict the progression of the overall performance (i.e. MIPS) in addition to the nature of any performance inefficiency.

This thesis has also covered the analysis of several in-production applications in order to demonstrate the usefulness of the folding. Within these analyses, the reader has encountered several facts worth mentioning.

- A large computation fraction of the execution time is spent on a tiny portion of the source code, honoring to some extent the 80/20 rule[1]. This fact, when combined with Amdahl's, law motivates the analysis methodology described in this thesis. The methodology aims at exploring the (few) most time-consuming regions of code as the return would be higher if there were any chance of enhancing the region performance. This thesis has shown that very small and simple code modifications in tiny regions have resulted in performance improvements up to 34%.

- The identification of the performance flaws and the corresponding source code is helpful for understanding the application behavior and modifying the application code to improve its performance. Even if the compiler considers many optimization transformations, there are situations that prevent the compiler from applying them. This thesis has shown that simple source code changes still play an interesting role in software development because they boost the application performance when the compiler is not using them. However, it is worth mentioning that not every application may benefit from such manual modifications. In fact, it looks more promising using algorithms that require lower computational intensity (i.e. instructions) to shorten the time-to-solution, but this work direction typically requires additional application understanding.

- Sometimes there are reasons that prevent the application developer from sharing the application source code with the analyst. Still, simple questions from the analyst to the developer may provide significant insight in understanding the behavior of the application. For instance, a developer of one application analyzed throughout this thesis was not allowed to share the application source code. However, the folding clearly exposes performance bottlenecks and their location. When the developer was requested the surrounding code lines, he accepted and the later analysis helped to improve the application performance.

- Multithreaded/multicore environments are difficult to analyze, in particular, when exploring interactions between cores and the shared resources. The folding mechanism has exposed the performance behavior in multi-core executions and its results have indicated

[1] The Pareto Principle, sometimes referred as the 80/20 rule, states that a small proportion of products in a market often generate a large proportion of sales.

the interaction between processes executing in different cores and pointed out the reasons why this occurs.

6.1.2 Source code time-evolution

The correlation between performance and source code is essential for determining which portions of the application behave poorly and those that behave well. This thesis has evaluated two approaches to associate performance and the source code.

The first approach relies on the fact that applications are written as a sequence of loops and that each performs differently. This approach focuses on identifying phases with uniform performance within the shape of the folding results by taking advantage of the piece-wise linear regressions. Once the regression identifies these phases, the mechanism correlates the performance observed on each of these phases to the loops according to the sampled source code reference. That way, the phases define a mapping between application performance and source code regions. The work in this thesis has extended the Cube code visualizer from the Scalasca [78] project so that it presents this mapping to the user.

The second approach addresses the correlation between performance and source code without considering the shape of the performance metrics but by exploring the temporal evolution of the call-stack segments captured at sample points. This approach has presented a two-phase mechanism to provide the temporal evolution of the call-stack segments. The first phase creates a temporal sequence of call-stacks; however, given that only a segment of the call-stack is captured call-stack then needs to be reconstructed. This thesis has therefore presented a bio-inspired algorithm that identifies regions of similarity between call-stack segments in order to deduce the necessary line of call-path ancestors. The second phase selects a handful representative routines that are closer to the application activity and dismisses the extremely short routines according to the user requests. This mechanism has served to determine the duration the representative routines, observe their chronological order and display the evolution within the source code, in addition to correlating them with performance plots.

6.1.3 Address-space time-evolution

This thesis has also presented an extension to the folding mechanism that displays the memory access patterns of computing region and their time evolution along time and correlated with source code and performance. This extension relies on the ability of recent hardware mechanisms available in current processors to sample instructions and then attribute performance data that includes the addresses referenced to each sample. The extension has proved valuable for giving detailed insight into optimized application binaries, such as the detection of the most dominant data streams and their temporal evolution along computing regions. The analyses provided have shown situations in which the compiler has replaced a complete loop by a function call, multiple and simultaneous memory streams for particular regions of code and identified possible redundant work.

6.1.4 Energy metrics

This thesis has also shown the usefulness of employing novel capabilities from recent processors to obtain power and energy metrics and the correlation with performance metrics. These capabilities have allowed the evolution of both the performance and the power to be studied in a wide variety of serial benchmarks and also in parallel applications executed in different scenarios within production systems. The availability of these type of tools allows continuous development of HPC systems and improves their use not only in performance, but also in energy dimensions.

With respect to the results, the most significant fraction of the power dissipated relates to the static part and consequently, the effective way to reduce the energy consumed is following the *haste and halt* approach. This approach refers to the running of the application at the maximum speed and then stop the processor until more work is ready for execution. Still, the folding results have exposed that both performance and energy consumption are influenced

by similar factors (such as the application executed, the processor frequency, the occupancy of the socket). In particular, the results indicate that the DRAM power consumption presents the higher variability along the computing region as a result of the DRAM memory accesses. In this direction, improving the access to the memory hierarchy serves to reduce the power consumption but also the execution time and consequently, the overall dissipated energy. In addition, the results have also shown that applications do not reach the processor TDP and that regions with peaks of energy consumption are relatively short in duration. These energy footprints are valuable for determining when to enable the acceleration of low-consumption phases and limit the acceleration for a certain duration of the execution.

6.2 Future research directions

This work has covered several aspects to describe finely the progression of the performance metrics. While these descriptions have been valuable to increase the performance of several applications, this research also have arisen several open lines and future work that are described below.

6.2.1 Alternative sampling sources

Sampling is one of the monitoring mechanisms used for capturing punctual data and in this thesis, so it is crucial apply the folding. In this environment, sampled data enables the folding mechanism to provide performance metrics as the instrumented region progresses. All the experiments in this thesis have used time-based (or cycle-based) sampling mechanisms, which ensures that the collected samples run independently with respect to the application activity. Extrae was also been extended within the frame of this thesis to integrate variations in the sampling periods to ensure that samples are randomly distributed among the computing regions, whatever their duration.

Still, it is possible to use alternative sampling sources as, for instance, hardware counters. In this situation, the processor invokes the sampling handler every time a given number of accounted events by the PMU have occurred. As the evolution of the performance counters (such as instructions, branches and cache misses) and the application activity are tightly correlated, their use would help to locate regions of code with specific performance inefficiencies. For instance, when searching for data cache misses, sampling through the cache miss performance counter would capture most samples in those regions of code where the cache misses is higher. Consequently, the number of samples and the quantity of details would be higher in these particular regions. However, addressing this problem through this direction also poses a drawback because the number of samples is uneven and there may be regions with little or no samples at all.

6.2.2 Apply folding to non-user regions

The application analyses in this thesis have mainly focused on user regions because these represent the largest part of the computation time and their analyses may encourage subsequent improvements from the developer side. Still, nothing prevents the folding mechanism from being applied to non-user regions of code, such as regions of code relating to the MPI or OpenMP run-times, and from helping the developers of these run-times to gain new insight. There may be several interesting studies applied to these run-times. For instance, it would be interesting to detect when an incoming message actually arrives at an MPI process. PMPI instrumentation [70] does not currently uncover this information. However, certain experiments indicate (as oneself expects) that copying the message from the network device to the main memory generates a quantity of data cache misses and this could be observed using the folding. In OpenMP applications, it would be valuable to detect whether a thread in a spin (i.e. busy-wait) lock degrades the memory performance of the remaining threads due to continuous pooling of the memory bus. While the regions associated with the routines of these run-times represent

repetitive patterns, they are likely to present duration variability between invocations; so, the folding mechanism may require some changes to address this variability.

6.2.3 Expert systems

Despite the use of the analytical performance models, the interpretation of the values of the performance counters requires some processor architecture knowledge due to the complex semantics of the performance counters. This requirement prevents some users from using performance tools and this has prompted approaches that easily map the performance counter information into valuable insight that is likely to improve the application performance after exploring the application behavior. Some performance tools have already included rule-based (or expert) systems to point out the performance bottlenecks and even, include some guidelines for increasing the application performance. The detailed results provided by the folding mechanism in both performance and source code dimensions make it reasonable to adopt an expert system to provide textual reports to the analyst and hints on how to improve the application performance.

6.2.4 Syntactic-level application reconstruction

The framework described in this thesis provides an enormous amount of application details that cover performance, source code and even, data structures. This thesis offers several examples that illustrate how an analyst interprets the results of the framework, even without any prior knowledge of the application source code structure. On the other hand, application developers know the application code structure, but they may find it difficult to follow the analyst findings and map them out in the application where these are not sufficiently clear. However, the chances for analysts to report on the application using the source code depends on the availability of the application source code. So where the source code is not available, they will need to describe the application structure from the folding results. This description is likely to respond to a mechanical exploration of the folding results and interpreting when certain activity such as routine enter and exits occur. The exploration of the results exposes information such as the presence of loops or calls to subroutines. It, therefore seems valuable to reconstruct the application at the syntactic level. This type of output would allow analysts and developers to understand the application, while making it easier for developers to integrate them, thanks to the language being closer to the application source code structure. In addition, this source code reconstruction may be annotated with performance metrics providing an additional analysis experience.

Appendices

U
User guide

U.1 Quick start guide

The Folding is a mechanism that provides instantaneous performance metrics, source code references and memory references[1]. This mechanism receives a trace-file (currently generated by Extrae- see further details on generating a trace-file for the Folding in Appendix G) and generates plots and an additional trace-file depicting the fine evolution of the performance. The Folding uses information captured through instrumentation and sampling mechanisms and smartly combines them. In this context, the samples are gathered from scattered computing regions into a synthetic region by preserving their relative time within their original region so that the sampled information determines how the performance evolves within the region. Consequently, the folded samples represent the progression in shorter periods of time no matter the monitoring sampling frequency, and also, the longer the runs the more samples get mapped into the synthetic instance. The framework has shown mean differences up to 5% when comparing results obtained sampling frequencies that are two orders of magnitude more frequent.

U.1.1 Decompressing the package

The Folding package is distributed in a .tar.bz2 file that can be uncompressed in the working directory by executing the following command:

```
# tar xvz folding-1.0rc8-x86_64.tar.bz2
```

where `folding-1.0rc8-x86_64.tar.bz2` refers to the Folding package as distributed from the BSC web page[2].

U.1.2 Contents of the package

After decompressing the package, the working directory should be populated with the directories (and corresponding descriptions) as listed in Table U.1.

[1] This last option is experimental at the moment of writing this document
[2] http://www.bsc.es/computer-sciences/performance-tools/downloads

Table U.1
Contents of the folding package.

Directory	Contents
bin/	Binary packages
etc/	
extrae-configurations/	Minimal configuration files for Extrae
models/	Configuration files to calculate performance models
basic/	
ibm-power5/	
ibm-power7/	
ibm-power8/	
intel-haswell/	
intel-nehalem/	
intel-sandybridge/	
include/	Header files for the development of 3rd party tools
lib/	Libraries for the folding
share/	Miscellaneous files
cfg/	Configuration files for Paraver
doc/	Documentation
html	
examples/	
folding-writer/	Example on how to generate data for the folding
user-functions/	Sample tracefile with manually instrumented regions
clusters/	Sample tracefile with automatically detected regions

U.1.3 Quick run

This section provides examples of two types of execution of the Folding tool. These examples take benefit of the included sample trace-files from the package. For further information on how to generate trace-files for the Folding tool, check Appendix G.

U.1.3.1 Applied to manually instrumented regions

This first example uses a trace-file from the 444.namd SPEC benchmark that contains manually instrumented information that is located in

```
${FOLDING_HOME}/etc/share/examples/user-functions
```

This trace-file was generated by Extrae and delimiting the main loop using the Extrae API[3], more precisely the `Extrae_user_function` which emits events with label `User function` (or event type 60000019). To apply the Folding process to this trace-file, simply execute the following commands:

```
# cd ${FOLDING_HOME}/etc/share/examples/user-functions
# ${FOLDING_HOME}/bin/folding 444.namd.prv "User function"
```

U.1.3.2 Applied to automatically characterized regions

This example consists of a trace-file for the Nemo application when executed in MareNostrum3. This trace-file contains information regarding automatically characterized regions. This characterization has been done using the Clustering tool[4]. This tool enriches the trace-file by adding

[3] Please refer to `http://www.bsc.es/computer-sciences/performance-tools/documentation` for the latest Extrae User's Guide.

[4] Please refer to `http://www.bsc.es/computer-sciences/performance-tools/documentation` for the latest documentation with respect to the Clustering tool.

events labeled as `Cluster ID` (and event type 90000001) into the trace-file. In this context, these events identify similar computation regions based on the event value. To apply the Folding process to this trace-file, simply execute the following commands:

```
# cd ${FOLDING_HOME}/etc/share/examples/user-functions
# ${FOLDING_HOME}/bin/folding \
  nemo.exe.128tasks.chop1.clustered.prv "Cluster ID"
```

This trace-file also contains all the necessary performance counters in order to take benefit of several performance models based on performance counters. Simply add the `-model intel-sandybridge` option to the Folding script to generate the plots with information of the models instead of providing each performance counter individually. The commands to execute should look like this:

```
# cd ${FOLDING_HOME}/etc/share/examples/user-functions
# ${FOLDING_HOME}/bin/folding -model intel-sandybridge \
  nemo.exe.128tasks.chop1.clustered.prv "Cluster ID"
```

U.1.4 Exploring the results

The Folding mechanism generates two types of output inside a directory named as the trace-file given (without the `.prv` suffix). The first type of results include a set of gnuplot files where each of these represents the evolution of the performance counters within the region. The tool also generates a Paraver trace-file with synthetic information derived from the Folding mechanism.

U.1.4.1 Using gnuplot

With respect to the gnuplot files, the Folding mechanism generates as many files as the combination of analyzed regions (clusters, OpenMP outlined routines, taskified OmpSs routines, or manually delimited regions) and the counters gathered during the application execution. The user can easily list the generated gnuplot files calling `ls *.gnuplot` within the directory created. The name of the gnuplot files contain the trace-file prefix, the identification of the region folded, and the performance counter shown. For instance, the example described in Section U.1.3.1 generates output files that can be explored by executing the command:

```
# gnuplot -persist \
  444.namd.codeblocks.fused.any.any.any.main.Group_0.PAPI_TOT_INS.\
  gnuplot
```

When executing the aforementioned command, the `gnuplot` command should open a window that resembles that in Figure U.1[5]. The Figure shows that the application faces six phases that execute at 4,500 MIPS approximately. Most of the code occurs in three code locations (being line 76 the most observed line), and we also observe that phases related to high MIPS are related with the activity in the middle of the code-line plot.

This file refers to the user routine `main` (which was manually instrumented) of the trace-file 444.namd.prv and provides information of the total graduated instructions (`PAPI_TOT_INS`). The user will notice that there are additional files for the different performance counters and they can explore them individually. The Folding also generates an additional plot that combines the metrics of all the counters into a single plot. This plot mainly provides information with respect to the MIPS rate (referenced on the right Y-axis), and ratio of the remaining performance counters per instruction (referenced on the left Y-axis). For the particular case of the example from Section U.1.3.1, this plot can be explored calling:

[5] Warning! If the user has problems to open the gnuplot, they should check whether the gnuplot installation is compatible and supports the default terminal. Otherwise, simply select the appropriate terminal (or leave it blank) in the first four lines in the gnuplot script

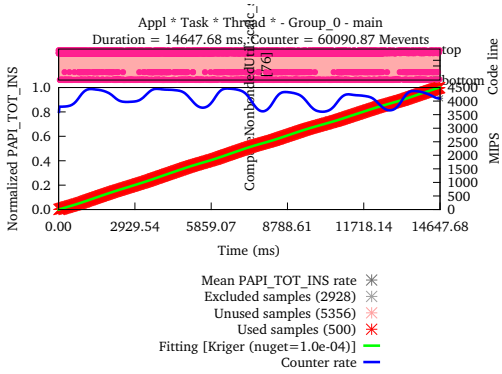


Figure U.1
Evolution of graduated instructions for 444.namd.

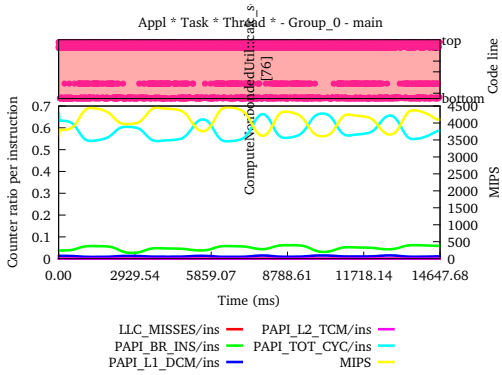


Figure U.2
Evolution of multiple counters for 444.namd.

```
# gnuplot -persist \
  444.namd.codeblocks.fused.any.any.any.main.\
  Group_0.ratio_per_instruction.gnuplot
```

This command should generate an output combining all the performance counter slopes as shown in Figure U.2.

The aforementioned instructions also apply to the automatically delimited example described in Section U.1.3.2. In this case, the region names are numbered as `Cluster_1` to `Cluster_11`, but they also contain the trace-file prefix and the performance counters to explore them individually. If the user requested the performance models, then additional gnuplot files are created to provide information regarding these models. For the particular case of the Intel SandyBridge model, it generates three models that always generate the MIPS rate and add different metrics:

- *instruction-mix*
  Gives insight of the type of instructions executed along the region.

- *architecture-impact*
  Provides information regarding to the cache misses at different levels and the branch mispredictions along the region.

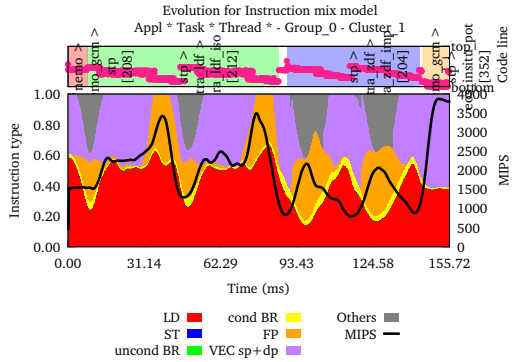


Figure U.3
Instruction mix decomposition for Cluster 1 of Nemo.

- *stall-distribution*
  This plot shows information regarding on which components of the processor are stalling the processor pipeline.

For instance, to open the instruction mix for the region labeled as Cluster 1 of the Nemo application executed in Section U.1.3.2, the user needs to open the plot invoking the commands below and should obtain a plot similar to Figure U.3. The reader may see that the application shows two distinctive phases (green and blue) and within each of them there are two repetitions of the same performance.

```
# gnuplot -persist \
  nemo.exe.128tasks.chop1.clustered.codeblocks.fused.any.any.any.\
  Cluster_1.Group_0.instructionmix.gnuplot
```

The tool also provides a GUI-based tool to explore the plots. the user may invoke a visualizer named `wxfolding-viewer`, by invoking it from the newly created directory such as:

```
# ${FOLDING_HOME}/bin/wxfolding-viewer *.wxfolding
```

U.1.4.2 Using Paraver

The Folding process generates a trace-file with a suffix `.folded.prv` that lets Paraver to display some parts of the folded results. The Folding package includes several configuration files in the `${FOLDING_HOME}/share/cfg` directory for Paraver to help analysing the results. From the configuration files contained in that directory, we outline the following:

- *views/*
  - `win_folded_type.cfg`
    Generates a time-line that shows in which instances the Folding results have been integrated. This helps correlating the original trace-file and its contents with the folded trace-file. Notice that only one instance per type (where type refers to function, cluster, *etc...*) is folded.
  - `win_folded_mips.cfg`
    Generates a time-line showing a signal of the MIPS rate within the folded instances. See in Figure U.4 a time-line depicting the MIPS rate achieved in the example trace-file for 444.namd.

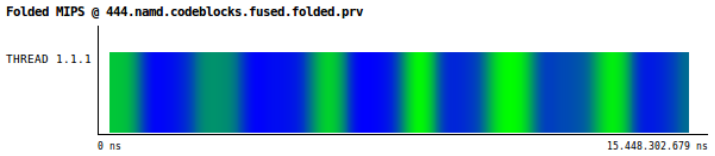


Figure U.4
Evolution of graduated instructions for 444.namd in Paraver.

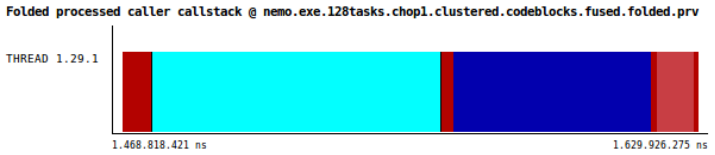


Figure U.5
Paraver time-line showing the callers for Cluster 1 of Nemo.

  - `win_folded_processed_call-stack_caller.cfg`
    Generates a time-line showing the most time-dominant routines as they have been executed within the folded instances. Figure U.5 shows a time-line depicting the called routines for the Nemo example.
  - `win_folded_processed_call-stack_callerline.cfg`
    Generates a time-line showing the most time-dominant source code references (as pair of line and file) as they have been executed within the folded instances.

- *histograms/*
  - `3dh_folded_mips_per_caller.cfg`
    Generates a histogram that shows the achieved MIPS rate depending on the caller (columns) for a particular region folded.
  - `3dh_folded_mips_per_callerline.cfg`
    Generates a histogram that shows the achieved MIPS rate depending on the source code references (pair of line and file in columns) for a particular region folded.

U.2 Configuration, build and installation

This section describes how to build and install the Folding package. The Folding package (and its dependencies) requires the Boost library (only the headers suffice), a C compiler, a Fortran compiler and a C++ compiler that supports the C++ 2011 specification (such as g++ version 4.8). This package optionally uses the `strucchange` package from the R statistical application (and may execute in parallel if the `doParallel` is available) to use the piece-wise linear regression interpolation mechanism. Additionally, the Folding package requires the libtools package to be installed. This package helps on the parsing of Paraver trace-files and can be downloaded from the BSC download web page.

U.2.1 Libtools package

This package can be downloaded from the BSC web page and requires the boost header files[6]. If the boost header files are located in the system's default, simply run the following command:

[6]The libtools package has been successfully tested against version from 1.48 to 1.54.

```
# ./configure --prefix=/home/harald/aplic/libtools/1.0 \
  && make && make install
```

where `-prefix` indicates the destination folder for this package.
If the boost header files are located elsewhere in the system, run the following command:

```
# ./configure --prefix=/home/harald/aplic/libtools/1.0 \
  --with-boost=/path/to/boost \
  && make && make install
```

U.2.2 Folding package

The most basic configuration for the Folding package honors the following commands:

```
# ./configure --with-libtools=$HOME/aplic/libtools/1.0 \
  --prefix=$HOME/aplic/folding/1.0rc8 && \
  make && make install
```

where `-with-libtools` refers to the location of the libtools package installed in Section U.2.1 and `-prefix` indicates where to install the Folding tool. If the compilation and installation succeed, the contents of the target installation should look like as the contents defined in Section U.1.2.

The Folding tool supports several compilation flags that modify the behavior or enable additional functionalities of the tool. The following list groups the flags according to the behavior they enable.

- `-with-clustering-suite=<DIR>`
  The Folding tool relies on the similarity between the folded instances in order to generate its results. By default, the Folding tool includes two mechanisms to reduce the noise that appear from using instances with significant different behavior. However, this flag allows using the BSC Clustering suite as a third alternative in order to reduce the noise.

- `-with-R=<DIR1>, -with-cube=<DIR2>, -with-clang=<DIR3>`
  Enables the usage of piece-wise linear regressions on top of the strucchange package[7] from the R statistical application[8]. This functionality requires the clang compiler[9] and can generate input files for a modified version of the Cube3 performance analysis package[10].

- `-with-boost=<DIR>`
  This flag lets the Folding to use a given Boost installation package.

- `-enable-gui`
  The results of the Folding tool is a set of gnuplot files that have to explored manually. If this flag is given at the configure step, the Folding package would include a GUI written in Python that helps exploring all the results from the tool.

- `-enable-callstack-analysis`
  This flag enables the call-stack analysis of the segments captured during the measurement step. Enabling this option results in gnuplot files that depict the performance progression collocated with the source code progression.

- `-enable-reference-analysis`
  This flag enables the memory references analysis of the references captured during the

[7] http://cran.r-project.org/web/packages/strucchange/index.html
[8] http://www.r-project.org
[9] http://clang.llvm.org
[10] http://www.scalasca.org/software/cube-3.x/download.html

measurement step (currently, through the `perf` system tool). Enabling this option results in gnuplot files that depict the performance progression collocated with the memory address space and the sampled references.

G

Generate a trace-file for the Folding

This chapter covers the minimum and necessary steps so as to configure Extrae[1] in order to use its resulting trace-files for the Folding process. There are three requirements when monitoring an application with Extrae in order to take the most benefit from the Folding tool. First, it is necessary to enable the sampling mechanism in addition to the instrumentation mechanism (see Section G.1). Second, it is convenient to collect the appropriate performance counters for the underlying processor (see Section G.2). Finally, Extrae needs to capture a segment of the call-stack in order to allow the Folding to provide information regarding the progression of the executed routines. The forthcoming sections provide information on how to enable these functionalities through the XML tags for the Extrae configuration file.

G.1 Enabling the sampling mechanism

Extrae is an instrumentation package that by default collects information from different parallel runtimes, including but not limited to: MPI, OpenMP, pthreads, CUDA and OpenCL (and even combinations of them). Extrae can be configured so that it also uses sampling mechanisms to capture performance metrics on a periodic basis. There are currently two alternatives to enable sampling in Extrae: using alarm signals and using performance counters. For the sake of simplicity, this document only covers the alarm-based sampling. However, if the reader would like to enable the sampling using the performance counters they must look at section 4.9 in the Extrae User's Manual for more details.

Listing G.1
Enable default time-based sampling in Extrae.

```
1 <sampling enabled="yes" type="default" period="50m" variability="10m"/>
```

The XML statements in Listing G.1 need to be included in the Extrae configuration file. These statements indicate Extrae that sampling is enabled (`enabled="yes"`). They also tell Extrae to capture samples every 50 milliseconds (ms) with a random variability of 10 ms, that means that samples will be randomly collected with a periodicity of 50±10 ms. With respect to type, it determines which timer domain is used (see `man 2 setitimer` or `man 3p setitimer` for

[1] Please refer to http://www.bsc.es/computer-sciences/performance-tools/documentation for the latest Extrae User's Guide.

further information on time domains). Available options are: `real` (which is also the `default` value, `virtual` and `prof` (which use the SIGALRM, SIGVTALRM and SIGPROF respectively). The default timing accumulates real time, but only issues samples at master thread. To let all the threads collect samples, the type must be set to either `virtual` or `prof`.

Additionally, the Folding mechanism is able to combine several performance models and generate summarized results that simplify understanding the behavior of the node-level performance. Since these performance models are heavily-tighted with the performance counters available on each processor architecture and family, the following sections provide Extrae XML configuration files ready to use on several architectures. Since each architecture has different characteristics, the user may need to tune the XML presented there to make sure that all the list performance counters are gathered appropriately.

G.2 Collecting the appropriate performance counters

The Folding mechanism provides, among other type of information, the progression of performance metrics along a delimited region through instrumentation points. These performance metrics include the progression of performance counters of every performance counter by default. To generate these kind of reports, Extrae must collect the performance counters during the application execution and this is achieved by defining counter sets into the `<counters>` section of the Extrae configuration file (see Section 4.19 of the Extrae User's guide for more information).

There has been research that has developed some performance models based on performance counters ratios among performance counters in order to ease the analysis of the reports. Each of these performance models aims at providing insight of different aspects of the application and system during the execution. Since the availability of the performance counters changes from processor to processor (even in the same processor family), the following sections describe the performance counters that are meant to be collected in order to calculate these performance models. These sections include the minimal `<counters>` sections to be added in a previously existing Extrae configuration file, but the Folding package also includes full Extrae configuration files in `${FOLDING_HOME}/etc/extrae-configurations`.

G.2.1 Intel Haswell processors

Listing G.2
Counter definition sets for the Extrae configuration file when used on Intel Haswell procesors.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM,PAPI_L2_DCM
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L3_TCM,RESOURCE_STALLS:SB,RESOURCE_STALLS:ROB
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_SR_INS,PAPI_BR_CN,PAPI_BR_UCN
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_BR_MSP,PAPI_LD_INS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,RESOURCE_STALLS,RESOURCE_STALLS:RS
  </set>
</cpu>
```

The listing G.2 indicates Extrae to arrange five performance counter sets with performance counters that are available on Intel Haswell processors. The collection of these performance counters allows the Folding to apply the models contained in the `${FOLDING_HOME}/etc/models/intel-haswell` that include: instruction mix, architecture impact and stall distribution. Unfortunately, the PMU of the Intel Haswell processors do not count neither floating point nor vector instructions.

G.2.2 Intel SandyBridge processors

Listing G.3
Counter definition sets for the Extrae configuration file when used on Intel SandyBridge procesors.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM,PAPI_L2_DCM,PAPI_L3_TCM,PAPI_BR_MSP,
    PAPI_BR_UCN,PAPI_BR_CN
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_VEC_SP,PAPI_LD_INS,PAPI_SR_INS,RESOURCE_STALLS,
    RESOURCE_STALLS:RS,RESOURCE_STALLS:ROB
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_VEC_DP,PAPI_FP_INS,RESOURCE_STALLS:SB,
    RESOURCE_STALLS:LB
  </set>
</cpu>
```

The listing G.4 indicates Extrae to configure five performance counter sets with performance counters that are available on Intel SandyBridge processors. The collection of these performance counters allows the Folding to apply the models contained in the `${FOLDING_HOME}/etc/models/intel-sandybridge` that include: instruction mix, architecture impact and stall distribution.

G.2.3 Intel Nehalem processors

The listing G.3 indicates Extrae to prepare three performance counter sets with performance counters that are available on Intel Nehalem processors. The collection of these performance counters allows the Folding to apply the models contained in the `${FOLDING_HOME}/etc/models/intel-nehalem` that include: instruction mix, architecture impact and stall distribution.

G.2.4 IBM Power8 processors

The listing G.5 indicates Extrae to arrange six performance counter sets with performance counters that are available on IBM Power8 (and similar) processors. The collection of these performance counters allows the Folding to calculate the CPIStack model for the IBM Power8 processor which is contained in `${FOLDING_HOME}/etc/models/ibm-power8`.

G.2.5 IBM Power7 processors

The listing G.6 indicates Extrae to prepare six performance counter sets with performance counters that are available on IBM Power7 (and similar) processors. The collection of these performance counters allows the Folding to calculate the CPIStack model for the IBM Power7 processor which is contained in `${FOLDING_HOME}/etc/models/ibm-power7`.

Listing G.4
Counter definition sets for the Extrae configuration file when used on Intel Nehalem procesors.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM,PAPI_L2_DCM,PAPI_L3_TCM
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_BR_MSP,PAPI_BR_UCN,PAPI_BR_CN,RESOURCE_STALLS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_VEC_DP,PAPI_VEC_SP,PAPI_FP_INS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_LD_INS,PAPI_SR_INS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,RESOURCE_STALLS:LOAD,RESOURCE_STALLS:STORE,
    RESOURCE_STALLS:ROB_FULL,RESOURCE_STALLS:RS_FULL
  </set>
</cpu>
```

Listing G.5
Counter definition sets for the Extrae configuration file when used on IBM Power8 procesors.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL,PM_CMPLU_STALL_DCACHE_MISS,
    PM_CMPLU_STALL_THRD,PM_GRP_CMPL
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_BRU,PM_GCT_NOSLOT_CYC,
    PM_CMPLU_STALL_FXU
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_SCALAR,PM_CMPLU_STALL_LSU
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_STORE,PM_CMPLU_STALL_DMISS_L3MISS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_VECTOR,PM_CMPLU_STALL_REJECT
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_DMISS_L2L3
  </set>
</cpu>
```

G.2.6 IBM Power5 processors

The listing G.7 indicates Extrae to configure six performance counter sets with performance counters that are available on IBM Power5 (and similar) processors. The collection of these performance counters allows the Folding to calculate the CPIStack model for the IBM Power5 processor which is contained in `${FOLDING_HOME}/etc/models/ibm-power5`.

Listing G.6
Counter definition sets for the Extrae configuration file when used on IBM Power7 procesors.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL,PM_CMPLU_STALL_DCACHE_MISS,
    PM_CMPLU_STALL_THRD,PM_GRP_CMPL
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_DFU,PM_CMPLU_STALL_IFU,
    PM_GCT_NOSLOT_CYC
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_FXU,PM_CMPLU_STALL_SCALAR
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_LSU
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_STORE
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_RUN_INST_CMPL,PM_RUN_CYC,PM_CMPLU_STALL_VECTOR
  </set>
</cpu>
```

G.2.7 Other architectures

The previous definitions of counter sets included performance counters that are available on the specific stated machines. Since these performance counters may not be available on all the systems, the package also provides a group of counter sets that are available on a variety of systems. Listing G.8 defines three Extrae counter sets that are available on many systems (caveat here, not all systems provide them). With the use of these counter sets, the Folding can apply the models contained in the `${FOLDING_HOME}/etc/models/basic` that include: instruction mix and architecture impact.

G.3 Capturing the call-stack at sample points

By default, the sampling mechanism captures the performance counters indicated in the `counters` section and the Program Counter interrupted at the sample point. The Folding provides the instantaneous progression of the routines that last at least a minimum given duration. To enable this type of analysis, it is necessary to instruct Extrae to capture a portion of the call-stack during its execution. Listing G.9 shows how to enable the collection of the call-stack at the sample points in the Extrae configuration file. The mandatory lines to capture the call-stack at sample points are lines 1 and 4. Line 1 indicates that this section must be processed and Line 4 tells Extrae to capture levels 1 to 5 from the call-stack (where 1 refers to the level below to the top of the call-stack).

Listing G.7
Counter definition sets for the Extrae configuration file when used on IBM Power5 procesors.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_INST_CMPL,PM_CYC,PM_GCT_EMPTY_CYC,PM_LSU_LMQ_SRQ_EMPTY_CYC,
    PM_HV_CYC,PM_1PLUS_PPC_CMPL,PM_GRP_CMPL,PM_TB_BIT_TRANS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_INST_CMPL,PM_CYC,PM_FLUSH_BR_MPRED,PM_BR_MPRED_TA,
    PM_GCT_EMPTY_IC_MISS,PM_GCT_EMPTY_BR_MPRED,PM_L1_WRITE_CYC
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_INST_CMPL,PM_CYC,PM_LSU_FLUSH,PM_FLUSH_LSU_BR_MPRED,PM_CMPLU_STALL_LSU,
    PM_CMPLU_STALL_ERAT_MISS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_INST_CMPL,PM_CYC,PM_GCT_EMPTY_SRQ_FULL,PM_FXU_FIN,PM_FPU_FIN,
    PM_CMPLU_STALL_FXU,PM_FXU_BUSY,PM_CMPLU_STALL_DIV
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_INST_CMPL,PM_CYC,PM_IOPS_CMPL,PM_CMPLU_STALL_FDIV,PM_FPU_FSQRT,
    PM_CMPLU_STALL_FPU,PM_FPU_FDIV,PM_FPU_FMA
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PM_INST_CMPL,PM_CYC,PM_CMPLU_STALL_OTHER,PM_CMPLU_STALL_DCACHE_MISS,
    PM_LSU_DERAT_MISS,PM_CMPLU_STALL_REJECT,PM_LD_MISS_L1,PM_LD_REF_L1
  </set>
</cpu>
```

Listing G.8
Basic counter definition sets for other processors not stated before.

```
<cpu enabled="yes" starting-set-distribution="cyclic">
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM,PAPI_L2_DCM,PAPI_L3_TCM
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_BR_CN,PAPI_BR_UCN,PAPI_LD_INS,PAPI_SR_INS
  </set>
  <set enabled="yes" domain="all" changeat-time="500000us">
    PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_VEC_SP,PAPI_VEC_DP,PAPI_FP_INS,PAPI_BR_MSP
  </set>
</cpu>
```

Listing G.9
Collect call-stack information at sample points.

```
<callers enabled="yes">
  <mpi enabled="yes">1-3</mpi>
  <pacx enabled="no">1-3</pacx>
  <sampling enabled="yes">1-5</sampling>
</callers>
```

T Tool design

While the end user executes a single command to apply the Folding tool, this command hides two major components that are executed sequentially and all the outputs are generated into a newly created directory with the name of the input trace-file. The first component processes a user-given trace-file that contains instrumented and sampled data and generates a textual file that contains sequences of instances and samples. The second component takes these sequences of instances and samples, then applies the contouring algorithm, any performance model, and the call-stack processing, and, finally, it generates the output results. Both components are grouped together within the `folding.sh` appearing to the user that the Folding simply consists of a single tool.

T.1 First component: trace-file processing

The first component is divided into three phases that are executed one after another with the user-given trace-file and each of these parse the given trace-file and generates another trace-file that will be used in the subsequent phase as depicted in Figure T.1. Each of these phases are built in a similar fashion. They parse the input trace-file and keep in memory information regarding the thread state, and eventually, add information to the output. The phases are:

1. `codeblocks` (found in `src/codeblocks`)
   This phase attributes to each sample information regarding to the loop / code region that it belongs to according to the application source code.

2. `fuse` (found in `src/fuse`)
   This phase compacts the trace-file and ensures that the resulting trace-file is well formed.

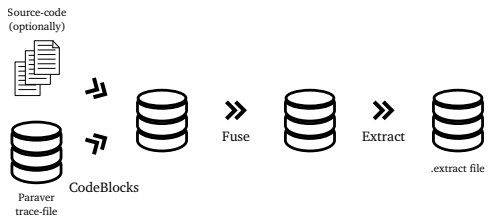


Figure T.1
Data-flow for the first component of the folding.

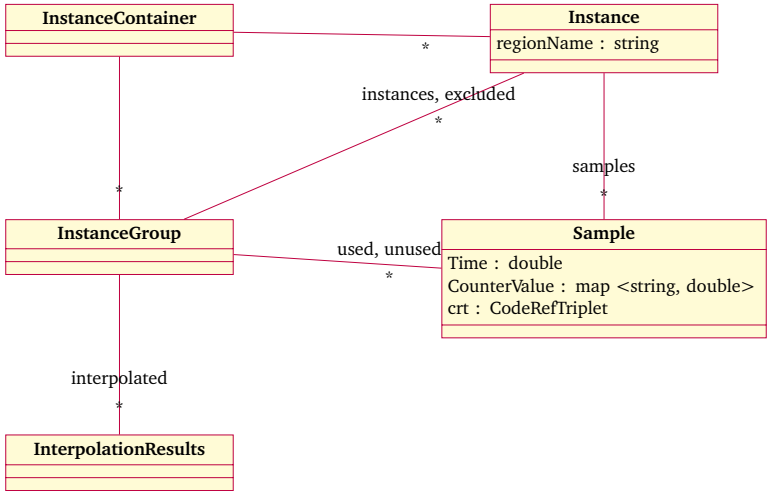


Figure T.2
Main instances and samples related diagram classes.

3. `extract` (found in `src/extract`)
   This is the final phase and extracts information regarding the instances and samples within the trace-file.

The output of this component is a set of files containing information relative to the application. The most notable output is the `.extract` file, which contains the sequence of instances and their samples. For instance, Listing T.1 shows the contents of the `.extract` file generated using the provided example to demonstrate the API facility. This listing contains information regarding one instance of the `FunctionA` region. The instance starts at time-stamp 1,000 ns and lasts 4,500 ns, and it executes up to 2,500 instructions (PAPI_TOT_INS) and takes 5,000 cycles (PAPI_TOT_CYC) to complete. This instance has two samples associated that ocurred at time-stamps 2,000 and 4,000, and each of those provides information regarding the aforementioned performance counters.

Listing T.1
Output example for the extract phase of the Folding mechanism.

```
1 I 2 2 2 FunctionA 1000 4500 2 PAPI_TOT_CYC 5000 PAPI_TOT_INS 2500
2 S 2000 1000 2 PAPI_TOT_CYC 2000 PAPI_TOT_INS 1000 0 0
3 S 4000 3000 2 PAPI_TOT_CYC 4000 PAPI_TOT_INS 2000 2 0 1 2 3 1 3 4 5 0
```

T.2 Second component: applying the folding

The main objective of this component relies on processing the instances and samples extracted and generate the output results. These results include the temporal evolution of the performance counters, any models requested by the user, the source code references and memory references progression, and the results are written in gnuplot and Paraver trace-files. This section gives a summarized view of the Folding work-flow by depicting the most notable class diagrams found in the application source code.

Figure T.2 shows a portion of the classes that are most important within this tool. The classes *Instance* and *Sample* refer to the instances and samples as-is, without any further processing and

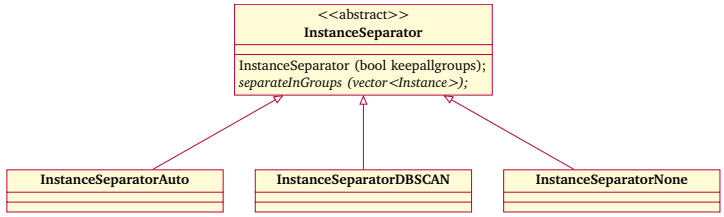


Figure T.3
Instance selection related diagram classes.

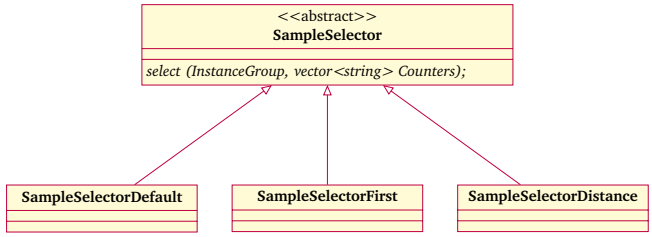


Figure T.4
Sample selector related diagram classes.

as generated by the `extract` tool, in which each *Instance* contains a set of *Sample*, and every *Instance* belongs to an *InstanceContainer*.

After reading every *Instance*, the Folding may apply a clustering algorithm (see Figure T.3) according to the duration of each instance in order to reduce the difference between folded *Instance*. Currently, there are three alternatives regarding the grouping.

- *InstanceSeparatorNone* groups all instances into a single group.
- *InstanceSeparatorAuto* automatically groups the instances according to their duration. The grouping partitions the time-space interval defined by the shortest and longest instances and looks for group of nearby instances.
- *InstanceSeparatorDBSCAN* groups the instances according to a DBSCAN algorithm applied to the duration of the instances. The DBSCAN algorithm groups together instances that are closely packed together (instances with many nearby neighbors) in terms of time and marks as outliers those instances that lie alone in low-density regions. This grouping uses the ClusteringSuite implementation from the BSC performance tools[1].

This grouping begets the *InstanceGroup* objects which contains references to those *Instance* that belong to that particular group. Then, the Folding removes the outliers to each *Instance* within every *InstanceGroup* and store the outliers and the remaining in the *excluded* and *instances* associations, respectively.

Since the complexity of the contouring algorithms depends on the number of points to connect, and therefore the number of samples to fold, the Folding tool supports limiting the number of samples given to these algorithms. Figure T.4 depicts the class diagram of the available *SampleSelector* mechanisms to limit the number of samples.

- *SampleSelectorDefault* the *select* method returns all of the samples within the *InstanceGroup*. This is useful when the user does not impose any limit to the number of samples to be folded.
- *SampleSelectorFirst* receives a threshold (N) in the class constructor. Then, the *select* method tags the first N samples for the processing while the rest are marked as *unused*.

[1] See `http://www.bsc.es/computer-sciences/performance-tools/downloads`.

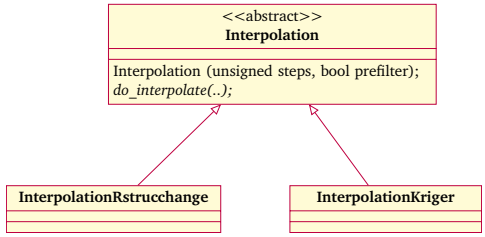


Figure T.5
Performance counter interpolation related diagram classes.

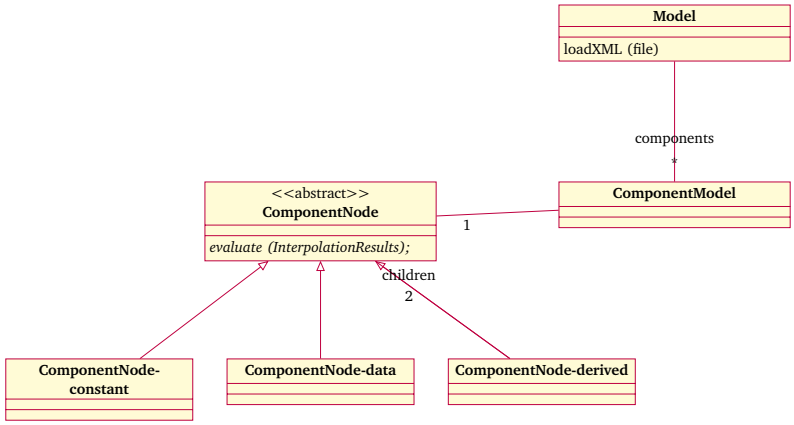


Figure T.6
Performance models related diagram classes.

- *SampleSelectorDistance* receives a threshold (N) in the class constructor. Then, the *select* method tags N samples that are equidistant within the *Instance* duration, while the rest are *unused*.

Then the Folding repeatedly applies the contouring algorithm to the *used* samples among the different *InstanceGroup* objects. The contouring algorithm applies to each performance counter individually, and as of writing this document, there are two approaches that honor the *Interpolation* super-class virtual method (mainly `do_interpolate`):

- *InterpolationKriger* uses the self-provided contouring algorithm based on the Kriging mechanism to implement the `do_interpolate`.
- *InterpolationRstrucchange* employs the strucchange package[2] from the R statistical package[3] to use piece-wise linear regressions to the folded samples. Additionally, this package may benefit from parallel environments if the doParallel package[4] is available on the system.

The interpolation results are stored, per performance counter, into *InterpolationResults* objects that are associated by *InstanceGroup* by the attribute *interpolated* (as depicted in Figure T.2). The *interpolated* attribute is implemented as a hash function indexed by the performance counter, so that the interpolation results can be fetched easily.

The Folding allows defining performance models based on performance counters using XML files (see Listing T.2 for exemplification purposes and `${FOLDING_HOME}/etc/models` for more

[2] http://cran.r-project.org/web/packages/strucchange/index.html
[3] http://www.r-project.org
[4] http://cran.r-project.org/package=doParallel

Listing T.2
Folding example model that generates the L1D and L2D misses per instruction, in addition to the MIPS rate

```
<?xml version='1.0'?>

<model name="sample" title-name="Sample model"
       y1="Ratios" y2="MIPS" y1-stacked="no">

    <component name="l1_dcm_ratio" title-name="L1 DCM" where="y1"
               color="red">
        <operation type='/'>
            <value> PAPI_L1_DCM </value>
            <value> PAPI_TOT_INS </value>
        </operation>
    </component>

    <component name="l2_dcm_ratio" title-name="L2 DCM" where="y1"
               color="blue">
        <operation type='/'>
            <value> PAPI_L2_DCM </value>
            <value> PAPI_TOT_INS </value>
        </operation>
    </component>

    <component name="mips" title-name="MIPS" where="y2" color="black">
        <value> PAPI_TOT_INS </value>
    </component>

</model>
```

detailed examples). Within every XML there are one or several components (in the last Listing these are: `l1_dcm_ratio`, `l2_dcm_ratio` and `mips`) that will be later represented in the resulting gnuplot using the selected colors and Y-axis (left [y1] or right [y2]). Each component refers to the instantaneous value of a certain performance counter (as in the `mips` component), a constant value or the operation (addition, subtraction, multiplication and division) between two other values (as in `l1_dcm_ratio` and `l2_dcm_ratio` components). The Folding implements the performance models based on performance counters employing the diagram classes show in Figure T.6. The XML model files are loaded into the *Model* class and each of them contains multiple components (*ComponentModel*). The *ComponentModel* implements the definition of the component on top of the *ComponentNode* derived sub-classes. These sub-classes allow referencing constant values (*ComponentNode_constant*), interpolated results from a specific performance counter (*ComponentNode_data*) and operation between other two *ComponentNode* objects).

With respect to the analysis of the call-stack, the Folding tool has implemented this analysis through the *CallstackProcessor* related-classes that receives a set of *Sample* objects to explore. Currently, the unique implementation available relies on aligning the call-stacks from the given samples and then exploring the call-stack frames at a given level whether consecutive samples refer to the same routine. If the number of samples surpasses a given threshold, then applies it recursively to the next level until no more levels are available or the number of samples do not surpass the threshold.

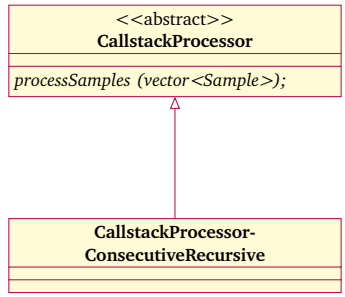


Figure T.7
Call-stack processing related diagram classes.

A API

This section covers the public API available in the Folding package. This API is meant to allow the Folding tool to interact with other performance analysis tools in addition to Extrae.

A.1 Generation of input files for the Folding

A.1.1 Usage example

The directory `${FOLDING_HOME}/share/examples/folding-writer` contains an example that shows how to generate an input file for the folding from a programatically point of view. The example can be compiled using the following command:

```
# cd ${FOLDING_HOME}/share/examples/folding-writer
# make
```

The Listing A.1 shows the example provided in the distributed/installed package. This example demonstrates how to programatically create an `.extract` file for the `interpolate` binary of the Folding package.

Listing A.1
Example of generating an input file for the Folding mechanism.

```
/*****************************************************************************\
 *                        ANALYSIS PERFORMANCE TOOLS                         *
 *                                Folding                                    *
 *             Instrumentation package for parallel applications             *
 *****************************************************************************
 *     ___     This library is free software; you can redistribute it and/or *
 *    /  __         modify it under the terms of the GNU LGPL as published   *
 *   /  /  _____    by the Free Software Foundation; either version 2.1      *
 *  /  /  /     \   of the License, or (at your option) any later version.   *
 * (  (  ( B S C )                                                           *
 *  \  \  \_____/   This library is distributed in hope that it will be      *
 *   \  \__         useful but WITHOUT ANY WARRANTY; without even the        *
 *    \___          implied warranty of MERCHANTABILITY or FITNESS FOR A     *
 *                  PARTICULAR PURPOSE. See the GNU LGPL for more details.   *
 *                                                                           *
 * You should have received a copy of the GNU Lesser General Public License  *
 * along with this library; if not, write to the Free Software Foundation,   *
 * Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA          *
 * The GNU LEsser General Public License is contained in the file COPYING.   *
```

```
20  *                                 ---------                                  *
21  *   Barcelona Supercomputing Center - Centro Nacional de Supercomputacion   *
22  \*****************************************************************************/
23
24  #include "folding-writer.H"
25  #include <fstream>
26
27  using namespace std;
28
29  int main (int argc, char *argv[])
30  {
31      string nameRegion = "FunctionA";
32      unsigned long long startRegion = 1000;
33      unsigned long long durationRegion = 4500;
34
35      /* NOTE:: Counters are given in deltas from their previous read, not as absolute values */
36      map<string, unsigned long long> c1;
37      c1["PAPI_TOT_INS"] = 1000;
38      c1["PAPI_TOT_CYC"] = 2000;
39      map<unsigned, CodeRefTriplet> crt1;
40      Sample *s1 = new Sample (2000, 2000-startRegion, c1, crt1);
41
42      map<string, unsigned long long> c2;
43      c2["PAPI_TOT_INS"] = 1000;
44      c2["PAPI_TOT_CYC"] = 2000;
45      map<unsigned, CodeRefTriplet> crt2;
46      CodeRefTriplet codeinfo_l0 (1,2,3);
47      CodeRefTriplet codeinfo_l1 (3,4,5);
48      crt2[0] = codeinfo_l0;
49      crt2[1] = codeinfo_l1;
50      Sample *s2 = new Sample (4000, 4000-startRegion, c2, crt2);
51
52      /* Last sample typically coincides with end of region -- see durationRegion,
53         Folding::Write won't emit in a S entry */
54      map<string, unsigned long long> c3;
55      c3["PAPI_TOT_INS"] = 500;
56      c3["PAPI_TOT_CYC"] = 1000;
57      map<unsigned, CodeRefTriplet> crt3;
58      Sample *s3 = new Sample (4500, 4500-startRegion, c3, crt3);
59
60      vector<Sample*> vs;
61      vs.push_back (s1);
62      vs.push_back (s2);
63      vs.push_back (s3);
64
65      ofstream f("output.extract");
66      if (f.is_open())
67      {
68          FoldingWriter::Write (f, nameRegion, 1, 1, 1, startRegion,
69            durationRegion, vs);
70          f.close();
71      }
72
73      return 0;
74  }
```

The given example considers that the region `FunctionA` has been identified somehow by the underlying monitoring mechanism, starts at 1,000 ns and lasts 4,500 ns (lines 31-33). Within this period of time, three samples have occurred (*s1-s3*, created in lines 40, 50 and 58, respectively). Samples contain performance counter information and source code references. The performance counter information is given in a relative manner, thus each sample contains the difference from the previous sample (or starting point). For instance, sample *s1* captured information from two performance counters (`PAPI_TOT_INS` and `PAPI_TOT_CYC`) that counted 1,000 and 2,000 events since the start of the region at time-stamp 2,000 ns (lines 36-40). The second sample (*s2*) does not only contain information from performance counters, but also contains a call-stack segment referencing two call-stack frames. The first frame (`codeinfo_l0`) refers to the routine coded as 1, which has source code information coded as 2, and AST-block information coded

as 3 (line 46). The same applies to second frame (`codeinfo_l1`) - (line 47). These frames are mapped into depths 0 and 1 (where 0 refers to the top of the call-stack) in lines 48 and 49, and then the sample is built using the performance counter information and the call-stack information in line 50. Finally, the last sample (*s3*) only accounted 500 and 1,000 events for the `PAPI_TOT_INS` and `PAPI_TOT_CYC` performance counters respectively, but did not capture any source code reference (lines 54-58). This last sample should coincide with the end of the region (`FunctionA`), and may not be necessarily information captured from a sample point, but from an instrumentation point that indicates the end of the region. All these samples are packed together in a STL vector container (lines 60-63), and then the `FoldingWriter::Write` static method dumps all the information from the samples using the given output stream (lines 65-71).

References

[1] L. Adhianto, J. Mellor-Crummey, and N.R. Tallent. **Effectively Presenting Call Path Profiles of Application Performance**. In: *Parallel Processing Workshops (ICPPW), 39th International Conference on*. 2,010, pp. 179–188.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. **HPCTOOLKIT: tools for performance analysis of optimized parallel programs**. In: *Concurrency and Computation: Practice and Experience* 22.6 (2,010), pp. 685–701. ISSN: 1532-0634.

[3] Xavier Aguilar, Karl Fürlinger, and Erwin Laure. **MPI Trace Compression Using Event Flow Graphs**. In: *Euro-Par Parallel Processing*. Vol. 8632. Lecture Notes in Computer Science. Springer International Publishing, 2,014, pp. 1–12. ISBN: 978-3-319-09872-2.

[4] Pedro Alonso, Rosa M. Badia, Jesús Labarta, Maria Barreda, Manuel F. Dolz, Rafael Mayo, Enrique S. Quintana-Ortí, and Ruymán Reyes. **Tools for Power-Energy Modelling and Analysis of Parallel Scientific Applications**. In: *Parallel Processing (ICPP), 41st International Conference on*. 2,012, pp. 420–429.

[5] Gene M. Amdahl. **Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities**. In: *Proceedings of the Spring Joint Computer Conference*. AFIPS'67 (Spring). Atlantic City, New Jersey: ACM, 1,967, pp. 483–485.

[6] Rocco Aversa, Beniamino Di Martino, Nicola Mazzocca, and Umberto Villano. **Reducing Parallel Program Simulation Complexity by Static Analysis**. In: *J. Supercomput.* 17.3 (2,000), pp. 299–310. ISSN: 0920-8542.

[7] M. Avila, Arnau Folch, G. Houzeaux, B. Eguzkitza, L. Prieto, and D. Cabezón. **Onshore Wind Farm Modelling**. Tech. rep. 2,012.

[8] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. **Online performance analysis by statistical sampling of microprocessor performance counters**. In: *ICS'05: Proceedings of the 19th annual international conference on Supercomputing*. ACM, 2,005, pp. 101–110.

[9] Rosa M. Badia, Francesc Escalé, Edgar Gabriel, Judit Gimenez, Rainer Keller, Jesús Labarta, and Matthias S. Müller. **Performance Prediction in a Grid Environment**. In: *Grid Computing, First European Across Grids Conference*. 2,003, pp. 257–264.

[10] D. H. Bailey et al. **The NAS Parallel Benchmarks**. In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. Supercomputing'91. Albuquerque, New Mexico, USA: ACM, 1,991, pp. 158–165. ISBN: 0-89791-459-7.

[11] Benjamin R Barsdell, David G Barnes, and Christopher J Fluke. **Advanced architectures for astrophysical supercomputing**. In: *arXiv preprint arXiv:1001.2048* (2,010).

[12] Hanefi Bayraktar and F.Sezer. Turalioglu. **A Kriging-based approach for locating a sampling site—in the assessment of air quality**. In: *Stochastic Environmental Research and Risk Assessment* 19.4 (2,005), pp. 301–305. ISSN: 1436-3240.

[13] D. Bedard, Min Yeol Lim, R. Fowler, and A. Porterfield. **PowerMon: Fine-grained and integrated power monitoring for commodity computer systems**. In: *IEEE SoutheastCon, Proceedings of the*. 2,010, pp. 479–484.

[14] Paul Bédaride, Augustin Degomme, Stéphane Genaud, Arnaud Legrand, George S. Markomanolis, Martin Quinson, Mark Stillwell, Frédéric Suter, and Brice Videau. **Toward Better Simulation of MPI Applications on Ethernet/TCP Networks**. In: *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation - 4th International Workshop*. 2,013, pp. 158–181.

[15] Andrew R. Bernat and Barton P. Miller. **Incremental Call-path Profiling: Research Articles**. In: *Concurrency and Computation: Practice and Experience* 19.11 (2,007), pp. 1533–1547. ISSN: 1532-0626.

[16] Ramon Bertran, Marc González, Xavier Martorell, Nacho Navarro, and Eduard Ayguadé. **Decomposable and Responsive Power Models for Multicore Processors Using Performance Counters**. In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. Tsukuba, Ibaraki, Japan: ACM, 2,010, pp. 147–158. ISBN: 978-1-4503-0018-6.

[17] K. Beyls and E.H. D'Hollander. **Refactoring for Data Locality**. In: *Computer* 42.2 (2,009), pp. 62–71. ISSN: 0018-9162.

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. **Cilk: An Efficient Multithreaded Runtime System**. In: *Journal of Parallel and Distributed Computing* 37.1 (1,996), pp. 55 –69. ISSN: 0743-7315.

[19] David Brooks, Vivek Tiwari, and Margaret Martonosi. **Wattch: a framework for architectural-level power analysis and optimizations**. In: *Proceedings of the 27th annual international symposium on Computer architecture*. ISCA '00. Vancouver, British Columbia, Canada: ACM, 2,000, pp. 83–94. ISBN: 1-58113-232-8.

[20] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. **A Portable Programming Interface for Performance Evaluation on Modern Processors**. In: *Int. J. High Perform. Comput. Appl.* 14.3 (2,000).
http://icl.cs.utk.edu/papi
Last accessed March, 2,015., pp. 189–204. ISSN: 1094-3420.

[21] Derek Bruening. **Efficient, Transparent, and Comprehensive Runtime Code Manipulation.** Thesis dissertation.
http://www.dynamorio.org
Last accessed March, 2,015. 2,004.

[22] H. Brunst, W.E. Nagel, and A.D. Malony. **A distributed performance analysis architecture for clusters**. In: *Proceedings of the IEEE International Conference on Cluster Computing*. 2,003, pp. 73–81.

[23] BSC Programming models team. **Mercurium User Manual**.
https://pm.bsc.es/projects/mcxx/wiki/UserManual
Last accessed March, 2,015.

[24] Bryan Buck and Jeffrey K. Hollingsworth. **An API for Runtime Code Patching**. In: *Int. J. High Perform. Comput. Appl.* 14.4 (2,000).
http://www.dyninst.org
Last accessed March, 2,015., pp. 317–329. ISSN: 1094-3420.

[25] Doug Burger and Todd M. Austin. **The SimpleScalar Tool Set, Version 2.0**. In: *SIGARCH Comput. Archit. News* 25.3 (1,997), pp. 13–25. ISSN: 0163-5964.

[26] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. **PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications**. In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'10. Washington, DC, USA: IEEE Computer Society, 2,010, pp. 1–11. ISBN: 978-1-4244-7559-9.

[27] Peter Calingaert. **System Performance Evaluation: Survey and Appraisal**. In: *Commun. ACM* 10.1 (1,967), pp. 12–18. ISSN: 0001-0782.

[28] H. N. Cantrell and A. L. Ellison. **Multiprogramming System Performance Measurement and Analysis**. In: *Proceedings of the Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1,968, pp. 213–221.

[29] T.E. Carlson, W. Heirman, and L. Eeckhout. **Sampled simulation of multi-threaded applications**. In: *Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium on*. 2,013, pp. 2–12.

[30] M. Casas and G. Bronevetsky. **Active Measurement of Memory Resource Consumption**. In: *Parallel and Distributed Processing Symposium, IEEE 28th International*. 2,014, pp. 995–1004.

[31] M. Casas and G. Bronevetsky. **Active Measurement of the Impact of Network Switch Utilization on Application Performance**. In: *Parallel and Distributed Processing Symposium, IEEE 28th International*. 2,014, pp. 165–174.

[32] Marc Casas, Rosa M. Badia, and Jesús Labarta. **Automatic Phase Detection of MPI Applications**. In: *Parallel Computing: Architectures, Algorithms and Applications, ParCo, Forschungszentrum Jülich and RWTH Aachen University, Germany*. 2,007, pp. 129–136.

[33] IBM Knowledge Center. **CPI events and metrics**.
www-01.ibm.com/support/knowledgecenter/linuxonibm/liaal/iplsdkcpievents.htm
Last accessed May, 2,015.

[34] Center for Information Services and High Performance Computing (ZIH) of TU Dresden. **Vampir tutorial: Performance data visualization**.
http://www.vampir.eu/tutorial/manual/performance_data_visualization
Last accessed March, 2,015.

[35] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. **Evaluation of the Intel Core i7 Turbo Boost Feature**. In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IISWC'09. IEEE Computer Society, 2,009, pp. 188–197. ISBN: 978-1-4244-5156-2.

[36] Thomas Chen, Ram Raghavan, Jason N Dale, and Eiji Iwata. **Cell broadband engine architecture and its first implementation—a performance view**. In: *IBM Journal of Research and Development* 51.5 (2,007), pp. 559–572.

[37] Intel Corporation. **Intel 64 and IA-32 Architectures Software Developer's Manual**. Vol. Volume 3B: System Programming Guide, Part 2. 2,015.

[38] Oracle Corporation. **Java Remote Method Invocations**.
http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
Last accessed March, 2,015.

[39] Oracle Corporation. **Oracle Solaris Studio**.
http://www.oracle.com//us/products/servers-storage/solaris/studio/overview/index.html
Last accessed, March 2,015.

[40] **Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual SR-3108 9.1**.
http://docs.cray.com/books/SR-3108_9.1/html-SR-3108_9.1/fqwqfdcsmg.html
Last accessed March, 2,015.

[41] Inc. Cray Research. **CRAY Y-MP Computer Systems Functional Description Manual**. http://bitsavers.informatik.uni-stuttgart.de/pdf/cray/CRAY_Y-MP/HR-04001-0C_Cray_Y-MP_Computer_Systems_Functional_Description_Jun90.pdf Last accessed March, 2,015. 1,989.

[42] L. Dagum and R. Menon. **OpenMP: an industry standard API for shared-memory programming**. In: *IEEE Computational Science and Engineering* 5.1 (1,998), pp. 46–55. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=660313.

[43] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. **RAPL: memory power estimation and capping**. In: *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. ISLPED '10. Austin, Texas, USA: ACM, 2,010, pp. 189–194. ISBN: 978-1-4503-0146-6.

[44] Wolfgang E. Denzel, Jian Li, Peter Walker, and Yuho Jin. **A Framework for End-to-end Simulation of High-performance Computing Systems**. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. Simutools '08. 2,008, 21:1–21:10. ISBN: 978-963-9799-20-2.

[45] Luiz DeRose and Felix Wolf. **CATCH — A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications**. In: *Euro-Par Parallel Processing*. Ed. by Burkhard Monien and Rainer Feldmann. Vol. 2400. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2,002, pp. 167–176. ISBN: 978-3-540-44049-9.

[46] Olivier Desjardins, Guillaume Blanquart, Guillaume Balarac, and Heinz Pitsch. **High order conservative finite difference scheme for variable density low Mach number turbulent flows**. In: *Journal of Computational Physics* 227.15 (2,008), pp. 7125 –7159. ISSN: 0021-9991. URL: http://www.sciencedirect.com/science/article/pii/S0021999108001666.

[47] **Discrete analysis**. Tech. rep. http://www.scalalife.eu/content/discrete-2 Last accessed June, 2,014. ScalaLife Competence Center, 2,011.

[48] Romain Dolbeau, Stéphane Bihan, and François Bodin. **HMPP™: A hybrid multi-core parallel programming environment**. In: *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*. 2,007.

[49] Jack Dongarra et al. **The International Exascale Software Project Roadmap**. In: *Int. J. High Perform. Comput. Appl.* 25.1 (2,011). http://hpc.sagepub.com/content/25/1/3 Last accessed March, 2,015., pp. 3–60. ISSN: 1094-3420.

[50] Jack J. Dongarra. **Performance of various computers using standard linear equations software**. In: *SIGARCH Comput. Archit. News* 20.3 (1,992), pp. 22–44. ISSN: 0163-5964.

[51] Jack J. Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. **A Proposal For A User-Level, Message Passing Interface In A Distributed Memory Environment**. 1,993.

[52] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. **SeqAn an efficient, generic C++ library for sequence analysis**. In: *BMC bioinformatics* 9.1 (2,008), p. 11.

[53] Damien Dosimont, Robin Lamarche-Perrin, Lucas Mello Schnorr, Guillaume Huard, and Jean-Marc Vincent. **A spatiotemporal data aggregation technique for performance analysis of large-scale execution traces**. In: *IEEE International Conference on Cluster Computing, CLUSTER*. 2,014, pp. 149–157.

[54] Damien Dosimont, Generoso Pagano, Guillaume Huard, Vania Marangozova-Martin, and Jean-Marc Vincent. **Efficient analysis methodology for huge application traces**. In: *International Conference on High Performance Computing & Simulation, HPCS*. 2,014, pp. 951–958.

[55] P. Drongowski, Lei Yu, F. Swehosky, S. Suthikulpanit, and R. Richter. **Incorporating Instruction-Based Sampling into AMD CodeAnalyst**. In: *Performance Analysis of Systems Software (ISPASS), IEEE International Symposium on*. 2,010, pp. 119–120.

[56] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. **OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures**. In: *Parallel Processing Letters* 21.2 (2,011), pp. 173–193.

[57] Marc Duranton, David Black-Shaffer, Sami Yehia, and Koen De Bosschere. **The HIPEAC vision 2011/2012**.
http://www.hipeac.net/system/files/hipeac-roadmap2011.pdf
Last accessed April, 2,014. 2,012.

[58] Lieven Eeckhout, Smaïl Niar, and Koen De Bosschere. **Optimal sample length for efficient cache simulation**. In: *Journal of Systems Architecture* 51.9 (2,005), pp. 513–525.

[59] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. **OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis**. In: *OpenMP in the Era of Low Power Devices and Accelerators*. Vol. 8122. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2,013, pp. 171–185. ISBN: 978-3-642-40697-3.

[60] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. **Cache Pirating: Measuring the Curse of the Shared Cache**. In: *Parallel Processing (ICPP), International Conference on*. 2,011, pp. 165–175.

[61] Tarek A El-Ghazawi, William W Carlson, and Jesse M Draper. **UPC Language Specifications v1.1.1**. Tech. rep. 3. 2,003, p. 1.

[62] Brad Elkin and Venkat Indukuru. **Commonly Used Metrics for Performance Analysis POWER7**.
https://www.power.org/documentation/commonly-used-metrics-for-performance-analysis
Last accessed March, 2,015. 2,011.

[63] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. **A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise**. In: *KDD*. 1,996, pp. 226–231.

[64] Maja Etinski, Julita Corbalan, Jesús Labarta, Mateo Valero, and Alex Veidenbaum. **Power-aware load balancing of large scale MPI applications**. In: *Proceedings of the IEEE International Symposium on Parallel&Distributed Processing*. IPDPS'09. Washington, DC, USA: IEEE Computer Society, 2,009, pp. 1–8. ISBN: 978-1-4244-3751-1.

[65] Alan Eustace and Amitabh Srivastava. **ATOM: A Flexible Interface for Building High Performance Program Analysis Tools**. In: *Proceedings of the USENIX Technical Conference Proceedings*. TCON'95. New Orleans, Louisiana: USENIX Association, 1,995, pp. 25–25.

[66] **Extrae user guide**.
http://www.bsc.es/paraver
Last accessed March, 2,015. Barcelona Supercomputing Center. 2,014.

[67] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. **A performance counter architecture for computing accurate CPI components**. In: *SIGARCH Comput. Archit. News* 34 (2,006), pp. 175–184. ISSN: 0163-5964.

[68] Ivan Flores. **Lookahead control in the IBM System 370 model 165**. In: *Computer* 7.11 (1,974), pp. 24–38. ISSN: 0018-9162.

[69] M. Flynn. **Some Computer Organizations and Their Effectiveness**. In: *Computers, IEEE Transactions on* C-21.9 (1,972), pp. 948–960. ISSN: 0018-9340.

[70] Message-Passing Interface Forum. **MPI: A Message-Passing Interface Standard**. In:
http://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps
Last accessed March, 2,015. 1,994. Chap. 8: Profiling Interface, pp. 198–203.

[71] Free Software Foundation. **GNU Binutils**.
http://www.gnu.org/software/binutils
Last accessed March, 2,015.

[72] Free Software Foundation and the GCC team. **GCC, the GNU Compiler Collection**.
http://gcc.gnu.org
Last accessed March, 2,015.

[73] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. **Low-overhead Call Path Profiling of Unmodified, Optimized Code**. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: ACM, 2,005, pp. 81–90. ISBN: 1-59593-167-8.

[74] Nathan Froyd et al. **Low-overhead call path profiling of unmodified, optimized code**. In: *ICS05*. Cambridge, Massachusetts: ACM, 2,005, pp. 81–90. ISBN: 1-59593-167-8.

[75] Karl Fürlinger and Shirley Moore. **Continuous Runtime Profiling of OpenMP Applications**. In: *Parallel Computing: Architectures, Algorithms and Applications, ParCo Forschungszentrum Jülich and RWTH Aachen University*. 2,007, pp. 677–684.

[76] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. **PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications**. In: *IEEE Transactions on Parallel and Distributed Systems* 99.RapidPosts (2,009), pp. 658–671. ISSN: 1045-9219.

[77] M. Geimer, F. Wolf, B. Wylie, and B. Mohr. **A scalable tool architecture for diagnosing wait states in massively parallel applications**. In: *Parallel computing* 35 (2,009), pp. 375 –388. ISSN: 0167-8191. URL: http://juser.fz-juelich.de/record/6609.

[78] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. **The Scalasca Performance Toolset Architecture**. In: *Concurrency and Computation: Practice and Experience* 22.6 (2,010), pp. 702–719. ISSN: 1532-0626.

[79] L. Genovese et al. **Daubechies wavelets as a basis set for density functional pseudopotential calculations**. In: *J. Chem. Phys.* 129.1 (2,008), p. 014109. arXiv:0804.2583 [cond-mat.mtrl-sci].

[80] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. **Periscope: Advanced Techniques for Performance Analysis**. In: *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo, Department of Computer Architecture, University of Malaga, Spain*. 2,005, pp. 15–26.

[81] Marc-Oliver Gewaltig and Markus Diesmann. **NEST (NEural Simulation Tool)**. In: *Scholarpedia* 2.4 (2,007), p. 1430.

[82] Gibson. **Computer Structures: Reading and Examples**. Ed. by McGraw-Hill Book Company. Published reference unknown cited by C. Gordon Bell and Allen Newell.
http://research.microsoft.com/en-us/um/people/gbell/cgb%20files/computer%20structures%20readings%20and%20examples%201971.pdf
Last accessed June, 2,014. 1,971.

[83] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. **Dissecting On-Node Memory Access Performance: A Semantic Approach**. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2,014, pp. 166–176.

[84] Forschungszentrum Juelich Gmbh and Paul Gibbon. **PEPC: Pretty Efficient Parallel Coulomb-solver**. In: (2,003).

[85] J. González, J. Giménez, and J. Labarta. **Automatic detection of parallel applications computation phases**. In: *Parallel Distributed Processing, IEEE International Symposium on*. 2,009, pp. 1–11.

[86] Juan González, Judit Giménez, and Jesús Labarta. **Performance Data Extrapolation in Parallel Codes**. In: *ICPADS '10: Proceedings of the 16th International Conference on Parallel and Distributed Systems*. Shanghai, 2,010.

[87] Pierre Goovaerts. **Geostatistics for natural resources evaluation**. Oxford university press, 1,997.

[88] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. **Gprof: A call graph execution profiler**. In: *SIGPLAN'82: Proceedings of the SIGPLAN symposium on Compiler construction*. Boston, Massachusetts, United States: ACM, 1,982, pp. 120–126. ISBN: 0-89791-074-5.

[89] Robert J. Hall. **Call Path Profiling**. In: *Proceedings of the 14th International Conference on Software Engineering*. ICSE '92. Melbourne, Australia: ACM, 1,992, pp. 296–306. ISBN: 0-89791-504-6.

[90] P. Havlak and K. Kennedy. **An implementation of interprocedural bounded regular section analysis**. In: *Parallel and Distributed Systems, IEEE Transactions on* 2.3 (1,991), pp. 350–360. ISSN: 1045-9219.

[91] John L. Henning. **SPEC CPU2000: measuring CPU performance in the New Millennium**. In: *Computer* 33.7 (2,000), pp. 28–35. ISSN: 0018-9162.

[92] John L. Henning. **SPEC CPU2006 benchmark descriptions**. In: *SIGARCH Comput. Archit. News* 34.4 (2,006), pp. 1–17. ISSN: 0163-5964.

[93] Forrest Hoffman et al. **Terrestrial biogeochemistry in the community climate system model (CCSM)**. In: *Journal of Physics: Conference Series* 46.1 (2,006), p. 363. URL: http://stacks.iop.org/1742-6596/46/i=1/a=051.

[94] Jeffrey K. Hollingsworth and Barton P. Miller. **Dynamic Control of Performance Monitoring on Large Scale Parallel Systems**. In: *International Conference on Supercomputing*. 1,993, pp. 185–194. DOI: 10.1145/165939.165969. URL: http://doi.acm.org/10.1145/165939.165969.

[95] Kevin A. Huck and Allen D. Malony. **PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing**. In: *SC'05: Proceedings of the ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2,005, p. 41. ISBN: 1-59593-061-2.

[96] Kevin A. Huck, Allen D. Malony, and Alan Morris. **Design and Implementation of a Parallel Performance Data Management Framework**. In: *Proceedings of the International Conference on Parallel Processing*. ICPP'05. Washington, DC, USA: IEEE Computer Society, 2,005, pp. 473–482. ISBN: 0-7695-2380-3.

[97] Kevin A. Huck, Oscar Hernández, Van Bui, Sunita Chandrasekaran, Barbara Chapman, Allen D. Malony, Lois Curfman McInnes, and Boyana Norris. **Capturing performance knowledge for automated analysis**. In: *Proceedings of the ACM/IEEE conference on Supercomputing*. SC'08. Austin, Texas: IEEE Press, 2,008, 49:1–49:10. ISBN: 978-1-4244-2835-9.

[98] James W. Hurrell et al. **The community earth system model: a framework for collaborative research**. In: *Bulletin of the American Meteorological Society* 94.9 (2,013), pp. 1339–1360.

[99] **Hydrodynamics Challenge Problem**. Tech. rep.
https://codesign.llnl.gov/pdfs/LULESH2.0_Changes.pdf
Last accessed March, 2,015. Lawerence Livermore National Laboratory, 2,011.

[100] IBM. **IBM XL C/C++ Compilers**.
https://www.ibm.com/developerworks/rational/products/ccpcompiler
Last accessed March, 2,015.

[101] IBM Corporation. **Xprofiler main window**.
http://publib.boulder.ibm.com/infocenter/aix/v7r1/index.jsp?topic=com.ibm.aix.prftools/Fdoc/prftools/idprftools_profiling_xwin_disp.htm
Last accessed May, 2,014.

[102] Intel. **Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture**.
http://download.intel.com/support/processors/pentiumii/sb/24319002.pdf
Last accessed April, 2,014. 1,999.

[103] Intel. **Intel® Architecture Instruction Set Extensions Programming Reference**.
http://download-software.intel.com/sites/default/files/managed/68/8b/319433-019.pdf
Last accessed April, 2,014. 2,014.

[104] Intel Corporation. **Intel® Compilers**.
https://software.intel.com/en-us/intel-compilers
Last accessed March, 2,015.

[105] Edward H Isaaks and R Mohan Srivastava. **Applied geostatistics**. Oxford University Press, 1,989.

[106] Marty Itzkowitz et al. **Memory Profiling using Hardware Counters**. In: *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 2,003, p. 17. ISBN: 1-58113-695-1.

[107] Julien Jaeger, Peter Philippen, Eric Petit, Andres Charif Rubial, Christian Rössel, William Jalby, and Bernd Mohr. **Binary Instrumentation for Scalable Performance Measurement of OpenMP Applications**. In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Vol. 25. Advances in Parallel Computing.
http://juser.fz-juelich.de/record/152042
Last accessed March, 2,015. International Conference on Parallel Computing, Munich (Germany). 2,014, pp. 783 –792. ISBN: 978-1-61499-380-3.

[108] S. Jarp. **A methodology for using the Itanium-2 performance counters for bottleneck analysis**. Tech. rep.
http://sc.tamu.edu/whitepapers/altix/Performance_counters_final.pdf
Last accessed March, 2,015. HP Labs, 2,002.

[109] James Jeffers and James Reinders. **Intel Xeon Phi Coprocessor High Performance Programming**. Morgan Kaufmann, 2,013. ISBN: 0124104142.

[110] John Levon and others. **OProfile Manual**.
http://oprofile.sourceforge.net/doc/index.html
Last accessed March, 2,015.

[111] John Levon and others. **perf Wiki**.
https://perf.wiki.kernel.org/index.php/Main_Page
Last accessed March, 2,015.

[112] Rainer Keller, George Bosilca, Graham Fagg, Michael Resch, and JackJ. Dongarra. **Implementation and Usage of the PERUSE-Interface in Open MPI**. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Bernd Mohr, JesperLarsson Träff, Joachim Worringen, and Jack Dongarra. Vol. 4192. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2,006, pp. 347–355. ISBN: 978-3-540-39110-4.

[113] Jacques Chassin de Kergommeaux, Benhur de Oliveira Stein, and Pierre-Eric Bernard. **Pajé, an interactive visualization tool for tuning multi-threaded parallel applications**. In: *Parallel Computing* 26.10 (2,000), pp. 1253–1274.

[114] Khronos OpenCL Working Group. **The OpenCL specification**. Ed. by Aaftab Munshi et al.
https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf
Last accessed March, 2,015. 2,009.

[115] Ron Kikinis, SK Warfield, and Carl-Fredrik Westin. **High performance computing (HPC) in medical image analysis (MIA) at the surgical planning laboratory (SPL)**. In: *Proceedings of the Third High Performance Computing Asia Conference and Exhibition. Singapore, Singapore: IEEE*. Citeseer. 1,998.

[116] A. Knüpfer and W.E. Nagel. **Construction and compression of complete call graphs for post-mortem program trace analysis**. In: *Parallel Processing, International Conference on*. 2,005.

[117] Jonas HM Korndorfer and Lucas Mello Schnorr. **First efforts for the Parallelization of PajeNG**.

[118] Jesús Labarta. **New Analysis Techniques in the CEPBA-Tools Environment**. In: *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2,010, pp. 125–143. ISBN: 978-3-642-11260-7.

[119] Jesús Labarta, Sergi Girona, and Toni Cortés. **Analyzing scheduling policies using Dimemas**. In: *Parallel Computing* 23.1-2 (1,997). Environment and tools for parallel scientific computing, pp. 23 –34. ISSN: 0167-8191.

[120] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortés, and Luis Gregoris. **DiP: A Parallel Program Development Environment**. In: *Euro-Par, Vol. II*. 1,996, pp. 665–674.

[121] Gary Lakner, I-Hsin Chung, Guojing Cong, Scott Fadden, Nicholas Goracke, David Klepacki, Jeffrey Lien, Christoph Pospiech, Seetharami R Seelam, and Hui-Fang Wen. **IBM System Blue Gene Solution: Performance Analysis Tools**. Tech. rep. 2,008.

[122] James R. Larus. **Whole Program Paths**. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'99. Atlanta, Georgia, USA: ACM, 1,999, pp. 259–269. ISBN: 1-58113-094-5.

[123] Timo Lassmann, Oliver Frings, and Erik LL Sonnhammer. **Kalign2: high-performance multiple alignment of protein and nucleotide sequences allowing external features**. In: *Nucleic acids research* 37.3 (2,009), pp. 858–865.

[124] Etienne Le Sueur and Gernot Heiser. **Dynamic voltage and frequency scaling: the laws of diminishing returns**. In: *Proceedings of the International Conference on Power Aware Computing and Systems*. HotPower'10. Vancouver, BC, Canada: USENIX Association, 2,010, pp. 1–8.

[125] J.K.F. Lee and A.J. Smith. **Branch Prediction Strategies and Branch Target Buffer Design**. In: *Computer* 1 (1,984), pp. 6–22. ISSN: 0018-9162.

[126] Charles E. Leiserson. **The Cilk++ concurrency platform**. In: *The Journal of Supercomputing* 51.3 (2,010), pp. 244–257. ISSN: 0920-8542.

[127] Bil Lewis and Daniel J. Berg. **Multithreaded Programming with Pthreads**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1,998. ISBN: 0-13-680729-1.

[128] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. **OpenUH: an optimizing, portable OpenMP compiler**. In: *Concurrency and Computation: Practice and Experience* 19.18 (2,007), pp. 2317–2332. ISSN: 1532-0634.

[129] Z. Lin et al. **Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations**. In: *Science* 281.5384 (1,998), pp. 1835–1837.

[130] Xu Liu and John M. Mellor-Crummey. **A data-centric profiler for parallel programs**. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*. 2,013, 28:1–28:12.

[131] Germán Llort, Harald Servat, Juan González, Judit Giménez, and Jesús Labarta. **On the usefulness of object tracking techniques in performance analysis**. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA*. 2,013, p. 29.

[132] Germán Llort, Marc Casas, Harald Servat, Kevin A. Huck, Judit Giménez, and Jesús Labarta. **Trace Spectral Analysis toward Dynamic Levels of Detail**. In: *IEEE 17th International Conference on Parallel and Distributed Systems, ICPADS, Tainan, Taiwan*. 2,011, pp. 332–339.

[133] LLVM Project and University of Illinois at Urbana-Champaing. **clang: a C language family frontend for LLVM**.
http://clang.llvm.org
Last accessed March, 2,015.

[134] Gabriel H. Loh. **3D-Stacked Memory Architectures for Multi-core Processors**. In: *Computer Architecture, 35th International Symposium on*. 2,008, pp. 453–464.

[135] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. **PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation**. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'05. Chicago, IL, USA: ACM, 2,005, pp. 190–200. ISBN: 1-59593-056-6.

[136] Gurvan Madec. **NEMO ocean engine**. Tech. rep. http://www.nemo-ocean.eu/content/download/5302/31828/file/NEMO_book.pdf Last accessed March, 2,015. 2,008.

[137] Allen Malony, Sameer Shende, Wyatt Spear, CheeWai Lee, and Scott Biersdorff. **Advances in the TAU Performance System**. In: *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2,012, pp. 119–130. ISBN: 978-3-642-31475-9.

[138] **MareNostrum3 system architecture**. http://www.bsc.es/marenostrum-support-services/mn3 Last accessed, March 2,015.

[139] B. Maron, T. Chen, D. Vianney, B. Olszewski, S. Kunkel, and A. Mericas. **Workload characterization for the design of future servers**. In: *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 2,005, pp. 129–136. ISBN: 0-7803-9461-5.

[140] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. **MemSpy: Analyzing Memory System Bottlenecks in Programs**. In: *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS'92/PERFORMANCE'92. Newport, Rhode Island, USA: ACM, 1,992, pp. 1–12. ISBN: 0-89791-507-0.

[141] John D. McCalpin. **Memory Bandwidth and Machine Balance in Current High Performance Computers**. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1,995), pp. 19–25.

[142] Sally A. McKee. **Reflections on the Memory Wall**. In: *Proceedings of the 1st Conference on Computing Frontiers*. CF '04. Ischia, Italy: ACM, 2,004, pp. 162–. ISBN: 1-58113-741-9.

[143] Cameron McNairy and Don Soltis. **Itanium 2 Processor Microarchitecture**. In: *IEEE Micro* 23.2 (2,003), pp. 44–55. ISSN: 0272-1732.

[144] Arnaldo Carvalho de Melo. **The new linux "perf" tools**. In: *Linux Kongress*. 2,010.

[145] Alex Mericas. **Performance Characteristics of the POWER8 Processor**. http://www.setphaserstostun.org/hc26/HC26-12-day2-epub/HC26.12-8-Big-Iron-Servers-epub/HC26.12.810-POWER8-Mericas-IBM.pdf Last accessed March, 2,015.

[146] Message Passing Interface Forum. **MPI: A Message-Passing Interface Standard version 3.0**. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf Last accessed March, 2,015. 2,012.

[147] Dieteran Mey et al. **Score-P: A Unified Performance Measurement System for Petascale Applications**. In: *Competence in High Performance Computing*. Springer Berlin Heidelberg, 2,012, pp. 85–97. ISBN: 978-3-642-24024-9.

[148] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. **The Weather Reseach and Forecast Model: Software Architecture and Performance**. In: *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*. 2,004.

[149] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. **The Paradyn Parallel Performance Measurement Tool**. In: *IEEE Computer* 28.11 (1,995), pp. 37–46.

[150] Mimica, P., Giannios, D., and Aloy, M. A. **Deceleration of arbitrarily magnetized GRB ejecta: the complete evolution**. In: *Astronomy & Astrophysics* 494.3 (2,009), pp. 879–890.

[151] Cyriel Minkenberg, Wolfgang Denzel, German Rodriguez, and Robert Birke. **End-to-End Modeling and Simulation of High- Performance Computing Systems**. In: *Use Cases of Discrete Event Simulation*. Springer Berlin Heidelberg, 2,012, pp. 201–240. ISBN: 978-3-642-28776-3.

[152] Shirley V. Moore. **A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware**. In: *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*. London, UK: Springer-Verlag, 2,002, pp. 904–912. ISBN: 3-540-43593-X.

[153] Randall Morck et al. **Management ownership and market valuation: An empirical analysis**. In: *Journal of Financial Economics* 20 (1,988), pp. 293 –315. ISSN: 0304-405X.

[154] A. Morris, A.D. Malony, S. Shende, and K. Huck. **Design and Implementation of a Hybrid Parallel Performance Measurement System**. In: *Parallel Processing (ICPP), 39th International Conference on*. 2,010, pp. 492–501.

[155] Trevor Mudge. **Power: A First-Class Architectural Design Constraint**. In: *Computer* 34.4 (2,001), pp. 52–58. ISSN: 0018-9162.

[156] Jan Mußler, Daniel Lorenz, and Felix Wolf. **Reducing the Overhead of Direct Application Instrumentation Using Prior Static Analysis**. In: *Euro-Par*. 2,011, pp. 65–76.

[157] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. **VAMPIR: Visualization and Analysis of MPI Resources**. In: *Supercomputer* 12.1 (1,996), pp. 69–80.

[158] Tsuyoshi Nakamura. **BMDP program for piecewise linear regression**. In: *Computer Methods and Programs in Biomedicine* 23.1 (1,986), pp. 53 –55. ISSN: 0169-2607.

[159] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. **ScalaTrace: Scalable compression and replay of communication traces for high-performance computing**. In: *Journal of Parallel and Distributed Computing* 69.8 (2,009), pp. 696 –710. ISSN: 0743-7315.

[160] Cédric Notredame, Desmond G Higgins, and Jaap Heringa. **T-coffee: a novel method for fast and accurate multiple sequence alignment**. In: *Journal of Molecular Biology* 302.1 (2,000), pp. 205 –217. ISSN: 0022-2836.

[161] Robert W Numrich and John Reid. **Co-Array Fortran for parallel programming**. In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM. 1,998, pp. 1–31.

[162] CUDA Nvidia. **Compute unified device architecture programming guide**. 2,007.

[163] Object Management Group. **Common Object Request Broker Architecture**.
http://www.omg.org/spec
Last accessed March, 2,015.

[164] Yury Oleynik, Robert Mijaković, Isaías A. Comprés Ureña, Michael Firbach, and Michael Gerndt. **Recent Advances in Periscope for Performance Analysis and Tuning**. In: *Tools for High Performance Computing*. Springer International Publishing, 2,014, pp. 39–51. ISBN: 978-3-319-08143-4.

[165] OpenACC Working Group et al. **The OpenACC™ Application Programming Interface**.
http://www.openacc.org/sites/default/files/OpenACC20.pdf
Last accessed March, 2,015. 2,013.

[166] OpenMP Architecture Review Board. **OpenMP Application Program Interface Version 4.0**.
http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
Last accessed March, 2,015. ,2013.

[167] Andreas Papritz and Alfred Stein. **Spatial prediction by linear kriging**. In: *Spatial Statistics for Remote Sensing*. Vol. 1. Remote Sensing and Digital Image Processing. Springer Netherlands, 2,002, pp. 83–113. ISBN: 978-0-7923-5978-4.

[168] Antonio J. Peña and Pavan Balaji. **Toward the efficient use of multiple explicitly managed memory subsystems**. In: *IEEE International Conference on Cluster Computing*. 2,014, pp. 123–131.

[169] Brad Penoff, Alan Wagner, Michael Tüxen, and Irene Rüngeler. **MPI-NeTSim: A Network Simulation Module for MPI**. In: *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*. ICPADS'09. IEEE Computer Society, 2,009, pp. 464–471. ISBN: 978-0-7695-3900-3.

[170] Tad B. Pinkerton. **Performance Monitoring in a Time-sharing System**. In: *Commun. ACM* 12.11 (1,969), pp. 608–610. ISSN: 0001-0782.

[171] Carles Pons, Daniel Jiménez-González, Cecilia González-Alvarez, Harald Servat, Daniel Cabrera-Benitez, Xavier Aguilar, and Juan Fernández-Recio. **Cell-Dock: high-performance protein-protein docking**. In: *Bioinformatics* 28.18 (2,012), pp. 2394–2396.

[172] S. Prakash and R.L. Bagrodia. **MPI-SIM: using parallel simulation to evaluate MPI programs**. In: *Simulation Conference Proceedings, Winter*. Vol. 1. 1,998, pp. 467–474.

[173] R Core Team. **R: A Language and Environment for Statistical Computing**. http://www.R-project.org
Last accessed March, 2,015. R Foundation for Statistical Computing. Vienna, Austria, 2,013.

[174] Eugene Raichelson and George Collins. **A Method for Comparing the Internal Operating Speeds of Computers**. In: *Commun. ACM* 7.5 (1,964), pp. 309–310. ISSN: 0001-0782.

[175] **RAMSES**. http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904
Last accessed March, 2,015.

[176] Ashay Rane and James Browne. **Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics**. In: *International Conference on Parallel Architectures and Compilation Techniques*. 2,012, pp. 147–156.

[177] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. **Ensemble-level Power Management for Dense Blade Servers**. In: *SIGARCH Comput. Archit. News* 34.2 (2,006), pp. 66–77. ISSN: 0163-5964.

[178] RedHat and the community. **D-Bus**. http://www.freedesktop.org/wiki/Software/dbus
Last accessed March, 2,015.

[179] D.A. Reed, P.C. Roth, R.A. Aydt, K.A. Shields, L.F. Tavera, R.J. Noe, and B.W. Schwartz. **Scalable performance analysis: the Pablo performance analysis environment**. In: *Scalable Parallel Libraries Conference, Proceedings of the*. 1,993, pp. 104–113.

[180] James Reinders. **Intel threading building blocks: outfitting C++ for multi-core processor parallelism**. O'Reilly Media, Inc., 2,007. ISBN: 0596514808.

[181] P. E. Ross. **Why CPU Frequency Stalled**. In: *IEEE Spectr.* 45.4 (2,008), pp. 72–72. ISSN: 0018-9235.

[182] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. **Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge**. In: *IEEE Micro* 32 (2,012), pp. 20–27. ISSN: 0272-1732.

[183] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. **Adagio: making DVS practical for complex HPC applications**. In: *Proceedings of the 23rd international conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: ACM, 2,009, pp. 460–469. ISBN: 978-1-60558-498-0.

[184] Romelia Salomon-Ferrer, David A. Case, and Ross C. Walker. **An overview of the Amber biomolecular simulation package**. In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 3.2 (2,013), pp. 198–210. ISSN: 1759-0884.

[185] Freescale semiconductor. **AltiVec™ Technology Programming Interface Manual**. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf Last accessed March, 2,015. 1,999.

[186] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. **Bio-inspired call-stack reconstruction for performance analysis**. Tech. rep. UPC-DAC-RR-2014-20. Barcelona Supercomputing Center, Universitat Politècnica de Catalunya, 2,014.

[187] Harald Servat, Germán Llort, Judit Giménez, and Jesús Labarta. **Detailed and simultaneous power and performance analysis**. In: *Concurrency and Computation: Practice and Experience* (2,013), n/a–n/a. ISSN: 1532-0634.

[188] Harald Servat, Germán Llort, Judit Giménez, and Jesús Labarta. **Detailed Performance Analysis Using Coarse Grain Sampling**. In: *Euro-Par Workshops*. 2,009, pp. 185–198.

[189] Harald Servat, Germán Llort, Judit Giménez, Kevin A. Huck, and Jesús Labarta. **Folding: Detailed Analysis with Coarse Sampling**. In: *Parallel Tools Workshop*. 2,011, pp. 105–118.

[190] Harald Servat, Germán Llort, Kevin A. Huck, Judit Giménez, and Jesús Labarta. **Framework for a productive performance optimization**. In: *Parallel Computing* 39.8 (2,013), pp. 336–353.

[191] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. **Identifying Code Phases Using Piece-Wise Linear Regressions**. In: *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*. IPDPS'14. IEEE Computer Society, 2,014, pp. 941–951. ISBN: 978-1-4799-3800-1.

[192] Harald Servat, Germán Llort, Juan González, Judit Giménez, and Jesús Labarta. **Low-overhead detection of memory access patterns and their time evolution**. Tech. rep. UPC-DAC-RR-2015-1. Barcelona Supercomputing Center, Universitat Politècnica de Catalunya, 2,015.

[193] Harald Servat, Germán Llort, Judit Giménez, Kevin A. Huck, and Jesús Labarta. **Unveiling Internal Evolution of Parallel Application Computation Phases**. In: *ICPP*. 2,011, pp. 155–164.

[194] Sameer S. Shende and Allen D. Malony. **The TAU Parallel Performance System**. In: *Int. J. High Perform. Comput. Appl.* 20.2 (2,006), pp. 287–311. ISSN: 1094-3420.

[195] Timothy Sherwood, Erez Perelman, and Brad Calder. **Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications**. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2,001, pp. 3–14.

[196] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. **Automatically characterizing large scale program behavior**. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. 2,002, pp. 45–57.

[197] Karan Singh, Major Bhadauria, and Sally A. McKee. **Real Time Power Estimation and Thread Scheduling via Performance Counters**. In: *SIGARCH Comput. Archit. News* 37.2 (2,009), pp. 46–55. ISSN: 0163-5964.

[198] R. Smith et al. **The Parallel Ocean Program (POP) Reference Manual**. Tech. rep. http://www.cesm.ucar.edu/models/cesm1.0/pop2/doc/sci/POPRefManual.pdf Last accessed March, 2,015. 2,003.

[199] T.F. Smith and M.S. Waterman. **Identification of common molecular subsequences**. In: *Journal of Molecular Biology* 147.1 (1,981), pp. 195 –197. ISSN: 0022-2836.

[200] José M Soler, Emilio Artacho, Julian D Gale, Alberto García, Javier Junquera, Pablo Ordejón, and Daniel Sánchez-Portal. **The SIESTA method for ab initio order-N materials simulation**. In: *Journal of Physics: Condensed Matter* 14.11 (2,002), p. 2745.

[201] David F Stevens. **System evaluation on the Control Data 6600.** In: *IFIP Congress (1)*. 1,968, pp. 542–547.

[202] STG Systems Performance Team. **CPI analysis on POWER5, Part 2: Introducing the CPI breakdown model**.
https://www.ibm.com/developerworks/library/pa-cpipower2
Last accessed June, 2,014. 2,006.

[203] Andrew Stone, John Dennis, and Michelle Strout. **Establishing a Miniapp as a programmability proxy (poster)**. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New York, NY, USA: ACM, 2,012, pp. 333–334.

[204] Andrew Stone, John Dennis, and Michelle Mills Strout. **The CGPOP Miniapp, Version 1.0**. Tech. rep. Technical Report CS-11-103. Colorado State University, 2,011.

[205] Vladimir Subotic, Roger Ferrer, José Carlos Sancho, Jesús Labarta, and Mateo Valero. **Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications**. In: *Euro-Par Parallel Processing*. 2,011, pp. 39–51. ISBN: 978-3-642-23399-9.

[206] Sun Microsystems, Inc. **RPC: Remote Procedure Call Protocol Specification Version 2**.
http://tools.ietf.org/html/rfc1057
Last accessed March, 2,015. 1,988.

[207] Zoltán Szebeny, Felix Wolf, and Brian J.N. Wylie. **Space-efficient time-series call-path profiling of parallel applications**. In: *SC'09*. Portland, Oregon: ACM, 2,009, 37:1–37:12. ISBN: 978-1-60558-744-8.

[208] Z. Szebenyi, T. Gamblin, M. Schulz, B.R. de Supinski, F. Wolf, and B. J N Wylie. **Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs**. In: *Parallel Distributed Processing Symposium (IPDPS), IEEE International*. 2,011, pp. 640–651.

[209] Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. **Performance Analysis of Long-running Applications**. In: *Proc. of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS) PhD Forum, Anchorage, AK, USA*. IEEE Computer Society, 2,011, pp. 2100–2103. ISBN: 978-0-7695-4385-7.

[210] N Tallent et al. **HPCToolkit: performance tools for scientific computing**. In: *Journal of Physics: Conference Series* 125.1 (2,008), p. 012088.

[211] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. **Binary Analysis for Measurement and Attribution of Program Performance**. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'09. Dublin, Ireland: ACM, 2,009, pp. 441–452. ISBN: 978-1-60558-392-1.

[212] Nathan R. Tallent, John Mellor-Crummey, Michael Franco, Reed Landrum, and Laksono Adhianto. **Scalable Fine-grained Call Path Tracing**. In: *Proceedings of the International Conference on Supercomputing*. ICS'11. Tucson, Arizona, USA: ACM, 2,011, pp. 63–74. ISBN: 978-1-4503-0102-2.

[213] Technische Universitaet Muenchen. **Periscope Users Guide**.
http://www.lrr.in.tum.de/~periscop/Doc
Last accessed March, 2,015.

[214] **The Amber Molecular Dynamics Package version 11**.
http://ambermd.org
Last accessed March, 2,015.

[215] **The libunwind project**.
https://savannah.nongnu.org/projects/libunwind
Last accessed March, 2,015.

[216] **The Structural Simulation Toolkit**.
https://www.sst-simulator.org
Last accessed March, 2,015.

[217] James E. Thornton. **Parallel Operation in the Control Data 6600**. In: *Proceedings of the Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*. AFIPS '64 (Fall, part II). San Francisco, California: ACM, 1,965, pp. 33–40.

[218] Robert M. Tomasulo. **An efficient algorithm for exploiting multiple arithmetic units**. In: *IBM Journal of research and Development* 11.1 (1,967), pp. 25–33.

[219] Judith D. Toms and Mary L. Lesperance. **Piecewise regression: a tool for identifying ecological thresholds**. In: *Ecology* 84.8 (2,003), pp. 2034–2041. ISSN: 0012-9658.

[220] **Top 500 Supercomputing sites**.
http://www.top500.org
Last accessed March, 2,015.

[221] F. Trochu. **A contouring program based on dual Kriging interpolation**. In: *Engineering with Computers* 9.3 (1,993), pp. 160–177. ISSN: 0177-0667.

[222] RWTH Aachen University, Gesellschaft numerische Simulation GmbH, Technische Universitaet Dresden, University of Oregon, Forschungzentrum Juelich GmbH, German Research School for Simulation Sciences GmbH, and Technische Universitaet Muenchen. **OPARI2 User Manual 1.1.2**.
https://silc.zih.tu-dresden.de/opari2-current/pdf/opari2.pdf
Last accessed March, 2,015.

[223] Michael Van Biesbrouck, L. Eeckhout, and B. Calder. **Representative Multiprogram Workloads for Multithreaded Processor Simulation**. In: *Workload Characterization, IEEE 10th International Symposium on*. 2,007, pp. 193–203.

[224] Michael Van Biesbrouck, T. Sherwood, and B. Calder. **A co-phase matrix to guide simultaneous multithreading simulation**. In: *Performance Analysis of Systems and Software, IEEE International Symposium on*. 2,004, pp. 45–56.

[225] A. Varga. **Using the OMNeT++ discrete event simulation system in education**. In: *Education, IEEE Transactions on* 42.4 (1,999), 11 pp.–. ISSN: 0018-9359.

[226] J. Vera, F.J. Cazorla, A. Pajuelo, O.J. Santana, E. Fernández, and M. Valero. **FAME: FAirly MEasuring Multithreaded Architectures**. In: *Parallel Architecture and Compilation Techniques. 16th International Conference on*. 2,007, pp. 305–316.

[227] Jeffrey S. Vetter and Michael O. McCracken. **Statistical Scalability Analysis of Communication Operations in Distributed Applications**. In: *SIGPLAN Not.* 36.7 (2,001), pp. 123–132. ISSN: 0362-1340.

[228] Chao Wang et al. **NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines**. In: *26th IEEE International Parallel and Distributed Processing Symposium*. 2,012, pp. 957–968.

[229] Vincent M. Weaver, Daniel Terpstra, and Shirley Moore. **Non-determinism and overcount on modern hardware performance counter implementations**. In: *ISPASS*. 2,013, pp. 215–224.

[230] Peter White. **Relative Effects of Central Processor and Input-output Speeds Upon Throughput on the Large Computer**. In: *Commun. ACM* 7.12 (1,964), pp. 711–714. ISSN: 0001-0782.

[231] C. Eric Wu, Gokul Kandiraju, and Pratap Pattnaik. **High-Performance Sorting Algorithms on AIX**. Tech. rep. IBM Research Division, 2,008.

[232] Wm. A. Wulf and Sally A. McKee. **Hitting the Memory Wall: Implications of the Obvious**. In: *SIGARCH Comput. Archit. News* 23.1 (1,995), pp. 20–24. ISSN: 0163-5964.

[233] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. **The design and use of SimplePower: a cycle-accurate energy estimation tool**. In: *Proceedings of the 37th Annual Design Automation Conference*. DAC '00. Los Angeles, California, United States: ACM, 2,000, pp. 340–345. ISBN: 1-58113-187-9.

[234] Tse-Yu Yeh and Yale N. Patt. **Two-level Adaptive Training Branch Prediction**. In: *Proceedings of the 24th Annual International Symposium on Microarchitecture*. MICRO 24. Albuquerque, New Mexico, Puerto Rico: ACM, 1,991, pp. 51–61. ISBN: 0-89791-460-0.

[235] Achim Zeileis, Friedrich Leisch, Kurt Hornik, and Christian Kleiber. **strucchange: An R Package for Testing for Structural Change in Linear Regression Models**. In: *Journal of Statistical Software* 7.2 (2,002).
http://www.jstatsoft.org/v07/i02
Last accessed March, 2,015., pp. 1–38.

[236] Gengbin Zheng, Gunavardhan Kakulapati, and L.V. Kale. **BigSim: a parallel simulator for performance prediction of extremely large parallel machines**. In: *Parallel and Distributed Processing Symposium, 18th International*. 2,004, p. 78.

[237] D. A. Zimmerman et al. **A comparison of seven geostatistically based inverse approaches to estimate transmissivities for modeling advective transport by groundwater flow**. In: *Water Resources Research* 34.6 (1,998), pp. 1373–1413. ISSN: 1944-7973.