TECHNISCHE
UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation

Knowledge-based Performance Monitoring for Large Scale HPC Architectures

Carla Beatriz Guillén Carías

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des Akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Thomas Huckle
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Hans Michael Gerndt
	2. Univ.-Prof. Dr. Dieter Kranzlmüller Ludwig-Maximilians-Universität München

Die Dissertation wurde am 02.02.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.05.2015 angenommen.

Acknowledgments

Firstly, I would like to thank the people who were directly involved in helping me with my thesis. Without them this dissertation wouldn't have been possible. Special thanks to Professor Michael Gerndt for the mentoring of this dissertation and his valuable support. He provided not only guidance on this thesis but equally supported me on another research project. I would like to express my gratitude to Professor Dieter Kranzlmüeller for his time reviewing my thesis. I am also indebted to Professor Arndt Bode for reading and providing comments to this document.
I would like to express my infinite gratitude to Dr. Matthias Brehm for providing insightful guidance on performance analysis and so much help in many other issues. Special thanks also to Dr. Wolfram Hesse who worked with me three years programming, provided support and advice in statistical matters, and helped me also in several other concerns. Many thanks to Jeanette Wilde, Dr. Sandra Mendez, Dr. Helmut Satzger, Dr. Nicolay Hammer, and Dr. Carmen Navarrete. They provided their generous help in many issues, like proof reading, translations, help with LaTeX, image processing, fruitful discussions, etc. I am grateful to the four students for parts of the implementation: Sebastian Schweiger, and the 3B gang: Benno Willoweit, Benjamin Degenhart, and Bernhard Bücherl.
I owe a debt of gratitude to Conny Wendler, who helped by taking over the workload of user administration to let me program and write this thesis and for being a great office mate. Thanks to all the colleagues in the Application Support Group who contributed in one way or another and for a great teamwork. Thanks to Sabine Linner-Pöffinger, who helped me with the printing of the thesis with dedicated attention. Thanks to many of the colleagues in the High Performance Computing department, especially the system administrators: Dr. Hans-Georg Kleinhenz, Johann Dobler, Dr. Markus Michael Müller, and Dr. Alexander Block.

I would like to thank my family and friends. Without them I wouldn't have even been able to start this work. Special thanks to my dear parents and my dear sister for their never ending affection, support, and for motivating me until the end. Special thanks to my boyfriend Julius for his loving care and support in writing the thesis.

If I forgot to mention someone who should have been here, I apologize to him/her.

This work was developed while being a scientific employee at the Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities, an institution to which I am very grateful for providing me the unique opportunity to research in the field of HPC.

Carla Guillén
Garching, Germany
October 2014

Abstract

This thesis presents concepts for system wide monitoring and performance analysis of supercomputers. They aim at detecting inefficiencies of running applications that are responsible for suboptimal use of HPC systems.
Massive parallel applications which run inefficiently on a supercomputer will block the use of the system and decrease the throughput of applications. The improvements in productivity of numerical simulations are typically conducted with performance analysis tools. In order to have an overview of all the applications that run on a supercomputer, the detection of applications with bottlenecks requires to be conducted automatically, on-line, and with a low time overhead. However, there is a lack of tools that provide on-line analysis without a significant overhead and without instrumentation of user codes for the collection of data on a systemwide basis.
In order to achieve a scalable systemwide monitoring with acceptable overhead, methods and algorithms have been developed and implemented in the PerSyst Tool. On-line analyses are performed with codified expert knowledge on strategy maps which are designed to reveal bottlenecks in an application. A strategy map is comprised of a tree-like structure whose nodes analyze and classify the monitored data. Scalability is achieved in the PerSyst Tool with a hierarchical distributed software architecture: a tree of agents which can operate autonomously and run continuously to measure, analyze, filter, and collect performance data. The architecture is designed to optimize the collection route and minimize the usage of the network interconnect. The performance data is reduced by using two main approaches. Firstly, depending on the resulting analysis the strategy maps determines to collect or discard performance data. Secondly, descriptive qualities of performance data are retained by using quantiles which largely reduce the raw data. Even though, quantiles provide a scalable solution by reducing data, the aggregations in the context of a hierarchy of agents can't be performed with exact calculations at all levels of the agent tree. To reduce the need for estimating quantiles, the mapping of performance data to agents is optimized which enables the precise calculation of quantiles as opposed to quantile estimation.
The concepts for scalable system-wide performance analysis were implemented in the PerSyst Tool and applied to three supercomputers with a different microarchitecture. Detailed results are provided for the Petaflop system SuperMUC, the largest of the three systems, at the Leibniz Supercomputing Centre.

Zusammenfassung

In der vorliegenden Arbeit wurden Konzepte für ein systemweites Monitoring mit integrierter Leistung-Analyse für Hochleistungsrechner realisiert, um ineffizient laufende Applikationen zu detektieren.
Ineffizient laufende massiv parallele Applikationen blockieren unnötigerweise große Teile des Systems und reduzieren den gesamt Durchsatz an Anwendungsprogrammen. Die Produktivität numerischer Simulationen lässt sich durch Leistungsanalysewerkzeuge erhöhen. Um einen Überblick über das Leistungsverhalten aller auf einem HPC-System laufenden Anwendungen zu bekommen, muss eine Überwachung und die anschließende Analyse automatisch, online, und mit geringem Zeit-Overhead erfolgen. Allerdings fehlen bisher Werkzeuge für eine systemweite Sammlung von Performance-Daten, in denen Online-Analysen ohne signifikanten Overhead, und ohne Benutzercode-Instrumentierung möglich sind.
Um ein solches skalierbares systemweites Monitoring mit akzeptablen Overhead zu realisieren wurden entsprechende Methoden und Algorithmen entwickelt und im Rahmen der vorliegenden Arbeit ein Performance-Werkzeug namens *PerSyst Tool* entwickelt. Online-Analysen erfolgen mit sogenannten Strategy-Maps, in denen Expertenwissen für die Aufdeckung von Bottlenecks in einer Anwendung kodiert ist. Eine Strategy-Map stellt eine baumähnliche Struktur dar, deren Knoten die Daten analysieren und klassifisieren. Die Skalierbarkeit des PerSyst-Tools wird durch eine hierarchisch verteilte Architektur erreicht, d.h. einem Baum bestehend aus Agenten, die autonom operieren können und dabei kontinuierlich Performance-Daten messen, analysieren, filtern und sammeln. Die Architektur ist so aufgebaut, dass die Netzwerkbelastung durch das Sammeln der Daten optimal ist. Eine Reduktion der Performance-Daten wird mit zwei Hauptmethoden erreicht: Erstens wird mittels der Strategie-Maps entschieden, welche Performance-Daten gesammelt oder verworfen werden können. Zweitens und entscheidender für die Datenreduktion ist die Verwendung von Quantilen, die die qualitativen und statistischen Eigenschaften der Verteilung der Performance-Daten erhalten. Die Quantil-Berechnungen wirken sich vorteilhaft hinsichtlich der Skalierung des Tools aus, wobei bei der Aggregation entlang des Agenten-Hierarchiebaums die exakte Berechnung der Quantile nicht auf allen Ebenen des Baums möglich ist und eine Schätzmethode verwendet wird, die die statistischen Eigenschaften erhält. Eine optimierte Verteilung der Daten im Baum ermöglicht in den meisten Fällen die genaue Berechnung der Quantile. Die im PerSyst-Tool implementierten Konzepte wurden auf drei Supercomputer mit verschiedenen Microarchitekturen portiert und evaluiert. Detaillierte Ergebnisse wurden in dieser Arbeit für das Petaflop-System SuperMUC des Leibniz-Rechenzentrums dargestellt.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

1 Introduction

High performance computing is an instrument for the sciences for performing numerical simulations. By conducting numerical simulations in large HPC architectures, scientific results can be obtained without the expenses of prototyping or even perform experiments which are not possible in real life. Even though HPC systems are significantly less costly than real experimentation, they still are expensive resources. Massive parallel applications which run inefficiently on a supercomputer will block the use of the system and thus prevent it from producing more scientific results compared to an efficient usage of the supercomputer. Optimizations of numerical simulations are typically conducted with the aid of performance analysis tools. There are more than 20 thousand application runs per month in a petaflop supercomputer [1]. In order to have an overview of the applications, the detection of applications with bottlenecks requires to be conducted automatically and with low time overheads. However, there is a lack of tools that provide on-line analysis without a significant overhead and without instrumentation of user codes for a collection of data on a systemwide basis.

This thesis presents a hierarchical analysis of performance characteristics of parallel applications to detect inefficiencies. Along with this analysis, the design characteristics, implementation, and validation of a systemwide performance monitoring and analysis tool, hereafter PerSyst Tool, for high performance computing systems are also described. The PerSyst Tool analyzes and monitors at application level as well as system level with negligible overhead.

The tool has been implemented, by design, in a hierarchy of software components, called *agents*, to collect and analyze monitoring information. The tree-like hierarchy ensures scalability in all aspects of the tool, especially the scalability for measuring the data synchronously. The tool performs analysis by including expert knowledge on performance patterns with decision trees to determine the analysis path.

A distinct feature of the tool is that it uses information of the placement of running applications in order to perform a balancing of the monitored data in the collecting components. The optimized distribution of the data to the components reduces significantly the communication network usage such that the network is only used partially. This ensures that the traffic in the network is minimized in order to avoid a congestion of the system's network with performance

[1] This information was taken from the statistical data collected at the Leibniz Supercomputer Centre. See http://www.lrz.de/services/compute/supermuc/statistics/.

data.
The tool supports a time based sampling rationale. It, therefore, does not require any instrumentation of code regions. The measurements are performed cyclically at equally spaced intervals. Monitoring information at a certain point in time, just like in a snapshot, is a possibility which can be used to monitor metrics such as instant CPU load. The collected data is always linked to the start of the measurement, also referred to as the *timestamp*. The monitoring tool has no other information of the running code except for the job that runs on the devices which are being monitored and the timestamp.
A limitation of monitoring without instrumentation is that performance data can not be related to the source code. Thus, no detailed analysis can be made to localize and optimize the bottleneck. The only available information are those provided by the batch scheduler.
The amount of raw performance data is reduced by applying filtering of information and aggregation at the level of the job. Aggregation is performed by calculating or estimating a fixed and defined amount of quantiles (like quartiles, deciles, or percentiles) of the observations. This allows the tool to retain the distribution of the observations without knowledge of the available data, while keeping the quality of the information.
The measured performance data is analyzed and classified before it is sent through the transport system of agents. The analysis is done with predefined decision trees, called *strategies*. The strategies are composed of performance analysis objects, or *execution properties*, that incorporate thresholds. The concept of a hierarchical analysis of performance characteristics with strategies and properties has been adapted for systemwide monitoring and is part of the performance analysis and autotuning tool Periscope [74]. Once an inefficiency is found, the analysis renders into assessment on performance issues and guidelines on how to handle the detected symptoms. Recommendations for the removal of the detected bottlenecks are not part of the tool's transport system but appear as a result of registered properties that are shown in the visualization.
The experience acquired in the design, development, and deployment of such a system in three high performance systems in user operation is presented in this document. The analysis with strategy maps was implemented on two architecures (the first system had a simpler monitoring and analysis of data). Detailed results are provided for the Petaflop system SuperMUC at the Leibniz Supercomputing Centre.

1.1 Motivation

Large parallel systems which are built with general purpose processors are used mainly for simulations in a variety of scientific fields (examples, as reported by the TOP500 organization [85], are: meteorology, automotive, aerospace, geophysics, chemistry, finance, and research on semiconductor). Algorithms differ widely among the fields of science which are being tested in a supercomputer. At the Leibniz Supercomputing Centre resources of the SuperMUC system are employed mostly for astrophysics, computational fluid dynamics, physics, life sciences,

chemistry, geophysics, high energy and particle physics, and solid state physics. SuperMUC is a Tier-0 system which started as the fourth fastest HPC system on the TOP500 [86] list in 2012 (with more than 3 Petaflops) and currently remains on this list among the top 12 fastest supercomputers. It comprises 148,608 Intel Sandy Bridge-EP and 8,200 Intel Westmere-EX cores. The HPC activities are promoted by the Bavarian Academy of Sciences and Humanities (BAdW), the Gauss Centre for Supercomputing, GCS, and the Kompetenznetzwerk fuer Wissenschaftliches Hoechstleistungsrechnen in Bayern, KONWIHR. Scientists all over Europe have access to SuperMUC via the Partnership for Advanced Supercomputing in Europe, PRACE. High performance systems in the order of petaflops are expensive resources, so special attention is given in maximizing the throughput of applications running on them. Inefficient applications running on a system will decrease this throughput and can block the system from running other applications which, in turn, hampers the users in obtaining more scientific results in a given period of time. A general roadmap has been identified [14] that strives to increase the efficient use of a supercomputer, the necessary steps are shown in Figure 1.1. The detection phase will reveal the codes which are running inefficiently on a supercomputer. This phase has two steps, the performance measurement and analysis. Once a code with performance problems has been identified, the subsequent phase strives to obtain an optimized code (the optimization effort phase). The analysis step can provide recommendations on how to do a detailed search or how to perform an optimization.

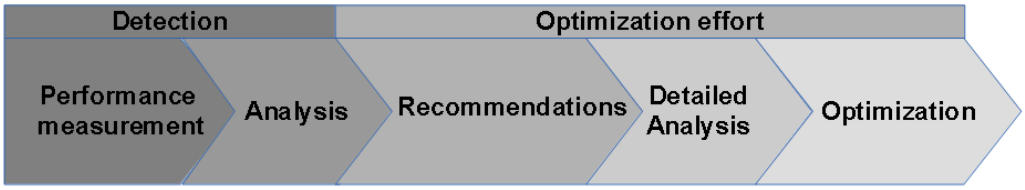


Figure 1.1: Optimization roadmap for applications running on a HPC System.

Detection phase

Monitoring (performance measurement) and automatically analyzing the performance of user codes in order to detect inefficient applications are the preliminary steps to identifying inefficiencies and improving the time-to-solution of codes. Although there are several tools (more than 15 tools/libraries are presented in the next chapter) that provide detailed analysis of performance on codes, these require user involvement. The users need to know how to optimize when a tool indicates that there is a part of the code which has bottlenecks, and they need some knowledge on architectural specific optimizations. Additionally, many of the tools require instrumentation which intrinsically adds time overhead which is at least proportional to the number of instrumented locations [9, 63, 116]. The overhead can drastically grow if the instrumentation is done incorrectly or if the overhead affects individual threads or processes in synchronized communication [67, 68] generating an incremental impact on the overall runtime. Problems may arise at the time of instrumenting or when running an instrumented code: a change of cache behaviour; the instrumentation interacts with the optimization of the compiler; and runtime overhead [29].

Also, users require additional education on the handling of these tools. A preliminary detection of inefficient codes does not require a detailed analysis like the one provided by tools which use instrumented code regions. Given the potential disadvantages that instrumentation brings, application instrumentation has not been considered in this thesis.

Optimization effort
A part of the next step in the roadmap of the optimization endeavour is to provide recommendations, which result from the preliminary analysis. The analysis for detection suggests which changes could be done to optimize a code. Thus, the application developers with bottlenecks in their codes can be given recommendations of optimizations from a first diagnosis.
Detailed analysis is already provided by several tools and optimizations can be done manually or with other automatic tuning tools, so this part of the roadmap is not covered in this dissertation.

Additional system performance information
A by-product of the gathered data is to have a systemwide overview on how an entire HPC system is being utilized. It is important, for instance, to see the percentage of the peak performance as a whole, measured through the use of the floating point operations per second, to compare its usage with other supercomputers. This motivates a synchronized measurement of the performance among applications and across idle nodes. The data can be used in the decision making of procurement of new systems.

1.2 Performance Analysis and Design Objectives

The functionality of the PerSyst Tool enables the monitoring and on-line analysis of the performance of all the applications running in a supercomputer. Analysis of performance data is required to accelerate the recognition of bottlenecks and as a means for raising an alarm. The analysis is also useful in providing recommendations as shown in the roadmap in Figure 1.1. Once an inefficiency is detected, the possible causes can be speculated and thus recommendations on how to remove the inefficiency are given. The specific objectives for performance analysis are:

- To provide on-line analyses based on expert knowledge such that bottlenecks are revealed and these can be related to coding practices. The codified knowledge includes the use of monitored data, derived performance metrics, and thresholds for the evaluation of the severity of bottlenecks.

- General recommendations are given when inefficiencies are detected. Recommendations are not required to be the final solution to remove a bottleneck but point to possible solutions or a course of action to further investigate the bottleneck. The recommendations are only given when an analysis renders an inefficiency.

- To design a hierarchy of analyses such that bottlenecks can be given a priority in order to enable an analysis path that improves the diagnoses of the bottlenecks.
- The codified knowledge for the analysis is configurable and extensible.
- To gain insight of the macroscopic performance of the system in order to see how a supercomputer is being used. Thus, the goal is not just to identify performance bottlenecks in running applications but also to provide an overview of general system performance.

The following system design objectives were pursued in order to enable a suitable operation in a supercomputer:

- Scalability of measurements: scalable and synchronized measurements across an HPC System. The objective is to master large scale machines with potentially several hundred thousand cores, i.e. in the petaflop range.
- Interoperability: the tool is extensible with third party tools and/or libraries in order to profit from new technologies.
- Scalability in the tool management: management time overheads do not increase with the number of nodes of the monitored HPC system.
- Scalability in accessing the data: the querying of the data is also scalable and performance data should be promptly available for querying.
- Portability: The systemwide monitoring tool is ported into different HPC systems, with different hardware architectures and independent of batch systems and/or resource managers.
- Keeping a negligible monitoring intrusion: the monitoring overhead is kept statistically insignificant in a continuously running system. The average time of an application with or without the PerSyst Tool remains statistically the same, with a statistical error of 1%.
- Reduction of information: reduce amount of data through filtering and aggregation. Aggregation of information is done with respect to cores not time. The changes of performance with respect to time for a given application is important to identify the stages of a running code. This information over time could be also used for another monitoring tool that uses instrumentation which can have, as an input, the time where the bottleneck occurs in order to later make a fine grained analysis with instrumented code.
- No required user eduction/training: the users do not require additional information or training in the development phase or production phase of their applications for the purpose of monitoring. No user training is required as they do not trigger the monitoring of their applications. They are also not required to perform any changes in their applications. Monitoring runs on the background and remains unnoticed to the application as opposed to actively interacting with the application.

- No additional user action: users are not required to perform extra tasks other than their normal workflow when using the supercomputing (this usually includes: programming, building, and running the applications). Additional user action can be a consequence of detected performance bottlenecks, but is not directly triggered by the tool.

The concepts presented in this thesis encompass monitoring, analysis, and transport of the data. Approaches to visualize and store the data (such as database approaches) even though necessary, are not covered.

Security, with regard to network communication, is a broad topic with well established solutions. Also, the risks associated to unauthorized access to performance data are not critical, since the performance data remains anonymous. Finally, the possible listeners are a reduced group, i.e. the users of a supercomputer which have special access to it. Therefore, security considerations between communicating components within the monitoring and analysis tool are also out of the scope of this thesis.

1.3 Structure of the Thesis

This thesis is organized in seven additional chapters. The chapter "Approaches To Performance Monitoring" surveys the existing approaches to monitoring solutions, their areas of impact, and how they have influenced the monitoring tool described in this thesis. The next chapter "Execution Properties and Strategies" deals with the analysis methods in order to recognize performance problems. Chapter "Functionality of the PerSyst Tool" describes in detail the funcitonality and system design of the tool. In the context of agent trees, Chapter "Statistical Aggregation of Performance Data" describes how statistical aggregation is conducted. Chapter "Portability and Adaptability" deals with the portability and adaptability issues. Chapter "Context of Evaluation and Results" covers the specific implementation details of the PerSyst Tool on three supercomputers in many different aspects, including different measurements on overhead and scalability. There is also a validation of the results of the tool. The results on monitoring applications are shown in this chapter. Finally, the "Conclusions and Discussions" chapter includes the interpretation of the overall results and conclusions.

2

Approaches To Performance Monitoring

The following section provides a classification of the approaches to performance monitoring. Additionally, the overlapping areas that the monitoring tool in this dissertation has with other tools are described. The taxonomy tables also help determine the relevant differences the tool has with respect to other performance tools. Classifications are shown according to: the level at which it applies, functional approaches, and according to the software architecture. Section 2.2 presents the tools which have overlapping areas of commonality with the PerSyst Tool.

2.1 Existing Approaches and Tools for Performance Monitoring

Performance monitoring tools register and process performance data in order to help detect bottlenecks or to help a developer to optimize a code. There are performance tools which are prominent given that they have steps into analysis of performance data and they do not perform just performance monitoring. A non-exhaustive list of tools and libraries (only few representative tools at each category are shown) which are dedicated to performance monitoring (measuring based on profiling, or tracing) are shown on Table 2.1.

Performance Monitoring	Performance Monitoring and Analysis
Ganglia [7], PAPI [79] pfmon [3], perf [26]	Periscope [38], Scalasca [66], TAU [105], Vampir [62]

Table 2.1: Examples of performance tools and libraries for performance monitoring and performance analysis

The tools and libraries for performance monitoring are either dedicated to: the performance analysis of a single application (application level); the collection of performance data for a system-wide overview for system administrators (system level); or a combination of the two. Direct measurement can be done without knowledge of the source code (there is no instrumentation involved) at system and application level. Once an application is instrumented, the monitoring is done at application level. Examples of other tools in the different monitoring levels are shown in Table 2.2.

Measurement Level	Example of tools
Application level	PAPI [79], Likwid [113]
Application level and system level	TAUoverSuperMON [81], NW-Perf [77]
System level	Ganglia [7], TAUoverSuper-MON [81]

Table 2.2: Taxonomy of performance tools and libraries according to level of monitoring

Performance monitoring of an application can be categorized into monitoring with instrumentation and without it. There are different instrumentation approaches that have been identified [29]. Instrumentation can be applied in the source code by inserting function calls that trigger the measurement and/or trace information collection prior to compilation [119]. Library level instrumentation links the library calls to a wrapper library with added functionality for monitoring. This approach can be automatic and unnoticed by the user if the wrapper libraries replace the desired library but links internally to the library. Binary instrumentation rewrites the binary by inserting binary code that collects the measurements [13]. The advantage of this method is that instrumentation is language independent and no recompilation is needed. Finally, dynamic instrumentation refers to the approach of inserting and removing instrumentation at runtime [9], clearly no recompilation is needed and the approach is also language independent. Table 2.3 shows the type of instrumentations with examples.

Source level	Periscope [38], Scalasca [66] , TAU [105], Vampir [62], Likwid [113]
Library level	IPM [35, 36]
Binary level	ATOM [31]
Dynamic at runtime	Pin [90], DynInst [9], DynaProf [78], DPCL [91]
No instrumentation	Likwid, perf [26], pfmon [3]

Table 2.3: Taxonomy of tools and libraries according to instrumention

From the forgoing three taxonomy tables presented, there is no tool or library which provides performance analysis without instrumentation at a system level with a synchronized measurement for large HPC architectures. The PerSyst Tool encompasses all these features and additionally enables the aggregation and reduction of performance data and optimizes the extraction/collection of performance data. These capabilities are desired due to:

- No instrumentation implies negligible impact or disturbance of running applications.
- Systemwide on-line performance analysis of application level to process the bulk of application runs in a large HPC system.

- Synchronized measurement allows for performance monitoring at system level. For example: systemwide collection of floating point operations, instructions, and other metrics.

- Selection and aggregation of data to handle a reduced amount of data without sacrificing the quality of the monitoring data.

2.2 State of the art

The previous section described tools which are only a fraction of the existent tools for performance measurements. This section will cover the tools which are in a direct relation to the PerSyst Tool in more detail, showing the differences to each tool.

Direct measurement tools
At the Leibniz Supercomputing Centre, tools like pfmon [3] from HP were called at regular intervals and its output is parsed, aggregated, and stored in the file system. The used approach measures the hardware events by using command line tools at the compute nodes which output their data to file systems [16]. These buffered files are then read by another component which stores them into a database. Subsequently, this component deletes the files. All of these procedures are carried out at regular intervals. The approach is implemented in two main steps: a script invokes the measuring tool and produces files in a common file system; then a script parses the output files and stores it into a database. A chronological scheduler was set to cyclically run the previous steps.
Differences or addressed shortcomings: This approach has the downside of having a costly I/O file system accesses which impedes the method to be scaled for machines with over hundreds of thousands of cores; it does not reduce the amount of information and does not provide analysis.

LIKWID
The LIKWID tool [112, 113] is multifaceted and one of its uses is for performance monitoring at system level. LIKWID is based on using the linux kernel module Model-Specific Registers (MSR) which can be found on Intel and AMD processors. A light weight daemon runs on the compute nodes and accepts Unix socket connections as a command to write on the MSR virtual files to read hardware events. The daemon makes probes at a specified cycle time and delivers these raw counters to the requesting client.
The tool can also be used without the daemon to run independently for a specific code as a performance tool. In this case, however, the tool requires special permissions to run. A given application can be instrumented to interact with LIKWID and do specific measurements for regions of the code. LIKWID uses predefined performance groups to select a group of events to be measured. These events are not presented raw to the users, instead derived metrics (rates, ratios and/or other calculations) are made available.
Differences or addressed shortcomings: Although LIKWID has been used as a library

on the PerSyst Tool, its software infrastructure is not adequate to be used as a stand alone monitoring system for automatic analysis and reduction of information. The client components are not designed on a hierarchical structure such that the tool scales to a large number of nodes.

NWPerf

NWPerf does systemwide monitoring on large scale supercomputing clusters at application level [77]. The scalability has not been experimentally tested, but given that it is based on a hierarchical system, the extension and scaling the system is possible. NWPerf measures data via modules at intervals synchronized by the Network Time Protocol (NTP) of a cluster. These modules operate as clients to send performance data over a multicast socket to a packet-handler component. The packet-handler uses a lightweight shared memory queue, which is emptied at intervals by a queue drainer. Data is decoded by the latter component and inserted to a database for permanent storage.

Differences or addressed shortcomings: Monitoring is done without applying on-line analysis, and has no mechanisms for filtering, reducing, or aggregating the information.

MRNet

The Multicast Reduction Network tool (MRNet) is a tool for parallel applications enabling high-throughput communications [17, 55, 93]. Although MRNet is not, per se, a performance measuring tool it can be used for these purposes. Other ways in which MRNet can be customized include debugging, system administration tools, operations of command and control, and data collection and reduction. MRNet uses the principle of a hierarchy of software in a tree topology, also referred to as a tree-based overlay network, for scaling to hundreds of thousands of cores. Multicast is done from the frontend downwards through the tree, until the command reaches the leaves of the tree-topology. Transport of data is done with a bottom-up logic, i.e. from the leaves of the tree to the frontend. Aggregation can be implemented by customisable filters to aggregate data packets. The filters, however, can aggregate data only from piece-wise continuous aggregation functions.

Differences or addressed shortcomings: When MRNet is used as a performance monitoring tool the collection of data is done by using the entire tree topology without optimizing the extraction of data. The purpose of the tool is very broad and does not specialize on performance monitoring.

TAU

A combination of systemwide monitoring and application level monitoring has been developed by Nataraj et al [81]. The authors used SuperMon [75], a monitoring and transportation system, to extract data from the compute nodes. The compute nodes delivered the data from mon (a single node data server) and also TAU [106] to provide application and system performance correlation. TAU provides the information of instrumented application. The tested scalability of Supermon is up to 2048 [108] nodes. Supermon handles client requests to extract performance information from the kernel instead of using a cyclic system-wide measurement rationale. TAU

can be used with other tools, for instance TAU over MRNet [65, 80], to meet the requirements of scalability.
Differences or addressed shortcomings: TAU relies on instrumentation of applications, which may bring unwanted side effects, like the ones discussed in the introductory chapter.

IPM
The Integrated Performance Monitor tool (IPM) is a scalable, portable, and low overhead profiling and tracing tool for application performance monitoring [36,107]. Among other performance measurements, IPM profiles several types of MPI calls. Based on an overloading of the Intelligent Platform Management Interface(IPMI), IPM can be used to application level monitoring with instrumentation. The instrumentation is done automatically, without the user having to activate it, as it is at a library level. All running applications are profiled with a low overhead. As the applications interact with the MPI library, the library registers the number of calls with IPM. Among other features, IPM uses a hashing mechanism, which codes program regions into a signature, and compresses them to represent more than one task and region, in order to lower its overhead. Due to the fact that results across applications are not available, there is not a possibility of having a systemwide overview of performance. IPM also does not have on-line analysis; the applications can be analysed on a post-mortem visualization.
Differences or addressed shortcomings: IPM relies on instrumentation of applications, which may bring unwanted side effects, like the ones discussed in the introductory chapter.

Ganglia
The open source project Ganglia enables distributed and systemwide monitoring for high-performance computing systems [7, 69]. Ganglia is widely used for system administration purposes and is considered to have a highly optimized means of data collection. Using a UDP communication protocol, Ganglia components transmit monitoring data in a unicast or multicast channel. Ganglia aims to achieve low per-node overheads and high concurrency [95], in part by using the UDP communication protocol. Per default, the *gmond* component of Ganglia is configured to send basic metrics such as system load and cpu utilization. User defined metrics contained in key-value pairs can easily be extended by means of added C or Python modules. The *gmond* component is also used as middle-ware to aggregate data and receive from other gmond daemons. Thus, it keeps an in-memory cache of monitored data. The gmetad component is a daemon which polls the gmond components periodically in order to store the information. The latter components are grouped in a hierarchical tree structure to enable high scalability. The default storage engine used is the Round Robin Database tool (RRD Tool) [84] which can be easily used for real-time visualization. Ganglia includes a web frontend component for visualization of performance information.
Differences or addressed shortcomings: Uses UDP, an unreliable communication protocol, and has no in-built capabilities for on-line analysis. The transport system even though optimized by using UDP, is not optimized by reducing the communication paths. Although highly scalable, Ganglia has no inbuilt capabilities of monitoring in a per job basis. In addition

the collection of data among compute nodes is not synchronised, such that it is not possible to correlate data.

The Parastation GridMonitor

The Parastation GridMonitor provides data collection at a desired request from a client [21]. As a commercial software, the GridMonitor's technology remains partially unknown. As such, it is not available to be adapted to a systemwide monitoring system that provides analysis and reduction of information. The Grid Monitor uses several components with different functionalities in order collect performance data [22]. One of these components, the *collector*, is the central process which captures data from the different agents which are running on the nodes. The collector, however, will only request information when a client triggers this request. The structure of data which is passed from an agent to a collector is a key, a value, and a timestamp.
Differences or addressed shortcomings: Monitoring is done without applying on-line analysis. Its technology remains partially unknown as is not open source.

The HOPSA Project

In the HOPSA Project [76] several tools have been integrated in a detection and analysis process for application performance monitoring. The tools carry out either systemwide performance analysis or application performance monitoring in separate stages. The system monitoring contributes with performance information for a holistic view and complements the use of application monitoring and analysis tools. The system monitoring on certain devices may not reflect the effects of one application, but of several applications. The use of application monitoring and analysis tools is constrained in the HOPSA Project to a set of well-known performance tools which cover different areas of performance bottlenecks (Dimemas [103], Paraver [64], Scalasca, ThreadSpotter [12] and Vampir).
Differences or addressed shortcomings: The approach presented by the HOPSA Project is different to the PerSyst Tool. While the PerSyst Tool combines systemwide monitoring and uses analysis for a preliminary detection of inefficient codes, the HOPSA Project has a workflow oriented to a detailed analysis which has an additional systemwide information support.

Periscope

Many of the features that are desirable for systemwide performance monitoring are already existent in the profiling based tool Periscope [38]. Periscope is a scalable tool for analyzing application performance. It enables a distributed on-line search for performance properties based on hardware counters as well as MPI and OpenMP properties [11, 39–41]. Using instrumented code, Periscope provides the possibility of defining a user region within the code [41]. There are also potential look-up regions within the instrumented section of the code which are automatically detected. These regions are analyzed and refined when bottlenecks are detected in order to delimit the section of the code which has a problem. Periscope binds the application with a monitoring request interface which activates the start and stop of measurements of the hardware counters at these regions [61].

In Periscope, the detection of the bottlenecks is done through the evaluation of *properties* in analysis *strategies*. The properties characterize the bottlenecks and are arranged into trees that are refined when a given bottleneck is encountered. These ideas have been used in the PerSyst Tool with some modifications—they have been adjusted to characterize the systemwide monitoring and applications as a black box.
Differences or addressed shortcomings: A tool like Persicope, while it can be effective in precisely uncovering inefficiencies, it is not adequate for the purpose of system wide analysis and monitoring. Periscope provides detailed analysis and is not a system wide monitoring tool itself (it does not have the capability of running as a daemon, thus providing continuous monitoring).

The PerSyst Tool is distinct in that it provides formalized on-line analysis without instrumentation, filtering unnecessary monitoring data and aggregating the monitored data using percentiles [42, 43]. The approach is to conduct a preliminary detection of inefficient codes (which may even be running in production mode as opposed to testing or scaling trials). Additionally, the hierarchical network is only used partially to communicate the monitoring data from the jobs and produce an output as local to the source of information as possible, in this way it avoids network congestion. It does not belong to the set of tools that use instrumentation to correlate data with code regions. The tool uses current technology to cope with scalability and demands to handle the volumes of data produced. Scalability issues are tackled by the hierarchical tree-like architecture of independent and communicating software components. The PerSyst Tool handles conflict monitoring by allowing users to switch off the monitoring infrastructure. Finally, adapting the PerSyst Tool to other tools with system level data, in order to be part of a holistic approach of data collection, is feasible [45].

3

Execution Properties and Strategies

In this Chapter, the analysis abstractions—the *execution properties* and *strategies*—used in the PerSyst Tool are provided.
An analysis done to one or more hardware events that aims to reveal an inefficiency is referred to as *execution properties* or simply *properties*. These abstractions are used in the tool's algorithms to evaluate performance. The properties are organized in tree-like structures that determines whether to continue performing a finer analysis or to ignore monitoring data when no bottleneck is detected. These analysis trees are called *strategies* in this thesis and are used to evaluate the performance in a code. The idea behind a tree structure of properties is to start at the root properties and refine to other properties to search for an inefficiency. Once an inefficiency has been identified at a root property, a more specific inspection will be made with the child properties to delimit the problem which causes it. This idea was taken from Periscope [38] which also refines properties depending on the implemented strategies only when it is needed.
A *strategy map* is an abstraction of analysis which includes the strategy along with recommendations (the latter are a product of the analyses performed). Strategy maps are introduced in the following section. The bottleneck list proposed by Treibig et al. [114] is extended in this section and used for the properties and strategies. The criterion used by the chip vendors [1, 2, 4, 52] to optimize a code has also been taken into account. The criteria used to prioritize the analysis of performance bottlenecks are:

- Limitations inherent to the code: algorithmic constraints, volume of data saturates the internal bandwidth, dominating I/O, etc.
- Optimizations that have the biggest impact on performance: optimizing load imbalance in comparison to optimizing loop-unrolling, optimizing memory accesses compared to optimizing branch prediction, and so on.
- Bottlenecks that typically arise more frequently.

These criteria are applied with the constraint that these performance bottlenecks should be measurable with current measurement tools and applicable for black box monitoring[1]. The

[1]The term *black box monitoring* will be used in this document to refer to monitoring of running executables without knowing the source code which is being run.

possible bottlenecks and optimizations can be formalized into a systematic analysis of code, having the code as a black box.

Section 3.2 provides a detailed definition and explanation of how properties have been implemented. The analyses are centered on the x86_64 Intel architectures: Westmere-EX, and Sandy Bridge-EP. For each architecture the final strategies are provided which may differ from the strategy maps due to events which can't be measured with current available tools.

3.1 Strategies for Monitoring and Analysis

A strategy allows a controlled selection of properties by defining relations among them. A parent-child relation in a property defines a broader problem in the parent property and a more delimited problem in the child property, such that a more detailed diagnosis to an observed performance problem can be done in the latter. For example: the property for stalled cycles is a parent to stalled cycles due to instruction starvation. A strategy may comprise a simple list of properties or a more complex list of trees of properties. All of the strategy nodes in a tree are properties that analyze a specific bottleneck. Cyclic property structures, for instance property A is parent of property B and viceversa, are, by definition, not possible given that this would contradict the idea of doing a refinement in a child property. A strategy has one or more root properties which will always be evaluated, if the measurement data for them is available. If a root property has one or more child properties, these will be evaluated if the root property reports the existence of a bottleneck. Refining over properties or stopping and ignoring a sub-strategy tree is controlled in the same way.

In the following sections strategy maps are shown. The term *'high'* and *'low'*, when used in the context of a property and a strategy map, is precisely defined by a threshold. A threshold is a quantity which delimits what is considered to be an inefficiency (An explanation on how thresholds values can be selected is available in Section 3.2). Each figure illustrates a strategy map. Note that the section numbering has a correspondence to the items shown in the figures. In the strategy maps the boxes can correspond to a property or to an analysis done at the end of the application run (post-mortem stage). When the word 'Strategy:' appears in the title of a section, the section corresponds to a strategy map in a figure and marks the beginning of a block of sections which are related to this figure. The arrows in a strategy map represent selections: when no qualifier (such as 'high' and 'low') is indicated the branch should always be evaluated. Each strategy map branches to several analyses that have to be parsed in order to identify the bottleneck. While red boxes correspond to recommendations, green circles indicate that no optimization is required for the examined performance pattern.

3.1.1 Strategy: Memory bandwidth analysis

The most dominant aspect of single core performance is related to memory movements in the processor and, thus, it is important to classify a code with an input data set as memory bound or compute bound [114]. Memory boundedness limits the optimizations that can be applied at the level of the pipeline. Regardless of additional stalls that the pipeline may suffer, the memory accesses will dominate the latency since they are many times more expensive and will thus hide latencies due to other stalled cycles. A memory transfer has an average latency of 50 to 70 compute cycles (see Tables 3.1 and 3.2 of the two presented architectures in the Sections 3.3 and 3.4). These are average values and not a precise cost in terms of cycles, given that software prefetching and memory affinity in NUMA architectures mitigate the waiting cycles for memory in the DRAM [88]. On the other hand, the opposite effect appears (longer latency) if data in the L1 and L2 caches has to be written on main memory.
The strategy starts by analyzing the memory bandwidth; at this point it is determined whether there is memory saturation or not. Figure 3.1 illustrates the memory bandwidth map.
IF memory bandwidth is *high*:

GO TO Section 3.1.2

ELSE GO TO Section 3.1.9

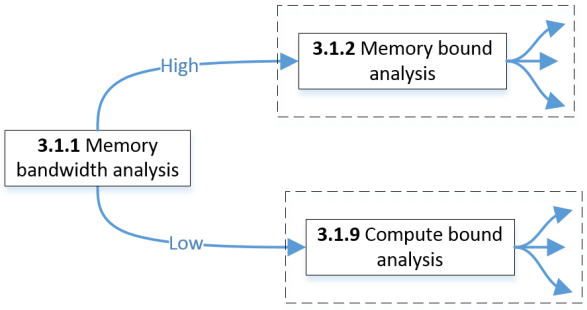


Figure 3.1: Memory bandwidth strategy map

3.1.2 Analysis: Memory bound

If an application is memory bound, the usage of data from the DRAM requires inspection. If the loaded data volume into the cache and registers is not entirely used for the computations, then the unused data volume should be avoided. If the same data is loaded several times to carry out computations, a recommendation is to improve the data locality such that the data is loaded once and all the necessary computations are carried in one go before other data is loaded for computations. If the code is already optimized for moving data blocks to a minimal amount without performing strided accesses, and it is still memory bound, then there is little work to do by optimizing the computations themselves, since the compute latencies will be hidden by the memory transfer latencies.
Figure 3.2 presents the strategy map for the analysis of memory bound code. Trying to detect

strided access or inefficient temporal locality can not be determined separately. However, by examining the metrics shown in Figure 3.2, it can be determined that it has at least one of the two problems, without a distinction if it is a strided access problem or a memory locality problem.

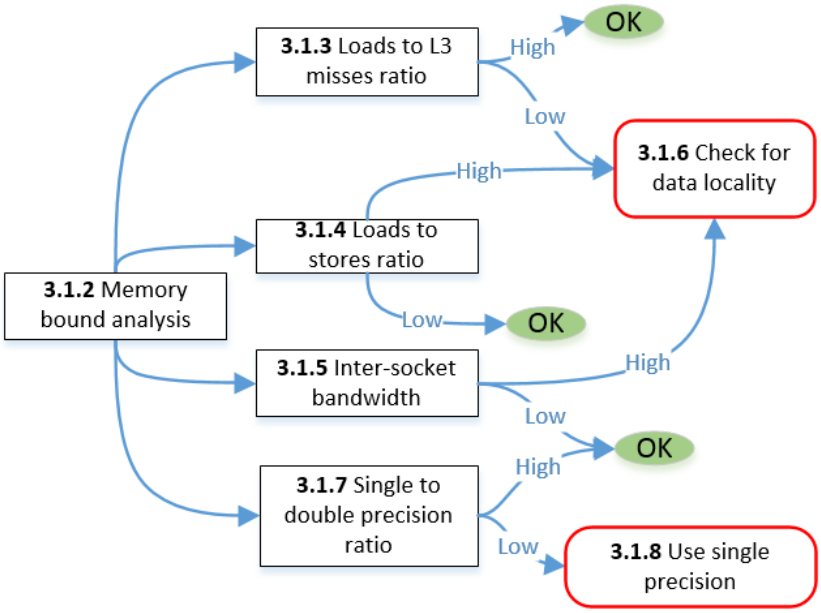


Figure 3.2: Memory bound code strategy map

3.1.3 Analysis: Loads to L3 misses ratio

A diagnosis to determine whether the code suffers from strided accesses or memory locality can be done by examining the ratio of data loads by L3 cache misses. If loads are constantly causing cache misses, this ratio will be *low* and this indicates a bad data locality.
IF loads to L3 misses ratio is *low*:

GO TO Section 3.1.6

ELSE code needs no optimization with respect to this ratio.

3.1.4 Analysis: Loads to stores ratio

The ratio of the loads to the stores gives an idea of the amount of data needed (loaded) in order to produce new data (stored). The lower this ratio, the more efficiently the new data is produced. A *high* ratio is not necessarily a bottleneck but it does give an idea of how the data volume is being used.
IF loads to stores ratio is *high*:

GO TO Section 3.1.6

ELSE code needs no optimization with respect to this ratio.

3.1.5 Analysis: Inter-socket bandwidth

Certain Intel architectures feature two or more sockets with shared memory, with each socket containing several cores. The inter-socket bandwidth can be monitored to examine the data locality. In the case of certain Intel architecture types, the transfer rate through the Quick Path Interconnect, QPI, can be used for this purpose. *High* transfer rates are an indication that data used in one hardware socket has to be fetched in another one (for example, there is bad data locality due to the first touch policy). If there is a *high* bandwidth between sockets, i.e. the NUMA bandwidth, it is recommended that the data placement be checked.
IF inter-socket bandwidth is *high*:
GO TO Section 3.1.6
ELSE code needs no optimization with respect to inter-socket bandwidth.

3.1.6 Recommendation: Check for data locality

This recommendation aims to improve the data locality. It should be checked whether the data volume is not being used due to:

- Strided accesses: if this is the case improve data spatial locality.
- Inefficient temporal data locality: if this is the case try to improve the reutilization of data
- Check first touch policy in OpenMP threads: make sure that data is initialized at the threads were it will be used.

3.1.7 Analysis: Single to double precision ratio

The type of floating point precision that has been used can be analyzed via the ratio of single to double precision. Using single precision instead of double precision halves the memory consumption of the floating point data and the corresponding data transfers.
IF single to double precision ratio is *low*:
GO TO Section 3.1.8
ELSE code needs no optimization with respect to this ratio.

3.1.8 Recommendation: Use single precision

This recommendation consists of reducing the memory footprint of the application. If the code does not suffer from numerical instability with single precision, then single precision should be used as the code will not only perform better, but it will also take up less memory and perform less data movements in terms of bytes. For memory bound codes this can be very beneficial if the application can become compute bound. Therefore, recommendations to a code which uses double precision include analyzing the feasibility of using single precision.

3.1.9 Strategy: Compute bound analysis

If a code is compute bound, the code should also address latencies in moving data that may bring the computation temporarily to a stall. However, these are not necessarily the dominant cause of bad performance. Figure 3.3 shows the different aspects of performance that are explored in a compute bound code.

3.1.10 Analysis: Floating point operation rate

The rate of floating point operations has been endorsed by the scientific computing community as part of the standard set of metrics for performance evaluation. The higher the rate the more efficient the code is considered (an exception is when phoney arithmetic calculations are included in the code just to increase this rate).
IF floating point operation rate is *low*:
GO TO Section 3.1.11 THEN Section 3.1.16
ELSE code needs no optimization with respect to the flop rate.

3.1.11 Analysis: Ratio of vector to scalar operations

Codes with data level parallelism can be vectorized with special instructions that are available in some architectures. Intel provides, for instance, instruction sets with SSE or AVX vectorizations (or both). These instruction sets allow the architecture to perform several operations at once. The higher the ratio of the total number of vectorized operations (with either AVX and/or SSE) to the number of scalar operations, the more efficient the code will execute. Vector instructions can perform (depending on the architecture and if they are in single precision) between 2 to 16 times the number of operations compared to scalar operations. Thus, an equivalent speedup can be achieved by using these vectorized operations.
IF vector to scalar operations ratio is *low*:
GO TO Section 3.1.12
ELSE GO TO Section 3.1.13.

3.1.12 Recommendation: Check data level parallelism

The recommendation when a code has little or no vectorized operations, is to try to remove the dependencies in the loop kernel. If the code exhibits data level parallelism then the recommendation is to compile the code with AVX compilation flag.

3.1.13 Analysis: AVX to SSE ratio

The ratio of the different types of vectorizations can also be measured. The ratio of AVX instructions and SSE instructions is inspected, if the architecture supports AVX instructions. In this item, scalar operations are not taken into account.
IF AVX to SSE ratio is *low*:

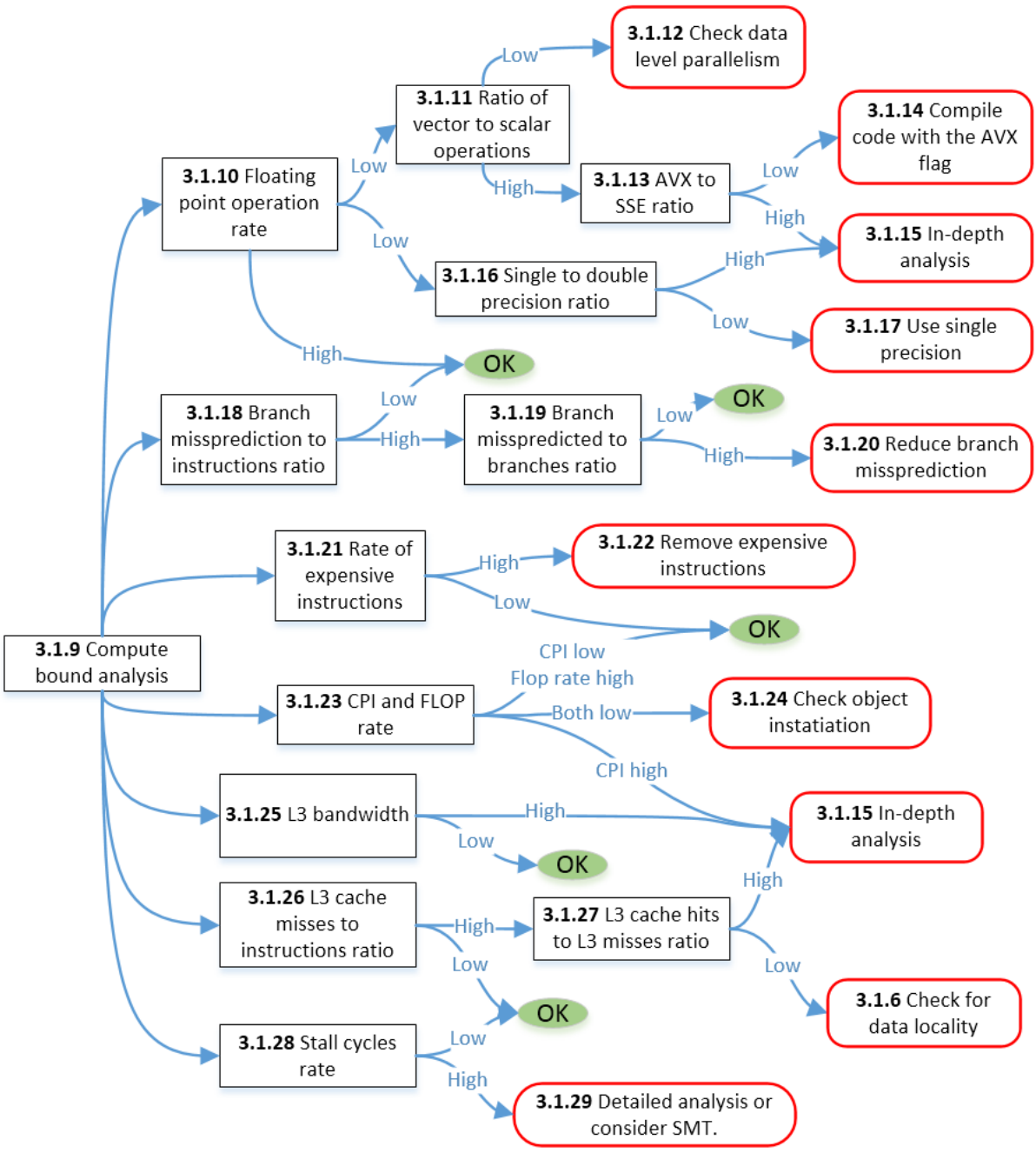


Figure 3.3: Compute bound code Strategy map

GO TO Section 3.1.14
ELSE GO TO Section 3.1.15.

3.1.14 Recommendation: Compile with the AVX flag

A code that has been compiled with SSE instructions has a degree of data level parallelism present. The question is then whether this data level parallelism can be incremented, or whether it is enough to leverage the AVX instructions by simply compiling with the AVX flag (this is the case where flags were simply not used before).

3.1.15 Recommendation: In-depth analysis

If all the analysis failed to give a recommendation, the application developer should instrument the code for a in-depth analysis of performance and code regions. The developer can check for other causes of inefficient performance.

3.1.16 Analysis: Single to double precision ratio

Section 3.1.7 analyzes single precision to double precision of floating point operations to determine whether the memory can be reduced in the case of memory bound code. In some architectures two single precision floating point operations can be performed in the place of one double precision operation, which makes the former more efficient. Thus, the analysis of single precision usage is also needed to improve the performance in compute bound codes.
IF single to double precision ratio is *low*:
GO TO Section 3.1.17
ELSE GO TO Section 3.1.15.

3.1.17 Recommendation: Use single precision

As a recommendation, the application developer should evaluate if the use of single precision is possible. A speed-up of almost a factor of 2 is possible if there are no dependencies.

3.1.18 Analysis: Branch misspredict to instructions ratio

By monitoring the branch misspredict rate to the instructions, wasted CPU cycles due to mispredictions can be detected. The waste of cycles implies not only wasted time but energy, as the execution of mispredicted instructions make the processor work harder.
IF branch miss prediction is *high*:
GO TO Section 3.1.19
ELSE code needs no optimization with respect to mispredicted branches.

3.1.19 Analysis: Branch misspredict to branches ratio

The rate of branch miss predictions with respect to total number of branches will guide the optimization actions. This analysis is done to obtain more insight of the performance with

respect to branches. Thus, only a recommendation will follow.
GO TO Section 3.1.20

3.1.20 Recommendation: Measures against branch misspredictions

The recommendation given is find the conditionals where it is known that jumps (branches) are being performed and analyze the possibility of removing or reordering the conditionals. By removing branches, the misspredictions will subsequently be removed [60]. The developer can allow the branch predictor to correctly predict branches when these unnecessary branches are removed. It is also recommended to try different compiler optimizations.

3.1.21 Analysis: Rate of expensive instructions

Another item to analyze is whether a code has expensive instructions (like divisions, exponentials, square roots, etc), to be found within the compute kernel.
IF expensive instruction rate is *high*:
GO TO Section 3.1.22
ELSE code needs no optimization with respect to expensive instructions.

3.1.22 Recommendation: Remove expensive instructions

The application developer should avoid using expensive instructions within the loop kernel. The recommendation is to reduce associated latencies by checking the expensive instructions in the kernel. The following optimizations can be done:

- If there is a division, the recommendation is to find the reciprocal of the fraction and use multiplication (i.e. change a fraction which is known to its decimal form). The Intel compiler flag `-no-prec-div` gives a hint to the compiler to replace the division by a multiplication with a reciprocal.

- For other expensive operations, it is recommended to try to do the expensive operation outside the kernel (outside of other loops as well).

- If the previous optimizations are not possible, then the application developer should consider implementing a look-up table with the correspondence of a value to the transformation of an expensive function that is computed only once and accessed within the kernel loop, avoiding thus the repetitive expensive instruction. There will be a trade off between wasted cycles due to expensive instructions and memory accesses (look up table), so this solution requires careful evaluation.

3.1.23 Analysis: CPI and Flop rate

Another item to examine is the ratio of *cpi* (clock cycles divided by instructions) and the flop rate. This analysis targets applications which should typically be floating point intensive. The phenomenon of having a *low* ratio of clocks per instruction and, simultaneously, a *low* floating point operation rate, could indicate that the code performs unwanted operations [114]. Incorrect inlining will also produce this effect, when inlined functions which aren't visible to the application developer insert other inlined functions unknowingly. These usually happens when inlining constructors which initialize other classes and/or inherit from other classes.
IF CPI is *high*:
GO TO Section 3.1.15
ELSE IF CPI is *low* AND flop rate is *low*:
GO TO Section 3.1.24
ELSE code needs no optimizations.

3.1.24 Recommendation: Check object instantiation

A typical bottleneck which exhibits the symptom of *low* CPI and *low* floating point rate is the excessive object instantiation in C++ [114]. Object instantiation in object oriented languages (like C++) creates more latency with unnecessary instructions than normal in-built types. These are usually hidden functions that are not explicitly written by the developer and must be included in the compiled code [72].Therefore, the recommendation is to analyze the possibility of using less objects (C struct in the case of C++) and, thus, to remove unnecessary instructions.

3.1.25 Analysis: L3 bandwidth

This analysis aims to determine whether a code is bound by the L3 cache accesses.
IF L3 bandwidth is *high*:
GO TO Section 3.1.15
ELSE code needs no optimizations with respect to L3 bandwidth.

3.1.26 Analysis: L3 cache misses to instructions ratio

If the L3 cache misses to instructions ratio is *high* compared to the total number of retired instructions, this indicates that the L3 cache misses dominate.
IF L3 cache misses to instructions ratio is *high*:
GO TO Section 3.1.27
ELSE code needs no optimizations with respect to L3 accesses.

3.1.27 Analysis: L3 cache hits to L3 cache misses ratio

If L3 cache misses are high with respect to the total number of instructions, the fraction of L3 cache hits to L3 cache misses will bring more insight into the cache utilization. Note that the recommendations given to this symptom are the same as the ones in Section 3.1.6.

IF L3 cache hits to misses ratio is *low*:
GO TO Section 3.1.6
ELSE GO TO Section 3.1.15.

3.1.28 Analysis: Stall cycles rate

The last issue to be analyzed for a compute bound code is to evaluate whether there is a high rate of pipeline and register stalls. Ideally, the processor's pipeline is filled with instructions to process. Bubbles (also known as nops, or no operations) inside the pipeline do not contribute to the processing of a code, they are wasted resources. Cycles which stall are filled with these bubbles and, thus, the rate of stalling can be taken as a measure of inefficiency. A *high* rate of stalls indicates that the pipeline is used inefficiently.
IF stalls cycles rate is *high*:
GO TO Section 3.1.29
ELSE code needs no optimizations.

3.1.29 Recommendation: Detailed analysis or consider SMT

This recommendation is to perform a more in depth analysis with an instrumented code and more hardware events per code region to detect the problem. A recommendation when the exact cause of the bottleneck in the code can not be determined is to use the hyperthreading technology in the modern Intel architectures. Hyperthreading technology is also known as Simultaneous Multi-Threading (SMT). It increases the throughput by allowing the pipeline to be fed from two separate sets of hardware registers. A code with many bubbles in the registers and in the pipeline will benefit by sharing these unused resources between two logical threads.

3.1.30 Strategy: I/O analysis

Other aspects of interest to analyze in an application include their I/O operations. The strategy map defined for this purpose is shown in Figure 3.4. This strategy is independent from the single core analysis and the memory bandwidth strategy maps previously explained. There are several metrics which give insight into the use of I/O. Disk I/O is extremely expensive and many times slower compared to computation cycles, at least 10,000 times more according to [27]. Even if I/O operations remain unavoidable, the forms of accessing a file system can be optimized in diverse ways.

3.1.31 Analysis: I/O imbalance

A parallel application with tens of thousands of cores or more can perform I/O from only one process. This first approach of performing I/O will cause a serious bottleneck in this particular process. Receiving (or sending) all the data from (or to) the rest of the processes produces the first bottleneck. Moving the bulk together to a parallel file system is another bottleneck. A

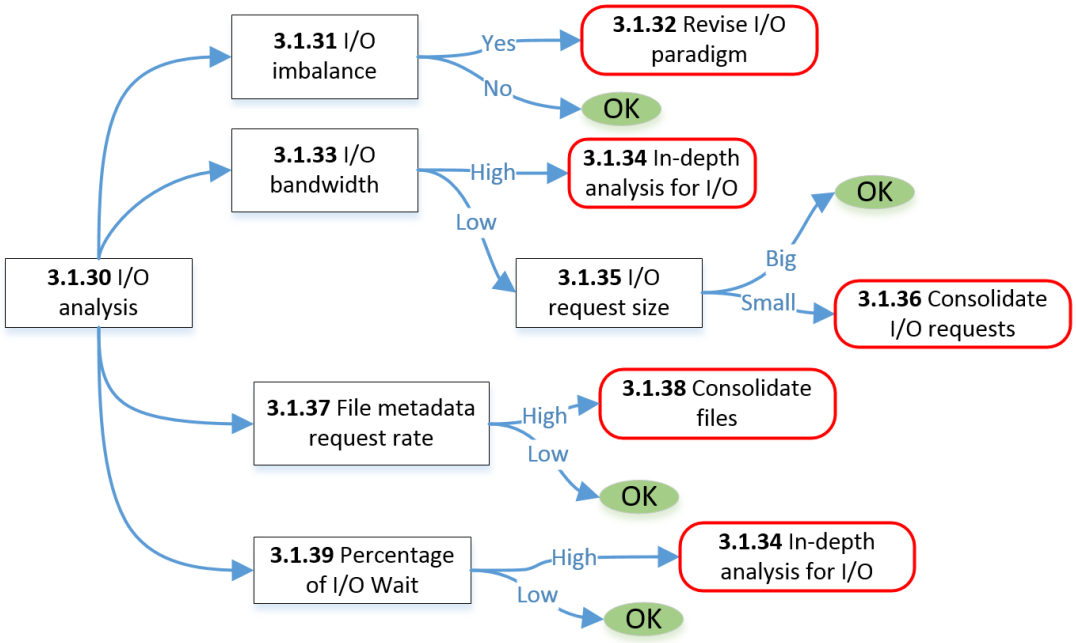


Figure 3.4: I/O Strategy Map

second approach is to perform I/O from each process. However, this may cause a bottleneck in the file system itself when receiving all of these requests, especially if all processes are opening different files (due to meta-data issues). A third approach is a hybrid form using the previous two variants, i.e. to perform I/O from some of the processes. The processes which perform I/O are called I/O master processes and these receive and send the data to their group of processes. Finding the best ratio of I/O master processes to the other processes will depend on several factors, like the amount of I/O written by the application, and the file system used and its configuration. The hybrid variant defines a number of processes in a group, and the entire requested cores are divided into groups. All the members of one group move the data, in a first step, to a designated process within this group and this process performs, in a second step, the I/O operations. The fastest communication for the first step is at the level of a shared memory instance, or a node, where no network communication is needed, only internal socket communication at most.

I/O imbalance refers to an unequal amount of requests and/or request sizes (I/O work) among I/O master processes which results in uneven time spent when performing I/O. Another form of I/O imbalance is when the ratio of I/O master processes to all the processes is suboptimal (the best example for this is the case where one process performs all of the I/O in a parallel application). In this case, the distribution of I/O work is done sub-optimally even if the I/O master processes perform an equal amount of requests and request sizes. Data gathering and scattering to and from I/O performing processes also causes a bottleneck and an ineffective parallel access to the file system is a result of inappropriate distribution of the I/O work.

Under the assumption that the ratio of I/O master processes is optimal, and not all processes are I/O master processes, processes which are exempt from performing I/O operations cannot

be considered imbalanced with respect to the processes which do.
I/O imbalance can also be caused by the file system, if two independent I/O requests of similar size take considerably different times for completion. In this case, the I/O imbalance is due to the unpredictability of the execution time of an I/O request, hindering the application developer and the runtime environment from properly balancing the I/O requests among the I/O master processes.
IF I/O operations are imbalanced:
GO TO Section 3.1.32
ELSE code needs no optimizations with respect to I/O imbalance.

3.1.32 Recommendation: Revise the I/O paradigm

Given the exceptions when considering I/O imbalance (file system conditions, optimal I/O master processes ratio, etc.), it is not straight forward to detect I/O imbalance with black box monitoring; or at least not for all of the different cases. However, the following recommendations can be given:

- Try to evenly distribute the I/O work among the I/O master processes to achieve potential speedups.
- Try to apply dynamic load balancing. Dynamic I/O load balancing has proven to be an effective technique to achieve better I/O performance [89].

3.1.33 Analysis: I/O bandwidth

I/O bandwidth gives an insight into the volume requested to the system. If the bandwidth is *small* this might indicate an inefficiency of the application due to small sized I/O requests. It could also be due to the load on the file system from other applications.
IF rate of I/O bandwidth is *high*:
GO TO Section 3.1.34
ELSE GO TO Section 3.1.35.

3.1.34 Recommendation: In-depth analysis for I/O

The recommendation is to perform in-depth I/O analysis with instrumentation (For example with Darshan [20]). This recommendation also includes a revision to the parallel paradigm. The objective is to try to reduce the amount of I/O.

3.1.35 Analysis: I/O requests size

Relatively frequent but *small* I/O requests is an indication of suboptimal use; the best approach is to try to bundle the I/O data to make larger I/O requests.
IF rate of I/O with *small* bandwidth is *high*:
GO TO Section 3.1.36
ELSE code needs no optimizations with respect to I/O request sizes.

3.1.36 Recommendation: Consolidate I/O requests

Optimizations to consolidate I/O requests include: bundling I/O requests into fewer but larger requests (applied in the Two-Phase optimization provided by the MPI-IO ROMIO standard [110]); data sieving techniques (also from the MPI-IO ROMIO); correcting I/O imbalance; using unformatted I/O; using compression; and using non-blocking (asynchronous) I/O.

3.1.37 Analysis: File metadata request rate

File metadata is requested when a file is opened. Sending this metadata to the requesting process has an associated time overhead. I/O operations can be performed to and from a single shared file. In this case the processes know where to access the files by using their calculated offset.

IF rate of I/O with small bandwidth is *high*:

GO TO Section 3.1.38

ELSE code needs no optimizations.

3.1.38 Recommendation: Consolidate files

The recommendation to reduce the metadata time overhead is to consolidate files, if there are many created. When the same file is being opened and closed, the recommendation is to keep it open until there are no more I/O operations accessing it.

3.1.39 Analysis: Percentage of I/O Wait

The I/O Wait metric provided by the SAR [33] utility from Linux, records the percentage of time used for heavy load in I/O requests.

IF percentage of I/O Wait is *high*:

GO TO Section 3.1.34

ELSE code needs no optimizations.

3.1.40 Strategy: Load imbalance analysis

Parallel applications that suffer from load imbalance (unequal work distributed among threads or processes) will have resources idling. In terms of CPU cycles, load imbalance is one of the most expensive problems in parallel applications in the range of petaflops [114]. A delayed working thread or process causes thousands of other processes to remain idle, which translates directly into a massive waste of resources. If a code is running with load imbalance, it can not be said that it is memory bound, even if a portion of the tasks or threads use the maximum available bandwidth [88]. Thus, in the context of parallel programming, load imbalance is the first priority of the list of performance bottlenecks. Load imbalance can be classified into intra-node load imbalance and inter-node load imbalance. The load imbalance strategy map presented in Figure 3.5 is divided into inter-node imbalance and intra-node imbalance.

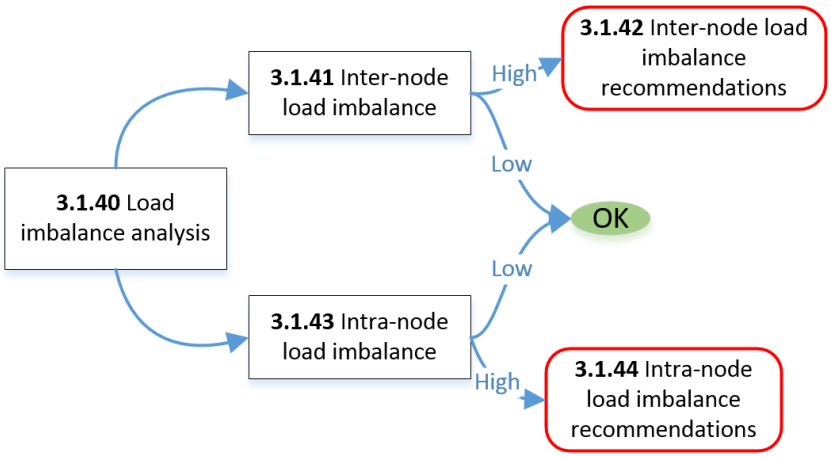


Figure 3.5: Imbalance Strategy Map

3.1.41 Analysis: Inter-node load imbalance

Inter-node process communication deals only with communication via an interconnect network. This type of imbalance is more severe as it involves the idling of more resources than in the intra-node load imbalance case.
IF inter-node load imbalance is high:
GO TO Section 3.1.42
ELSE code needs no optimizations with respect to inter-node load imbalance.

3.1.42 Recommendation: Inter-node load imbalance

Recommendations for an inter-node imbalance include an in-depth analysis of the parallel implementation and the parallel paradigm used. Other recommendations given to the application developer include the use of non-blocking communication. The application developer should consider using the Dynamic Load Balancing (DLB) scheme.

3.1.43 Analysis: Intra-node load imbalance

Intra-node load imbalance occurs only within a node. Thus, the amount of resources idling are bounded by the amount of resources in the node. Nevertheless, correcting imbalance problems may improve other bottlenecks (such as memory boundedness of a group of cores in the node).
IF intra-node load imbalance is high:
GO TO Section 3.1.44
ELSE code needs no optimizations with respect to intra-node load imbalance.

3.1.44 Recommendation: Intra-node load imbalance

In-depth analysis is recommended. If the code has been parallelized with message passing, the recommendation is to use threads (with libraries like OpenMP).

3.1.45 Strategy: Analysis of usage of other resources

The usage of other resources analysis map is used to correlate the information of other resources used by a parallel application with other obtained data. This map focuses on the monitoring of data, whose analyses are on a post-mortem stage[2]. Exceptions to this include the network usage. Figure 3.6 shows the strategy map for the monitoring of other resources.

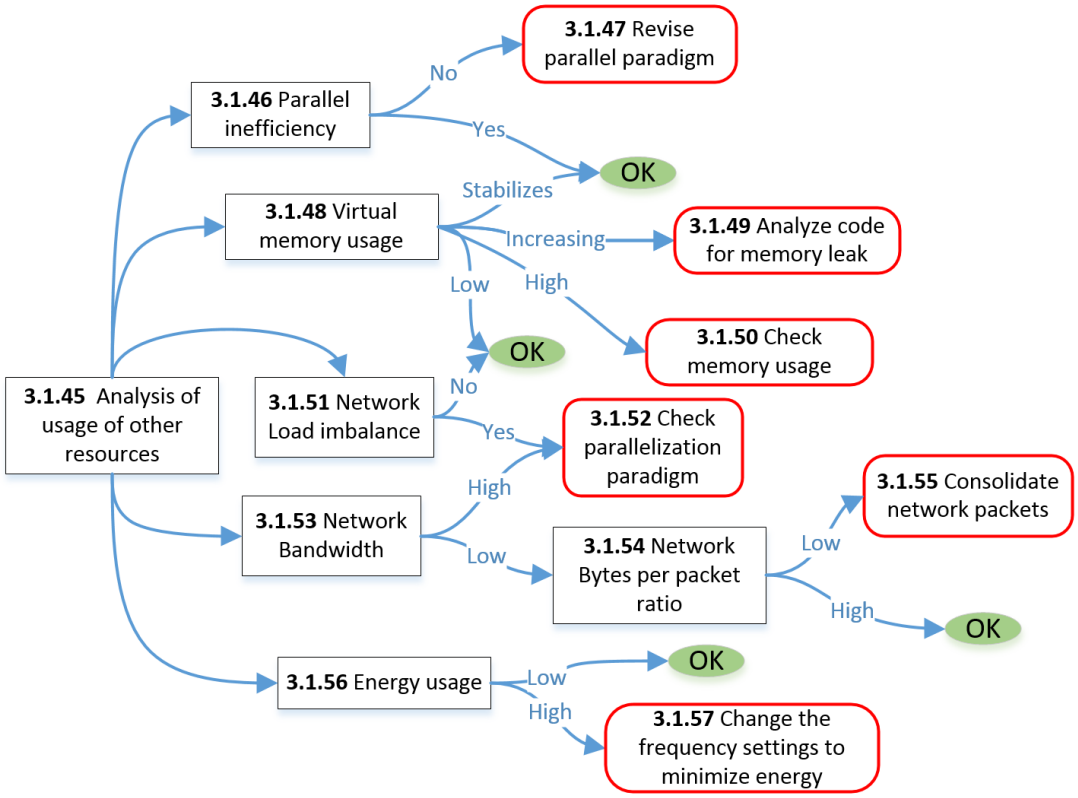


Figure 3.6: Other Resources Strategy Map

3.1.46 Monitoring: Parallel inefficiency

Parallel efficiency is defined as the speedup divided by the number of processors used. Even though parallel inefficiencies do not manifest themselves with a small number of cores [18], they will impede the scaling of the code. Scalability can be quantified by comparing the execution time of the same code run with different core numbers and calculating its speedup (with weak or strong scaling). Comparing speedups with the number of processors will determine whether a code is running with inefficient parallelism and whether the parallel paradigm should thus be revised. Detecting parallel efficiency requires comparing data between jobs. It is also usually the case that the analysis is done after the execution time is over, where the application developer would benefit more from comparing the entire performance metrics spanned over time between both jobs. A bigger effort on coding the analysis of parallel efficiency, plus the necessary changes

[2]The post-mortem stage refers to the stage after the application terminated, as opposed to on-line analysis.

to the framework, would render very little benefit. However, the measurements that contribute to this analysis in a post-mortem stage are indeed necessary. Monitoring without on-line analysis should not be left out, given that the measurements will provide helpful insight on the parallel efficiency at the visualization stage. The application developer will be able to compare at this point two or more jobs to see whether a code scales or not.
IF parallel inefficiency is detected:
GO TO Section 3.1.47
ELSE code needs no optimizations with respect to parallel paradigm.

3.1.47 Recommendation: Revise parallel paradigm

The recommendation is to use in-depth analysis tools to analyze the adequacy of the parallel paradigm used.

3.1.48 Analysis: Memory usage

The monitoring of the memory used can help to determine whether there is a memory leak. Operation policies in the supercomputing sites in Germany allow applications to run with a fixed amount of time, with resource managers preventing the application from running longer than the specified wallclock. A memory leak will make a continuously running programme eventually crash. Scientific application codes do not run continuously and can terminate successfully even with a memory leak. However, the fixing of a memory leak can potentially remedy certain problems within scalable efficiency, and memory usage. In an operation environment, a node can be reserved to use some of its cores, while one or two cores remain idle, due to memory constraints. This scenario is a workaround when the amount of memory per core is not sufficient. If this constraint, however, is due to a memory leak, removing the memory leak will reduce the amount of memory that can be assigned per core, allowing a better allocation of resources.
IF virtual memory is increasing:
GO TO Section 3.1.49
ELSE IF virtual memory is *high*:
GO TO Section 3.1.50
ELSE code needs no optimizations with respect to virtual memory.

3.1.49 Recommendation: Analyze code for memory leak

As a recommendation, the user should analyze the possibility of reducing the memory per core and fitting the application in a lower number of cores. Even if this is not the case, it is still suggested to check the code with a tool that uncovers memory leaks such as Valgrind [28].

3.1.50 Recommendation: Check memory usage

The possibility of reducing the total memory should be studied.

3.1.51 Analysis: Network interconnect imbalance

The network interconnect refers to the network connection beetween two compute nodes, or groups of nodes (such as racks, islands, and other group domains). Network interconnect imbalance refers to an unequal bandwidth transmission among several nodes. Imbalance in the network transmission can be analysed on a post-mortem stage. Transmission imbalance can be a symptom of unequal data sets at each node.
IF network imbalance has been detected:
GO TO Section 3.1.53
ELSE no optimization is needed with respect to network imbalance.

3.1.52 Analysis: Network interconnect bandwidth

Network bandwidth and its usage can also be monitored. Packets which are too small relative to the highest packet size should be consolidated to a larger one.
IF communication bandwidth is *high*:
GO TO Section 3.1.53
ELSE GO TO Section 3.1.54.

3.1.53 Recommendation: Check parallelization paradigm

High communication bandwidth is not necessarily a symptom of bad performance. However, an in-depth analysis and an evaluation of the parallelization paradigm used is recommended. Another recommendation is to reduce the amount of communication: The feasibility of consuming more data locally at a node, as opposed to sending it to other nodes, should be studied. If there is a network imbalance, dynamic balancing of data should be considered.

3.1.54 Analysis: Network Bytes per packet

The mean size of the packets across the network interconnect can be analyzed to see whether the network is being used efficiently. *Small* packet sizes do not optimally use the network interconnect.
IF the mean network bytes per packet is *small*:
GO TO Section 3.1.55
ELSE code needs no optimizations with respect to network usage.

3.1.55 Recommendation: Consolidate network packets

The recommendation when many small network packets are sent, is to consolidate them into a larger packet.

3.1.56 Analysis: Energy usage

Another resource to be monitored is the energy consumption (or instant power) of the application. The power consumed in a processor is an increasing superlinear function of the processor's frequency. Energy is the product of power and time, and by lowering the frequency of the processor the power consumption is also lowered. However, in most cases lowering the frequency will result in a longer runtime of the application, which in turn might result in an increase of the energy consumed. Thus, it is necessary to find the appropriate frequencies that optimize the energy consumption of an application. The energy consumption strategy will help to gain insight on how a resource is being utilized.
IF energy consumption is *high*:

GO TO Section 3.1.57

ELSE code needs no optimizations with respect to energy consumption.

3.1.57 Recommendation: energy consumption

The settings of the core frequency can be changed in order to influence the energy consumption. Performance data can be collected to apply a model which finds the optimal frequency for minimizing the energy consumption [10]. As a recommendation in the case of load imbalance, the thread (or task) with more load should be set to a higher frequency to minimize the idle time of the other threads [94]. Another recommendation is that the application developers experiment with different *governors*, or processor frequency policies, to minimize the energy consumption [82].

The foregoing sections, which dealt with the monitoring or analysis of resource usage, together comprise a unified scheme for monitoring. The independent strategies, i.e. strategies which are not connected to a tree, will be handled at the same level as the root of a tree strategy. Thus, the strategies for I/O, parallel efficiency, load imbalance, memory bandwidth, and resource usage will always be monitored. A tree strategy will start to monitor the root cause and only refine when this analysis deems it necessary to continue to evaluate other properties; the same concept is used in the Periscope search strategies [38]. In the next section, the analyses done at each stage in a strategy are transformed into concrete properties.

3.2 Execution Properties

The objective of performing a first diagnosis at the monitoring stage is fulfilled by defining analysis formulae based on hardware events and other available metrics. These equations, along with the severity and the threshold, formalize the search for bottlenecks and constitute a property. The APART Specification Language (ASL) [32] describes a way to formalize the performance patterns. This specification has been adapted to the needs of performance monitoring as a black box by including the *property value* (or monitoring value). The properties can directly relate an inefficiency with a low-performance coding practice. A property determines the degree of

severity of a performance deficiency, with the aid of two values: the *property value* and the *threshold*. The `Abstract Property` Class is shown in Listing 3.1.

Listing 3.1: Abstract Property Class

```
1  class Property {
   protected:
     float propVal;
     string *metricList;
5    FORMULA_ID formID;
     float severity;
     float threshold;
     float exponent;
9    float evaluateSeverity();
   public:
     Property();
     virtual ~Property();
13   virtual void evaluate(PerformanceDataBase &pdb, int dev_id) = 0;
     virtual int id() = 0;
     virtual bool condition();
     float getPropVal() const;
17   float getThreshold() const;
     float getSeverity() const;
   };
```

The property value and the severity are called `propVal` as shown in line 3, and `severity` as shown in line 6. The severity is a normalized value between 0 and 1, where 1 indicates a severe bottleneck and 0 no bottleneck. The array `metricList` holds the native hardware counters that will be used in the property.

The properties evaluate the performance at different domains (for instance at logical hyperthread level, at core level, processor, and node level) depending on the available hardware events for measurement. Every type of domain (also known as device) that is evaluated receives a unique identification, the device id, or as shown in Line 13 the `dev_id`, regardless of the domain they represent. The `PerformanceDataBase` holds the performance measurements that the `evaluate` method uses to calculate the `propVal`.
Given that object instantiation decreases performance, an optimization done to the Abstract `Property` Class in Listing 3.1 is to instantiate it only once. The `evaluate` method (Line 13) resets the member variables `propVal` and `severity` and evaluates them again. The *getters* are called right after this evaluation to extract the information to make an output (either to a database, a file system, or are sent through the network to another location).

The property `id`, see line 14, will return a designated identification number which is unique to this property.

The decision whether more specialized properties should be analyzed (the strategy defines the sequence of properties) is given by the `condition`, see line 15. According to the properties defined in a strategy, only those properties whose `condition` evaluates to true will be output. The `condition` evaluates to true if the severity is greater than zero or for those properties that require that they be collected even if their severity is zero, or false otherwise.

There are two common severity formulae used for the properties which can be used. They can also be overridden (member variables at line 5 and 8 from the Abstract `Property` Class in Listing 3.1). Since they are commonly used, they are available on the Abstract `Property` Class, and the `severity` can be computed by defining these two member variables and calling the `evaluateSeverity` method. The formula evaluated in the `evaluateSeverity()` method can take the following two possibilities:

- The first formula (`formID = FORMULA1`) forces the values to the range $0 \leq x \leq 1$ by calculating:

$$s(x) = \begin{cases} 0 & \text{if } x/t - 1 < 0 \\ 1 & \text{if } x/t - 1 > 1 \\ (\frac{x}{t} - 1)^p & \text{if } 0 \leq x/t - 1 \leq 1 \end{cases} \tag{3.1}$$

  where s is the severity, x the property value (`propVal`), p the exponent (equivalent to the `exponent` in Abstract `Property` Class in Listing 3.1), and t the `threshold`. This formula is used when an increase in the property value makes the bottleneck more severe (severity increases). The parameter p in this formula will accelerate the growth of the severity ($0 < p < 1$) or delay its growth ($p > 1$). Figure 3.7 shows the effect of p on the severity formula (the range can be wider and only the values $0.5 < p < 2$ are shown) and the threshold t is shown with a green line. On the graph, the property value is increasing to the right and shows no values, to make it generic for any property.

- The second formula (`formID = FORMULA2`), which is used when the severity is higher with lower property values, is expressed as:

$$s(x) = \begin{cases} 0 & \text{if } (x/t)^p > 1 \\ 1 & \text{if } (x/t)^p < 0 \\ 1 - (\frac{x}{t})^p & \text{if } 0 \leq (x/t)^p \leq 1 \end{cases} \tag{3.2}$$

  where s is the severity, x the property value (`propVal`), p the exponent (equivalent to the `exponent` in Abstract `Property` Class in Listing 3.1), and t the `threshold`. Figure 3.8

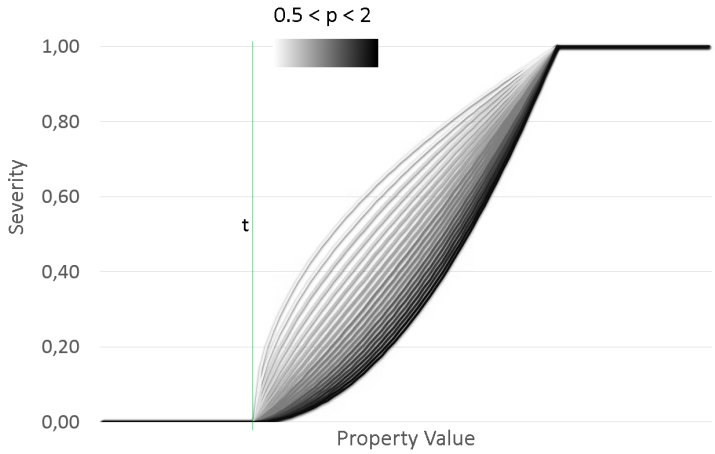


Figure 3.7: Severity formula `FORMULA1`

shows the effect of p on the severity formula. Similar to the previous severity formula p can delay or accelerate the growth of the severity. On the graph, the property value is increasing to the right and shows no values, making it generic for any property. The parameter p which can take values $p > 0$ is shown (only the range [0.5, 2] are shown); and the threshold, t, is shown with a green line.

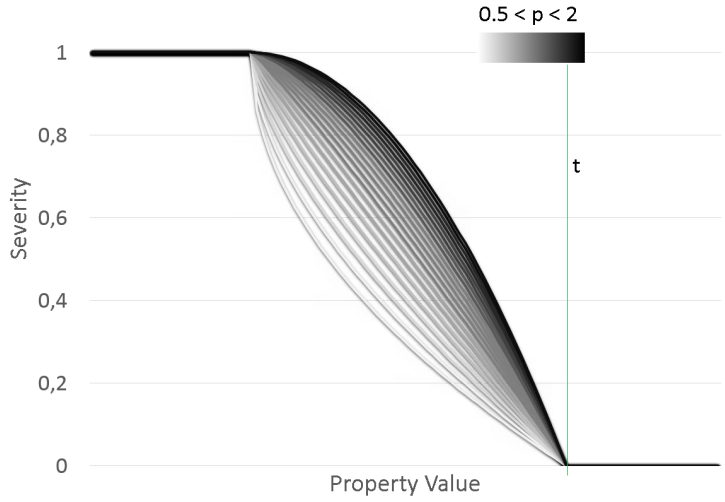


Figure 3.8: Severity formula `FORMULA2`

Thresholds can be chosen based on the following heuristics:

- A threshold can be based on the hardware characteristics and expert knowledge. For example: The threshold of the flop rate is 15% of the peak flop rate.
- A threshold can be based on a benchmark. For example: A code is considered memory bound when the compute node is saturated with memory accesses using benchmark "X".
- A threshold can be chosen at the point where the performance does not significantly change

when improving the property value. In this case, the definition of what is significant is based on the decision of the performance expert. For example: After increasing the I/O mean size request with more than 2MB, the performance gains are less than 5%.

- A threshold is chosen based on statistical data. For example: Take the raw data from all applications running in one week and set the threshold where the 80th percentile lies.

Example: The CPI property is given as an example for how the member variables are set on the Abstract `Property` Class (Listing 3.1). CPI uses the hardware events for clock cycles and instructions. The Abstract `Property` Class (Listing 3.1) can hold them in the `metricList` member variable as shown in Line 4. The count of clock cycles and instructions are, in general, directly available (rather than derived from other events) for all architectures. The events can also be derived from other events, and the `metricList` would then hold the native events, and the evaluation of the property (Line 13) should calculate the derived events. The `propVal`, clock cycles divided by instructions, is calculated by the `evaluate` method. The bottleneck can be considered to be more severe with an incrementing CPI after a threshold of 1.6. This threshold was obtained from observing application runs during one week, and taking the 20th percentile. The `severity` s can be calculated by using `FORMULA1`. In case of the CPI property the filtering will be activated, i.e. the property is only collected when the `severity` is greater than zero. Thus, the `condition` will evaluate to true if and only if the `severity` is greater than zero.

Special case: For the case of the Memory Bandwidth property, there is a need to determine whether there is saturation in order to trigger the corresponding analysis sub-tree; and not if there is a severity (see Figure 3.1). Even if the result is that the code is memory bound or compute bound the strategy refines to investigate the sub-strategy trees for both cases. If the `condition` is false, according to the main algorithm, the strategy stops searching into more detail. Thus, the Memory Bandwidth property needs to be replicated and coded into two properties:

- Compute Bound Property
- Memory Bound Property

The `id` as well as the `threshold` will be the same for both versions of the Memory Bandwidth property. However, the Memory Bandwidth Property designated to determine if a code is memory bound will return true from the `condition` method (line 15) if it is memory bound or false otherwise. Similarly, the Memory Bandwidth property for compute bound code will set the boolean flag to true when compute boundedness is detected, else false.

The implementation of the strategies is presented for two Intel processor architectures. In the following sections the properties which have very specific features due to differences within

the architecture are discussed. At the end of Sections 3.3 and 3.4 the strategy with an overview of the properties is shown. The remaining sections are dedicated to architecture independent properties.

3.3 Properties for the Westmere-EX Architecture

The Westmere-EX architecture [25] is a 10 core 32nm chip which belongs to the Intel Xeon Processor E7 family. This architecture has a peak performance of 9.6 GFlop/s per core, and features 30MB of cache. The architecture, also known as Intel Xeon E7-4870, runs at a maximum of 2.40 GHz without turbo mode with a Thermal Design Power (TDP) of 130W, and it can sustain up to 6.4 GT/s with the four Intel QPI links. The turbo mode allows the frequency to increase to 2.8GHz in fewer cores while keeping the TDP constant. The relevant features for code optimization are:

- The architecture decodes four x86 instructions per cycle.
- It uses the SSE4.2 instruction set that supports packed operations, particularly floating point operations.
- A crossbar is used to connect the last level caches and the memory controller.
- It supports the hyperthreading technology also known as Simultaneous Multi-Threading (SMT), which increases throughput by allowing the pipeline to be fed from two separate sets of hardware registers. The HEP-SPEC06 benchmark [57, 58] delivers around a 25% additional performance improvement thanks to the SMT technology. The architecture presents two hardware threads that are able to feed the pipeline as shown in Figure 3.10. While only hardware threads are shown in this figure, there is also the possibility of feeding more logical threads. The SMT feature can be deactivated, allowing the monitoring of eight events per core. However, when hyperthreading is activated, every thread has to keep track of their available counters and, even though performance counters can be measured per logical thread, the number of events to be measured at a time is halved.
- The hardware counters can be applied to different devices, making the collection of performance counters more complex compared to previous architectures (like in the case of Itanium architectures from Intel). There are events that can be measured per logical thread, others per core and finally per uncore device. The uncore devices are devices external to the core but are shared among the cores; for example: the Integrated Memory Controller (IMC), the Quick Path Interconnect (QPI), and the L3 cache [53].

Figure 3.9 shows a simplified diagram of the chip architecture. Micro-architectural events can be measured through the MSR kernel module available in Linux distributions. The MSR files are pseudo files which can be written with different masks and read after an elapsed time in order to access the event registers. Both uncore and core events are provided through the

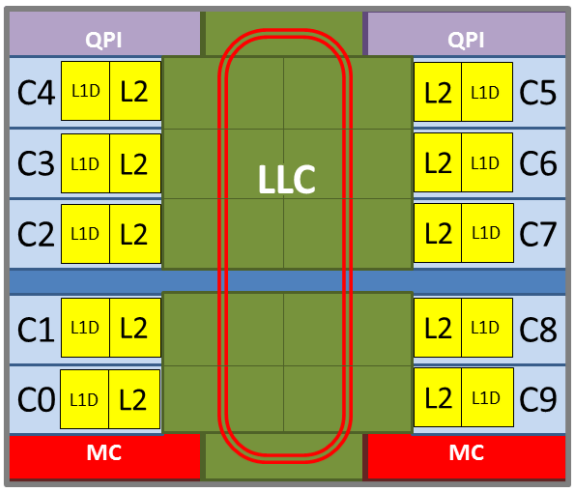


Figure 3.9: Westmere-EX Architecture

MSR files.

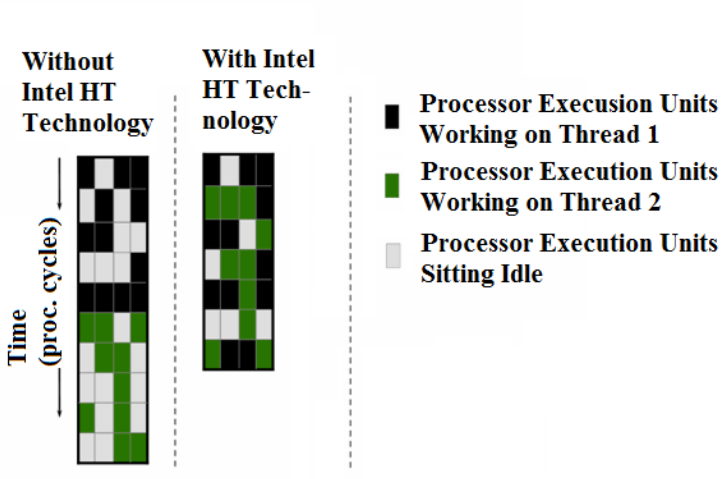


Figure 3.10: Pipeline with hyperthreading technology. Figure taken from [30].

The properties have been designed for the Westmere-EX architecture when the SMT is enabled. Due to space constraints in the formulae and long hardware counter names, many of them have been renamed. Every property is describe with five items: a general explanation with the hardware counters used, the domain where the property is measured or analyzed, how the property value is calculated, how the severity is calculated, and finally how the condition is calculated.

3.3.1 Floating Point Operations Rate Property

When defining flop/s it has been taken into account that this architecture has the ability of loading either four single precision floating point units or two double floating point units. These are measured with the following hardware events as defined in the developer's manual [4]:

- FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION hereafter called *SP*.
- FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION hereafter called *DP*.

The floating point operations are evaluated as scalar (only one floating point is executed) or packed (several floats are taken in one operation). These are:

- FP_COMP_OPS_EXE.SSE_FP_SCALAR hereafter called *Scalar*.
- FP_COMP_OPS_EXE.SSE_FP_PACKED hereafter called *Packed*.

Floating point operations are not measured as disjoint sets, which means that *SP* operations contain *Packed* and *Scalar* operations in single precision. *DP* operations also include the *Packed* and *Scalar* operations in double precision. On the other hand, *Scalar* operations include both single and double precision scalar operations, and analog to this, the *Packed* operations include both single, and double precision packed operations. Thus, there is no immediate way to recognize the real floating point operations without an estimation when performing black box monitoring.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value: Given that the floating point operations are not measured as disjoint sets in these events, an adjustment needs to be made to the proportions of each set in order to calculate the total floating point operations [15]. The scalar floating point operations are added to a weighted count of the packed floating point operations. The formula is as follows:

$$Flop/s = \frac{(4 \cdot SP + 2 \cdot kDP)}{SP + DP} \cdot \frac{Packed}{T} + \frac{Scalar}{T} \tag{3.3}$$

where T is the measuring time in seconds. This formula renders exact results if a code is only using either *SP* or *DP* operations without mixing them.
Severity: This property uses the severity formula `FORMULA2` (See formula 3.2).
Condition: This property is always collected. Therefore, the condition is always *true*.

3.3.2 Single Precision to Double Precision Flops Ratio Property

This property quantifies the ratio of single precision to double precision floating point operations. The analysis takes into account only the operations done together to not penalize twice the usage of double floating point operations.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are

added.

Property Value:

$$SP\_to\_DP\_ratio = \frac{SP}{DP} \tag{3.4}$$

Severity: This property uses the severity formula `FORMULA2` (See formula 3.2).

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.3 Packed to Scalar Flops Ratio Property

Like in the single precision to double precision flops ratio property, the analysis is done by considering the packed operations only once and which are performed simultaneously.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$PK\_to\_SC\_ratio = \frac{Packed}{Scalar} \tag{3.5}$$

Severity: This property uses the severity formula: `FORMULA2` (See formula 3.2).

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.4 Memory Bandwidth Properties

The Memory Bandwidth properties (memory bound and compute bound property) are essentially the same but have two different `condition` methods. The events to measure bandwith are only available per socket (uncore event). According to the Intel uncore manuals of the E7 processor family [53] the memory bandwidth can be measured with:

- The FVC_EV0_BBOX.CMDS_READS event counts the read commands from memory, hereafter $CmdReads$.
- The IMT_INSERTS_WR event counts the write inserts registered at the In Memory Table (IMT), hereafter $ImtWrites$.

Domain: Core level. The uncore events are weighted to obtain the memory bandwidth per core.

Property Value: The volume of data transfered (in Bytes) at the level of the node is as follows:

$$MemoryTransfer_{Node} = (CmdReads + ImtWrites) \cdot 64 \tag{3.6}$$

Here, the constant 64 is the size of the loaded bytes per read or write. Both events, $CmdReads$ and $ImtWrites$, are monitored per channels and all channels are added together to the entire node. Thus, to obtain a memory bandwidth per core, the L3_LAT_CACHE.MISS event has been used; an event which relates to a core, rather than to an uncore device. This event counts the cache misses on the last level cache and includes speculative traffic as documented in [4],

thus, it can be used to obtain a weight for the memory bandwidth used in a core. The ratio per core is obtained with the following formula for each core:

$$Device_i = \frac{L3\_LAT\_CACHE.MISS_i}{\sum_{devices}(L3\_LAT\_CACHE.MISS)} \tag{3.7}$$

The memory bandwidth of core i, is then calculated as:

$$MemoryBW = Device_i \cdot \frac{MemoryTransfer_{Node}}{T} \tag{3.8}$$

where T is the measuring time in seconds.
Severity: This property uses the severity formula: `FORMULA1` (See formula 3.1).
Condition: The condition of the Compute Bound replicate Property evaluates to *true* if and only if the memory bandwidth is smaller than or equal to the threshold. The condition of the Memory Bound replicate Property evaluates to *true* if and only if the memory bandwidth is greater than the threshold. Section 7.7 deals with how this threshold is set.

3.3.5 Intra-Node Load Imbalance Property

Load imbalance, in general, can be detected by examining the distance of the minimum value of a hardware counter and the maximum value of the same counter. Figure 3.11 shows six different cores used by an application, with measurements of the floating point operations per second in four different timestamps. Timestamp two through four have a load imbalance:

- two groups of cores are working with very different GFLOPS/s (monitoring interval 2);
- all cores have a different load (monitoring interval 3);
- and, at least one core has a different load from the rest (monitoring interval 4).

The only case on Figure 3.11 where no imbalance is present is on the first monitoring interval, the flops/s for all the cores is about the same rate. The rest of the cases show a common symptom, the lowest and the highest flop rate are *distant* to each other. The events involved can be selected according to the architecture. Flop/s reflects the load and work done in an arithmetic-intensive application and an ideal candidate for detecting load imbalance. However, if the architecture presents difficulties to correctly measure this metric, the instruction count would be an alternative for this property. The event e chosen, instructions or flops, for the calculation of intra-node load imbalance can be used to find the difference between the $e_{MAX} - e_{MIN}$, where $e_{MAX}, e_{MIN} \in E$, where E is the set of events for the chosen metric for all the devices belonging to the same node. Intra-node load imbalance can be calculated in this architecture by using $flops/s$ (this means the event chosen is $e = flops/s$).
Domain: Node level.
Property Value:

$$IntraNodeImbalance = flop/s_{MAX} - flop/s_{MIN} \tag{3.9}$$

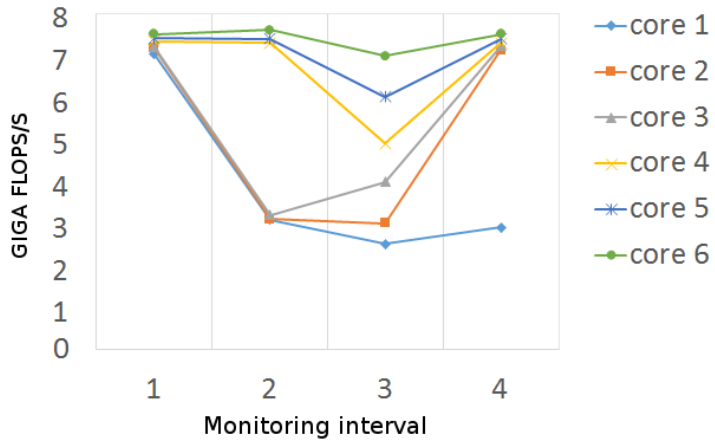


Figure 3.11: Detecting load imbalance with hardware counters

Severity: This property uses the severity formula: `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.6 Inter-Node Load Imbalance Property

Inter-node load imbalance can be detected by calculating the difference between $e_{MAX} - e_{MIN}$, where $e_{MAX} \in E_1$ and $e_{MIN} \in E_2$ and having $E_1 \neq E_2$, i.e. the maximum and the minimum do not belong to the same node, otherwise intra-node load imbalance would be detected. However, the algorithm to search for a maximum and a minimum will require decision trees if a maximum and minimum are found on the same node. A simpler approach to looking for the next maximum on another node, or the next minimum on another node is to calculate averages values at each node, i.e. μ_{E_i} where $i = 1, 2, 3, ..., N$ where N is the number of nodes in an application. Inter-node load imbalance can then be calculated by finding the difference between the maximum and the minimum average, i.e. $\mu_{MAX} - \mu_{MIN}$. If there is a measurement missing in a node, for example due to a counter overflow, the average hides this fact since the number of available observations to calculate the average is always taken and not the total number of devices. The average of a node will weaken the effects of a present intra-node load imbalance. Nonetheless, the previously defined property analyzes intra-node load imbalance and will detect this bottleneck.
Domain: Application level.
Property Value: Inter-node load imbalance can be calculated with the average of the $flops/s$ over a node, and comparing the averages of all nodes in a job, i.e. μ_{flops/s_i} where $i = 1, 2, 3, ..., N$ where N is the number of nodes in an application. The property value is then:

$$InterNodeImbalance = MAX(\mu_{flops/s}) - MIN(\mu_{flops/s}) \tag{3.10}$$

Severity: This property is always evaluated independent of the severity (uses no severity formula) given that inter-node imbalance is done at the post-mortem stage and not on-line.
Condition: The condition is always *true*.

3.3.7 Core Frequency Property

Monitoring the frequency of execution will provide insight to the usage of resources when correlated to the other properties. The frequency is measured with:

- The CPU_CLK_UNHALTED.CORE event counts the unhalted core cycles, i.e. the cycles where the core is active.
- The CPU_CLK_UNHALTED.REF event counts the unhalted reference cycles.

The events are available also at the level of the hyperthread (despite the name 'CORE' in the CPU_CLK_UNHALTED.CORE event).
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added. The cores in the same processor can't have different frequencies. However, the frequencies are taken for each core (even if they are the same in all the processor).
Property Value:

$$Frequency = \frac{CPU\_CLK\_UNHALTED.CORE}{CPU\_CLK\_UNHALTED.REF} \cdot F_0 \tag{3.11}$$

Where F_0 is the minimum available frequency provided by the architecture.
Severity: The severity formula is not used for this property as all values are collected.
Condition: The condition is always *true*.

3.3.8 Instruction Rate Property

The instruction rate on its own has also been defined as a property. The INST_RETIRED.ANY event counts the instructions which have been retired after execution.
Domain: Core level. The INST_RETIRED.ANY event from SMT threads that belong to a core are added together.
Property Value:

$$Instructions/s = INST\_RETIRED.ANY/T \tag{3.12}$$

where T is the measuring time in seconds.
Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is always *true*.

3.3.9 CPI Property

The clocks per instruction property is part of a set of widely used metrics for performance comparisons [88].
Domain: Core level. The counts of an event from two SMT threads that belong to a core are

added.

Property Value:

$$CPI = \frac{CPU\_CLK\_UNHALTED.CORE}{INSTR\_RETIRED.ANY} \tag{3.13}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1), except when CPI is *low* and the floating point operations rate is *low*. In this case then the severity formula of the flop/s is used.

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.10 Stall Cycles Property

In the Westmere-EX architecture the stall cycles property identifies the amount of cycles which were not dispatched to the execution units as a ratio to the number of all retired micro operations. The stalled cycles property is measured with:

- The UOPS_EXECUTED.PORT015_STALL_CYCLES event counts the micro operations which have stalled.
- The UOPS_RETIRED.ANY event counts all the micro operations.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$StallCycles = \frac{UOPS\_EXECUTED.PORT015\_STALL\_CYCLES}{UOPS\_RETIRED.ANY} \tag{3.14}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.11 Branch Misspredicted to Instructions Ratio Property

The branch misspredicted to the number of instructions is a ratio that provides insight into the usage of branches with respect to other instructions. The BR_MISP_RETIRED.ALL_BRANCHES event measures the mispredicted branches.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$BranchMisspredicion = \frac{BR\_MISP\_RETIRED.ALL\_BRANCHES}{INST\_RETIRED.ANY} \tag{3.15}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1)

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.12 Branch Misspredict ion to Branches Ratio Property

This property provides a more detailed analysis to the usage of branches than the previous property. The BR_INST_RETIRED.ALL_BRANCHES event measures the total amount of branches.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$BranchMisspredicion = \frac{BR\_MISP\_RETIRED.ALL\_BRANCHES}{BR\_INST\_RETIRED.ALL\_BRANCHES} \tag{3.16}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1)

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.13 Loads to Stores Ratio Property

The loads to stores ratio property is used to analyze to what extent the loaded data has been used to create "new" data to be stored. The property formula can be calculated from the following events:

- The MEM_INST_RETIRED.LOADS event that counts loads from memory, hereafter $MemLoad$.
- The MEM_INST_RETIRED.STORES event counts stores to memory, hereafter $MemStore$.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$LoadsToStoresRatio = \frac{MemLoad}{MemStore} \tag{3.17}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).

Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.14 L3 Cost Property

The last level cache cost property uses an estimate of the cost of the lost cycles when accessing the L3 cache. The latencies for transferring blocks from or to the L3 cache costs on average between 25 to 31 compute cycles as shown on Table 3.1. It is recommended in the Software Developer's Manual [4] to use 27 cycles as an average cost, which is taken into account in the property value formula. The MEM_LOAD_RETIRED.L3_UNSHARED_HIT event counts the L3 level cache hits, hereafter L3_Hits.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are

Cache level	Average cost of cycles
L1	6 cy
L2	12 cy
L3	25 - 31 cy
Memory	70 cy

Table 3.1: Westmere-EX: Cache latencies. Taken from the IA64 and IA32 Software Developers Manual [4].

added.
Property Value:

$$L3CyclesRatio = 27 \cdot \frac{L3\_Hits}{CPU\_CLK\_UNHALTED.CORE} \tag{3.18}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.15 L3 Misses to Instruction Ratio Property

If the ratio of L3 misses to instructions is *high* this means that L3 misses are significantly impacting the performance of the code. The MEM_LOAD_RETIRED.L3_MISS event counts the L3 cache misses, hereafter $L3\_Misses$.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$L3InstructionRatio = \frac{L3\_Misses}{INSTR\_RETIRED.ANY} \tag{3.19}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.16 L3 Bandwidth Property

For the L3 bandwidth property it is only necessary to calculate the bandwidth from two hardware events which counts the number of lines which have been evicted from and/or loaded to the L2 cache. These events are:

- The L2_LINES_IN.ANY event counts all the lines loaded into the L2 cache, hereafter L2_all.
- The L2_LINES_OUT.DEMAND_DIRTY event counts all the lines which have been evicted by demand, hereafter L2_dirty.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value: Each line transports 64 bytes, thus, the L3 bandwidth is given by:

$$L3Bandwidth = 64 \cdot \frac{(L2\_all + L2\_dirty)}{T} \tag{3.20}$$

where T is the measuring time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.17 L3 Hits to Misses Property

This property examines the amount of cache hits with respect to cache misses.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$L3HitstoMiss = \frac{L3\_Hits}{L3\_Misses} \tag{3.21}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.18 Expensive Instructions Property

The rate of expensive instructions property can be counted with the $ARITH.CYCLES\_DIV\_BUSY$ event. However, the event may produce wrong results when SMT is enabled [4].
Domain: Core level. The $ARITH.CYCLES\_DIV\_BUSY$ event from SMT threads that belong to a core are added together.
Property Value:

$$ExpensiveInstructions/s = \frac{ARITH.CYCLES\_DIV\_BUSY}{T} \tag{3.22}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.3.19 Strategy for the Westmere-EX

The designed properties for the Westmere-EX architecture are arranged into a hierarchy as illustrated on Figure 3.12. Given that a QPI transfer rate is not explicitly found on the uncore manual for the Intel Xeon Processor E7 family [53], the corresponding property was excluded from the strategy tree. The memory bandwidth strategy for compute bound and for memory

bound code both branch to a sub-strategy tree of floating point operations for analysis. The floating point operations are collected without filtering to see the performance of the entire machine. Due to the previous reasons, the sub-tree for floating point operations will be analyzed disconnectedly and independently of the memory bandwidth strategy and will become a root property itself.

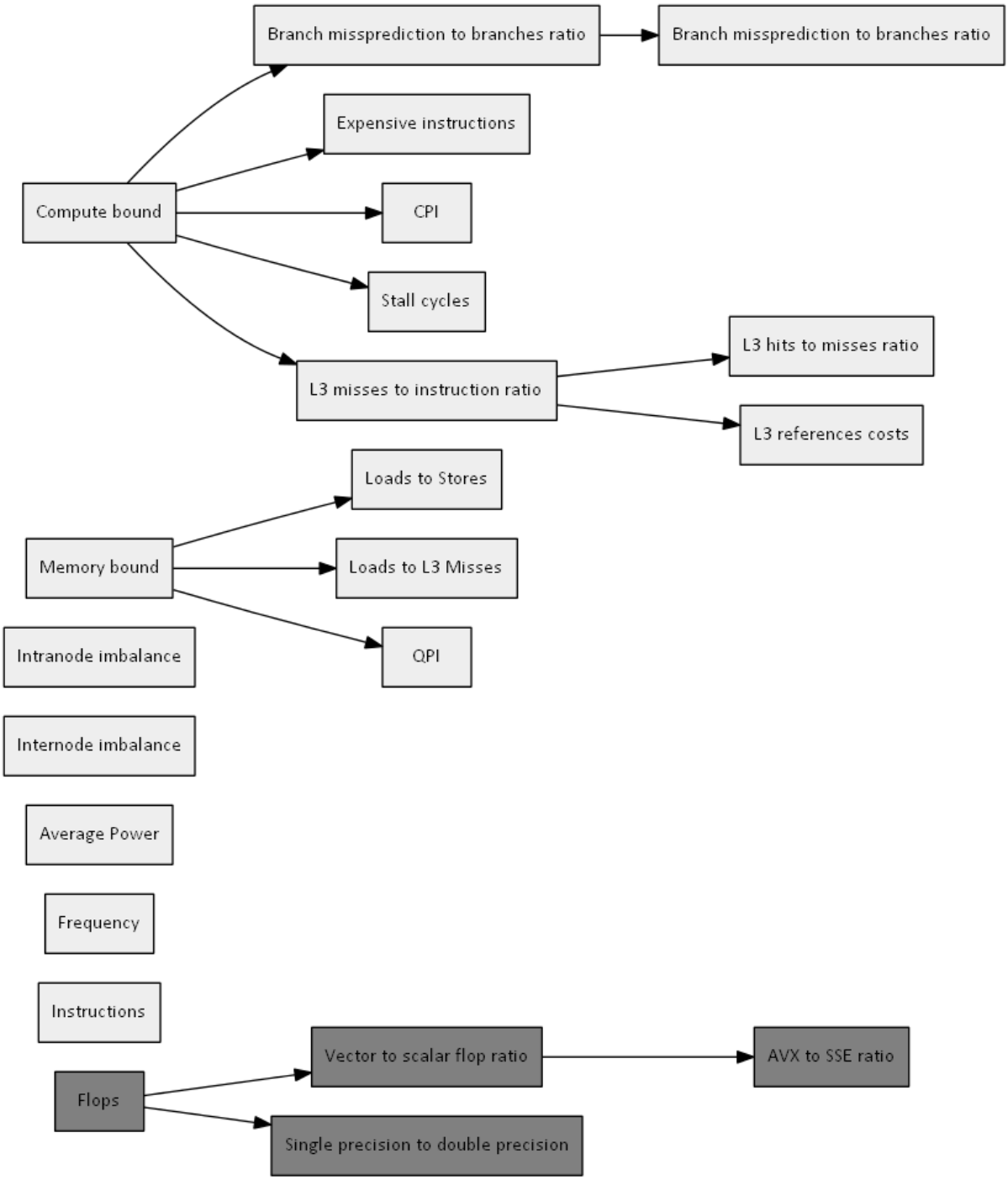


Figure 3.12: Strategy for the Westmere-EX Architecture

3.4 Properties for the Sandy Bridge-EP Architecture

The Sandy Bridge-EP micro-architecture [24], also known as Intel Xeon E5-2680, is an 8 core 32nm chip, has a base clock speed of 2.7 Ghz, 20M of L3 cache, and can sustain up to 8 GT/s with the four Intel QPI links. Like the Westmere-EX, it also uses the hyperthreading technology (SMT) and has a Thermal Design Power of 130W. The turbo mode allows the frequency to increase to 3.5GHz in fewer cores while keeping power within the designed envelope. The Sandy Bridge-EP introduced several new architectural features with respect to the Westmere-EX which require a different analysis than the preceding architecture. The L3 cache is distributed as segments at each core with a ring bus connecting all segments (Figure 3.13). This in turn reduces the penalty for last level cache transfers in comparison to the previous architecture (Table 3.2). Moreover, it has the Advanced Vector eXtensions (AVX) instruction set extension which uses a 256 bit SIMD register with twice the width of Streaming SMD Extensions (SSE) found in the Westmere-EX. An ADD and MULTIPLY pipelined instruction can be performed in one cycle. Table 3.2 shows the cache and memory latencies to move data to and from the registers.

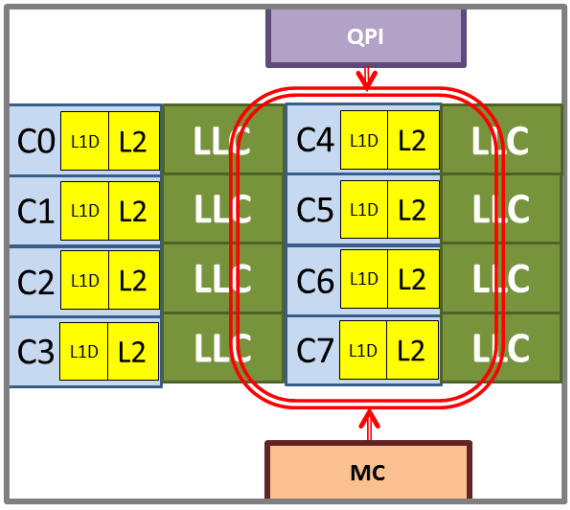


Figure 3.13: Sandy Bridge-EP Architecture. LLC stands for last level cache, also known as L3 cache.

Similar to the Westmere-EX architecture there are several devices that can be monitored: hardware thread, core (which contains two hardware threads), and uncore events which are the shared devices among the cores [54]. All the properties were designed considering SMT to be activated.

3.4.1 Floating Point Operations Rate Property

Floating point operations are flawed on this architecture [19, 111], the percentage of error to the number of flops oscillates, to up to a factor of six, making the flop count unreliable. The counters increase when the flop instruction is issued and not on retirement. So re-issues, due to

Cache level	Average cost of cycles
L1 Cache	6 cy
L2 Cache	12 cy
L3 Cache	25 - 31 cy
Memory	50 cy

Table 3.2: Sandy Bridge-EP: Cache latencies. Software Developer's Manual [4]

input arguments not being ready, will produce over counting of the event. The events provided to measure the flops are:

- The FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE event which counts the double precision floating point packed SSE operations, hereafter $Packed\_DP$. The architecture is capable of performing 2 floating operations of this kind at a time.
- The FP_COMP_OPS_EXE_SSE_FP_PACKED_SINGLE event which counts the single precision floating point SSE operations, hereafter $Packed\_SP$, the architecture can perform 4 flops of this kind at a time.
- The FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE event counts the double precision floating point operations, hereafter FP_DP.
- The FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE event counts the single floating point operations, hereafter FP_SP.
- The FP_256_PACKED_SINGLE event counts AVX floating point operations which corresponds to the eight packed flops at single precision, hereafter FP_SP_{AVX}.
- The FP_256_PACKED_DOUBLE event counts AVX floating point operations which corresponds to four packed flops at double precision, hereafter FP_DP_{AVX}.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value: Thus the counting of flops is expressed as:

$$Flops/s = (2 * Packed\_DP + 4 * Packed\_SP + FP\_DP + FP\_SP + 8 * FP\_SP_{AVX} + 4 * FP\_DP_{AVX})/T \quad (3.23)$$

where T is the measuring time in seconds. In contrast to the previous architecture, the events have been provided as disjunct sets. Thus, it is only necessary to multiply the number of real floating point operations and the number of counts in the event. In practice, counting the number of floating point operations presents one difficulty. When the Sandy Bridge-EP architecture uses the hyperthreading SMT technology, it is only possible to read four programmable counters at a time. Thus, to read the number of flops, it is necessary to apply either multiplexing or read them one after the other. Both cases are an estimation in the case of black box monitoring.

The PAPI tool [79] provides a generic interface to all architectures and, for the case of the Sandy Bridge-EP architecture, they are bound to only measure four counters. The PAPI interface only measures the amount of packed operations, not taking into account that one packed operation includes several real floating point operations. The interface adds this to the scalar operations, without including the AVX operation (i.e. the events FP_256_PACKED_SINGLE and FP_256_PACKED_DOUBLE are excluded) so a vectorized AVX code will deliver zero flops with this interface. Even though this is documented [6], the philosophy of having one interface for all architectures breaks with such an example. This is a reason why a generalization with a single interface to be used in many architectures should be avoided, and the events that contribute to a desired metric should be carefully studied to design a property.
Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: This property is always collected. Therefore, the condition is always *true*.

3.4.2 Vectorized to Scalar Ratio Property

The child property to the flops count is the ratio of all vectorizations, including AVX and SSE, with respect to the scalar operations. The following definitions are needed for this property:

$$vector = FP\_SP_{AVX} + FP\_DP_{AVX} + Packed\_DP + Packed\_SP \tag{3.24}$$

$$scalar = FP\_DP + FP\_SP \tag{3.25}$$

The amount of flops that are calculated at once are not considered in this ratio, otherwise the penalty for not using SSE and AVX increases.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$VectorToScalar = \frac{vector}{scalar} \tag{3.26}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.4.3 AVX to SSE Ratio Property

The child property AVX to SSE ratio is also a function of the hardware events that count flops. Two definitions are necessary for this property:

$$avx = FP\_SP_{AVX} + FP\_DP_{AVX} \tag{3.27}$$

$$sse = Packed\_DP + Packed\_SP \tag{3.28}$$

Like in the previous property, the amount of operations which are calculated should be left out to avoid a double penalty.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$FlopsAVX\_SSEratio = \frac{avx}{sse} \tag{3.29}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.4.4 Single Precision to Double Precision Property

The child property analyses the single precision to double precision ratio. The following definitions are needed for this property:

$$FP_{DP} = Packed\_DP + FP\_DP + FP\_DP_{AVX} \tag{3.30}$$

$$FP_{SP} = Packed\_SP + FP\_SP + FP\_SP_{AVX} \tag{3.31}$$

Like in the previous property, the amount of operations which are calculated should be left out to avoid a double penalty.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$FlopsSPtoDPRatio = \frac{FP_{SP}}{FP_{DP}} \tag{3.32}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.4.5 Core Frequency Property

The property for monitoring the frequency is the same one as shown in formula 3.11, and uses the same hardware events. It is known that in x86_64 architectures, the memory performance is influenced by the clock speeds [101]. In this architecture, the best memory bandwidths are obtained with the highest frequencies and a high number of cores, but using only one thread per core.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$Frequency = \frac{CPU\_CLK\_UNHALTED.CORE}{CPU\_CLK\_UNHALTED.REF} \cdot F_0 \tag{3.33}$$

Where F_0 is the minimum available frequency provided by the architecture.
Severity: The severity formula is not used for this property as all values are collected.
Condition: The condition is always *true*.

3.4.6 Instruction Rate Property

The instruction rate can be done in the same way as formula 3.12 (with the same events as in the Westmere-EX architecture).
Domain: Core level. The INST_RETIRED.ANY event from SMT threads that belong to a core are added together.
Property Value:

$$Instructions/s = INST\_RETIRED.ANY/T \tag{3.34}$$

where T is the measuring time in seconds.
Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is always *true*.

3.4.7 CPI Property

The CPI count can be done in the same way as formula 3.13 and with the same events as in the Westmere-EX architecture.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$CPI = \frac{CPU\_CLK\_UNHALTED.CORE}{INSTR\_RETIRED.ANY} \tag{3.35}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1), except when CPI is *low* and the floating point operations rate is *low*. In this case then the severity formula of the flop/s is used.
Condition: The condition is *true* if and only if the severity is greater than zero.

3.4.8 Memory Bandwidth Property

Analogously to the Westmere-EX architecture, this property is replicated to analyze compute bound and memory bound code. The properties compute exactly the same property value and severity, only the condition changes.
Domain: Core level. The uncore events are weighted to obtain the memory bandwidth per core.
Property Value: The memory bandwidth can be calculated by adding the memory events at the four memory controller channels. The volume of data (in Bytes) can be calculated with:

$$MemoryTransfer_{Node} = (CAS\_COUNT.RD + CAS\_COUNT.WR) \cdot 64 \tag{3.36}$$

The CAS_COUNT events for read and write (extension .RD and .WR) are monitored per channel and all the channels are added together. This will provide, however, a unique event for the entire node. Channels can't be assigned uniquely to cores, since they are shared resources.

Thus, to obtain a memory bandwidth per core, the L3_LAT_CACHE.MISS event has been used; an event which relates to a core, rather than to an uncore device. This event counts the cache misses on the last level cache and includes speculative traffic as documented in [4], thus, it can be used to obtain a percentage of the memory bandwidth used in a core. The ratio is obtained with the following formula:

$$Device_i = \frac{L3\_LAT\_CACHE.MISS_i}{\sum_{devices}(L3\_LAT\_CACHE.MISS)} \tag{3.37}$$

The memory bandwidth of core i, is then calculated as:

$$MemoryBW = Device_i \cdot MemoryTransfer_{Node}/T \tag{3.38}$$

where T is the measuring time in seconds.
Severity: This property uses the severity formula: FORMULA 1 (See formula 3.1).
Condition: The condition of the Compute Bound replicate property evaluates to *true* if and only if the memory bandwidth is smaller than or equal to the threshold. The condition of the Memory Bound replicate property evaluates to *true* if and only if the memory bandwidth is greater than the threshold.

3.4.9 Intra-node Load Imbalance Property

Intra-node load imbalance is calculated as described in Section 3.3 by using the difference between the maximum and the minimum of an event e. The rate of floating point operations is not used, due to the known overcounting of these operations. Even if the floating point operations would be exact, the six events can't be measured together, so only an estimation is available. The event chosen is, therefore, the *intruction* rate for this architecture.
Domain: Node level.
Property Value:

$$IntraNodeImbalance = instr/s_{MAX} - instr/s_{MIN} \tag{3.39}$$

where $instr/s_{MAX}, instr/s_{MIN} \in I_{Node}$ where I_{Node} is the set of *instruction* rates for the chosen metric for all the devices belonging to the same node.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the severity is greater than zero.

3.4.10 Inter-node Load Imbalance Property

Inter-node load imbalance can be detected by calculating the average of the instructions over a node, and comparing the averages of all nodes in a job, i.e. μ_{I_i} where $i = 1, 2, 3, ..., N$ where N is the number of nodes in an application, and by finding the distance of the minimum and the maximum average, i.e. $\mu_{MAX} - \mu_{MIN}$. Using flop/s is not the best choice, see the previous Property (Intra-node Load Imbalance Property).

Domain: Application level.
Property Value:

$$InterNodeImbalance = MAX(\mu_{I/s}) - MIN(\mu_{I/s}) \tag{3.40}$$

Severity: This property is always evaluated independent of the severity (uses no severity formula) given that inter-node imbalance is done at the post-mortem stage and not on-line.
Condition: The condition is always *true*.

3.4.11 Branch Misspredections to Instructions Ratio Property

The branch misspredictions to instructions ratio property reflects the rate of branch misspredictions to the total number of branches.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$BranchMissesToInstructions = \frac{BR\_MISP\_RETIRED.ALL\_BRANCHES}{INST\_RETIRED.ANY} \tag{3.41}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.12 Branch Misspredections to Branches Ratio Property

The branch misspredictions property reflects the rate of branch misspredictions to the total number of branches.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$BranchMisspredictionToBranches = \frac{BR\_MISP\_RETIRED.ALL\_BRANCHES}{BR\_INSTR\_RETIRED.ALL\_BRANCHES} \tag{3.42}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.13 Expensive Instructions Property

A long latency instruction count was not explicitly defined in the list of hardware events of this architecture. It was found, however, that the ARITH.FPU_DIV_ACTIVE event is useful for counting division related executions.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are

added.

Property Value:

$$ExpensiveInstructions/s = ARITH.FPU\_DIV\_ACTIVE/T \tag{3.43}$$

where T is the measurement time in seconds.

Severity: The severity formula used is `FORMULA1` (See formula 3.1).

Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.14 Loads to L3 Cache Misses Ratio Property

This property analyzes the ratio of loads to L3 cache misses.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$LoadsToMisses = \frac{MEM\_UOP\_RETIRED.LOADS}{MEM\_LOAD\_UOPS\_MISC\_RETIRED.LLC\_MISS} \tag{3.44}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).

Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.15 Loads To Stores Ratio Property

This property analyzes the ratio of loads to stores.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$LoadsToStores = \frac{MEM\_UOP\_RETIRED.LOADS}{MEM\_UOP\_RETIRED.STORES} \tag{3.45}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).

Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.16 L3 Cost Property

The last level cache cost uses, in the case of the Sandy Bridge-EP architecture, 26 cycles as an average for each cache miss (see Table 3.2). It uses the MEMLOAD_UOPS_RETIRED.LLC_HIT event, hereafter $L3Hits$.

Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.

Property Value:

$$L3CyclesRatio = 26 \cdot \frac{L3Hits}{CPU\_CLK\_UNHALTED.CORE} \tag{3.46}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.17 L3 Misses to Instruction Ratio Property

The L3 cache misses to the retired instructions ratio helps determine whether L3 misses are dominating the inefficiencies in the code. The MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS counts the number of L3 misses, hereafter $L3Misses$.
Domain: Core level. The counts of an event from two SMT threads that belong to a core are added.
Property Value:

$$L3InstructionRatio = \frac{L3Misses}{INSTR\_RETIRED.ANY} \tag{3.47}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.18 L3 Bandwidth Property

L3 bandwidth is calculated with the L2 lines which were loaded and the L2 evicted lines, similar to the Westmere-EX property. Every line contains 64 bytes. For the property value formula, the following hardware events have been used:

- L2_LINES_IN.ALL event counts the loaded lines to L2, hereafter $L2all$.
- L2_LINES_OUT.DEMAND_DIRTY event counts the evicted lines, hereafter $L2dirty$.

Domain: Core level. An event from two SMT threads that belong to a core is added together.
Property Value:

$$L3Bandwidth = 64 \cdot (L2all + L2dirty)/T \tag{3.48}$$

where T is the measuring time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.19 L3 Hits to Misses Ratio Property

This property examines the ratio of L3 cache hits with respect to L3 cache misses.
Domain: Core level. An event from two SMT threads that belong to a core is added together.
Property Value:

$$L3HitstoMiss = \frac{L3Hits}{L3Misses} \tag{3.49}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.20 QPI Property

The QPI Property for the Sandy Bridge-EP architecture can be measured in levels. The counter involved is the QPI_RATE_STATUS and the last three bits of this counter provides information of the GT/s level [54]. For example: 010 corresponds to a rate of 5.6 GT/s and 011 corresponds to 6.4GT/s.
Domain: Socket level.
Property Value:

$$QPI = decode(QPI\_RATE\_STATUS) \tag{3.50}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is set to *true* if and only if the severity is greager than zero.

3.4.21 Package Power Property

The measurement of average power on the measurement interval is available through the Running Average Power Limit, RAPL, counters. The RAPL counters provide an estimation of energy performance with some shortcomings; for instance, the wraparound overflow time is approximately every 60 seconds and not every component of the motherboard can be measured [102]. The Sandy Bridge-EP architecture has counters for measuring the DRAM, the package, and the power plane 0 and 1 [83].
Domain: Socket level.
Property Value:

$$PowerProcessor = POWER\_PKG.WATT \tag{3.51}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is always set to *true*.

3.4.22 DRAM Power Property

The measurement of average DRAM power on the measurement interval is available through the Running Average Power Limit, RAPL, counters.
Domain: The domain is at the level of a DRAM pertaining to one socket.
Property Value:

$$PowerDRAM = POWER\_DRAM.WATT \tag{3.52}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is always set to *true*.

3.4.23 Strategy for the Sandy Bridge-EP Architecture

The designed properties for the Sandy Bridge-EP architecture are arranged into a hierarchy as illustrated on Figure 3.14. Like in the strategy designed for the Westmere-EX, the floating point operation analysis has been assigned as root properties. The floating point operations are collected without filtering to see the performance of the entire machine.

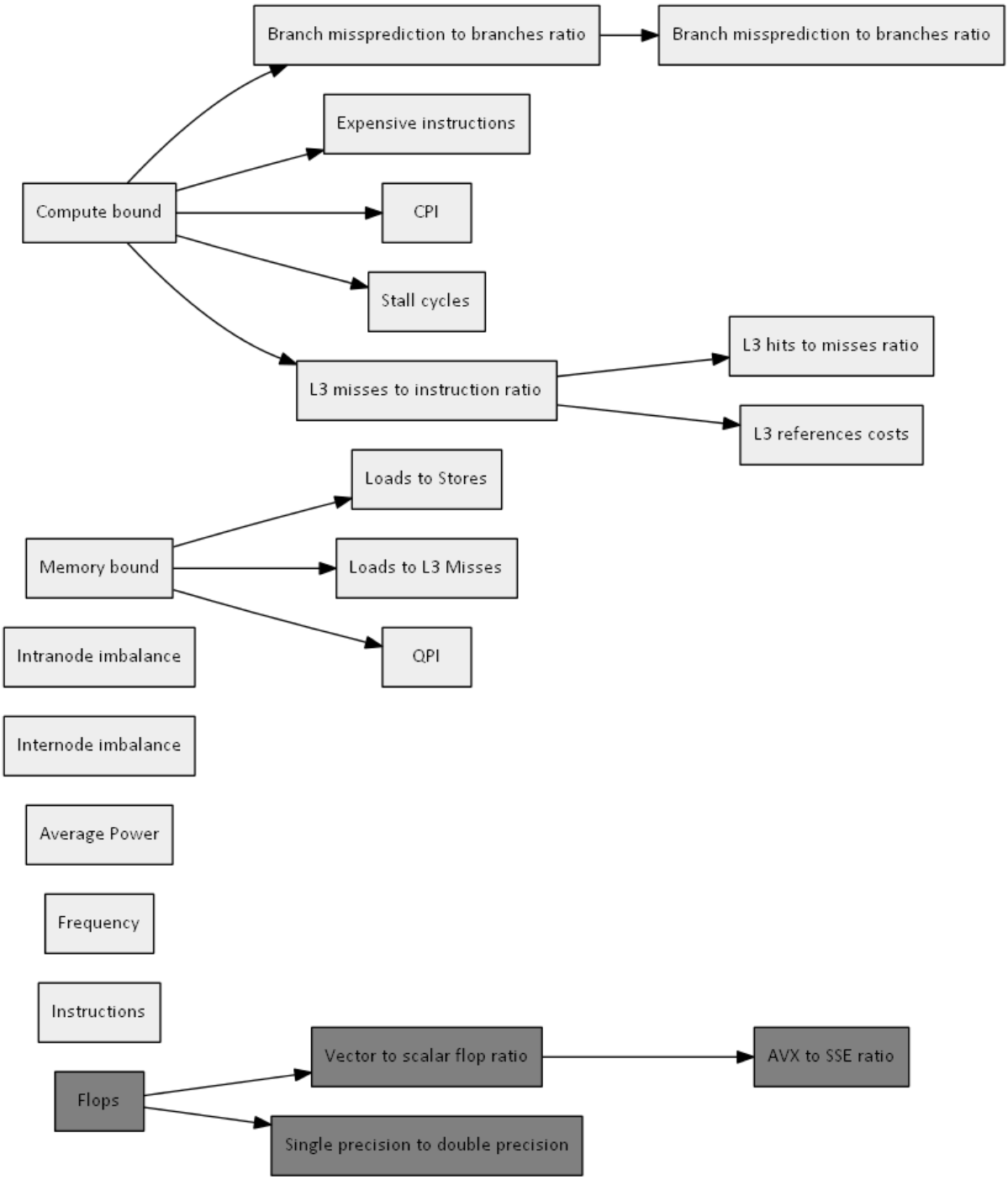


Figure 3.14: Strategy for the Sandy Bridge-EP Architecture

3.5 Architecture independent Properties

The monitoring of non-architectural dependent properties can be done by reading operating system specific information or via other tools which provide measurement on other components, like file system and network traffic. The following properties are also part of the monitoring scheme.

3.5.1 Memory Property

The memory usage can be parsed value from the kernel's virtual system: /proc/meminfo. This property helps detect if a memory leak is present.
Domain: Node level.
Property Value:

$$MemoryUsed = MemTotal - MemFree \tag{3.53}$$

Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is always set to *true*.

3.5.2 User Percent Property

The System Activity Report (SAR) [33] utility can be used to monitor real time performance giving a percentage of use of a resource. An advantage of the SAR tool is that it does not conflict when two measurements concur. The measurement is based on the Linux standard commands which can be measured across all micro-architectures as long as they are running on a Linux operating system. The definition of this property is: "Percentage of CPU utilization that occurred while executing at the user level (application). Note that this field includes time spent running virtual processors." [33]
Domain: Core level. The $User\%$ event from two SMT threads that belong to a core is added together.
Property Value:

$$USERPCT = User\% \tag{3.54}$$

Severity: The property uses a simplified severity formula and is defined as:

$$s = 1 - \frac{USERPCT}{t} \tag{3.55}$$

where t is a threshold, s is the severity, such that the higher the percentage of the resource is used, the better the performance is.
Condition: The condition is always set to *true*.

3.5.3 System Percent Property

The system percent is another measurement taken from the SAR tool. The event is defined as: "Percentage of CPU utilization that occurred while executing at the system level (kernel). Note that this field includes time spent servicing interrupts and softirqs." [33]
Domain: Core level. The $System\%$ event from two SMT threads that belong to a core is added together.
Property Value:

$$SYSTEMPCT = System\% \tag{3.56}$$

Severity:

$$s = \frac{SYSTEMPCT}{t} \tag{3.57}$$

where t is a threshold, and s is the severity. The application should try to avoid as much as possible using system calls.
Condition: The condition is always set to *true*.

3.5.4 I/O Wait Percent Property

The I/O wait is the third measurement taken from the SAR tool. The event is defined as: "Percentage of time that the CPU or CPUs were idle during which the system had an outstanding disk I/O request." [33]
Domain: Core level. The $IOWait\%$ event from two SMT threads that belong to a core is added together.
Property Value:

$$IOWAITPCT = IOWait\% \tag{3.58}$$

Severity:

$$s = \frac{IOWAITPCT}{t} \tag{3.59}$$

Where t refers to a threshold and s to the severity.
Condition: The condition is always set to *true*.

3.5.5 I/O Properties

Since I/O reading and writing to a file system is one of the slowest transfers of data. It has been observed that the values of this metric are always very small (typically, less than 3%) even if an application is performing intensive I/O. Other properties have to be considered to complement this property.
Other I/O properties are not dependent on the hardware architecture but on the underlying file system and available tools for measuring metrics. Typically, available information to be monitored includes bandwidth used for reading and writing. The following properties have been designed by using the metrics from the *mmpmon* [51] tool; a tool used by GPFS filesystems.

3.5.6 I/O Read Bandwidth Property

The I/O read bandwidth is measured by taking the total read bytes and dividing them by the basic measurement time.
Domain: Node level.
Property Value:

$$ReadBandwidth = Read\ Bytes/T \tag{3.60}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the property value is greater than zero (I/O read operations are being performed).

3.5.7 I/O Write Bandwidth Property

The I/O read bandwidth is measured by taking the total written bytes and dividing them by the basic measurement time.
Domain: Node level.
Property Value:

$$WriteBandwidth = Written\ Bytes/T \tag{3.61}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the property value is greater than zero (I/O write operations are being performed).

3.5.8 I/O Mean Read Request Size Property

This property analyzes the I/O mean read request size.
Domain: Node level.
Property Value: The property value can be calculated by dividing the I/O read bytes by the total amount of read operations.

$$ReadBWPerOp = \frac{ReadBandwidth}{Reads} \tag{3.62}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.9 I/O Mean Write Request Size Property

This property analyzes the I/O mean write request size.
Domain: Node level.

Property Value: The property value can be calculated by dividing the I/O read bytes by the total amount of write operation.

$$WriteBWPerOp = \frac{WriteBandwidth}{Writes} \tag{3.63}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.10 Number of File Open Operations Property

This property monitors the number of opens done in a second, a property which reflects the number of times when meta-data has been requested. Even though the values are given per second, it is not the intention to measure how fast this rate is, but if the number of open requests is high.
Domain: Node level.
Property Value:

$$Opens/s = IO\_Opens/T \tag{3.64}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.11 Number of File Close Operations Property

This property monitors the number of closes done in a second. Even though the values are given per second, it is not the intention to measure how fast this rate is, but if the number of close requests is high.
Domain: Node level.
Property Value:

$$Closes/s = IO\_Closes/T \tag{3.65}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA1` (See formula 3.1).
Condition: The condition is *true* if and only if the property value is greater than zero.

—

The following properties are related to the network usage of an InfiniBand fabric, and are based on the *perfquery* command line tool [92].

3.5.12 Transmission Bandwidth Property

This property monitors the transmitted network bytes per second (network bandwidth).
Domain: Node level.
Property Value:

$$XmitBW = Transmitted\ Bytes/T \tag{3.66}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.13 Received Bandwidth Property

This property monitors the received network bytes per second (network bandwidth).
Domain: Node level.
Property Value:

$$RecvdBW = Received\ Bytes/T \tag{3.67}$$

where T is the measurement time in seconds.
Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.14 Mean Transmitted Bytes per Packet Property

This property analyzes the mean transmitted bytes per packet.
Domain: Node level.
Property Value:

$$XmitBytesPerPacket = \frac{XmitBytes}{Packets} \tag{3.68}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.15 Mean Received Bytes per Packet Property

This property analyzes the mean received bytes per packet.
Domain: Node level.
Property Value:

$$RcvdBytesPerPacket = \frac{RcvdBytes}{Packets} \tag{3.69}$$

Severity: The severity formula used is `FORMULA2` (See formula 3.2).
Condition: The condition is *true* if and only if the property value is greater than zero.

3.5.16 Strategy for Micro-Architecture Independent Properties

The micro-architecture independent properties are evaluated at the node as root properties. The parent-child relations among the properties are shown in Figure 3.15.

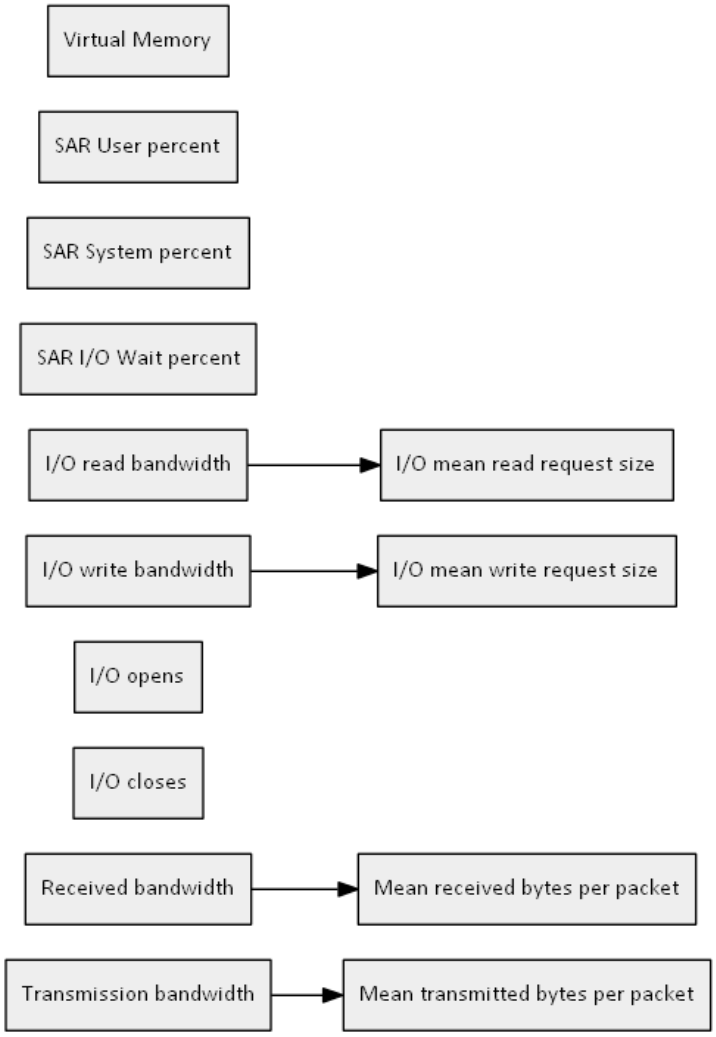


Figure 3.15: Strategy for architecture independent properties

4

Functionality of the PerSyst Tool

This chapter describes the functionality of the agent types in the hierarchy such as the communication interaction, the cycle control, and the transport of properties. The agent hierarchy is introduced in the following section. Section 4.2 describes the main functionality of the agents and the communication among them, including the framework implemented solutions for time control, concurrent measurements, and failure recovery.

4.1 Agent Hierarchy

The PerSyst Tool has been developed as distributed software with a tree agent hierarchy. The three types of agents are the synchronization agent, or SyncAgent; the Collector agent; and the PerSyst agent, as shown in Figure 4.1. The main functionalities of the SyncAgent are to synchronize measurement, the Collector agents collect the performance data, and the PerSyst agents perform the measurements. All of the agents comprise the transport system for analyzed performance data.

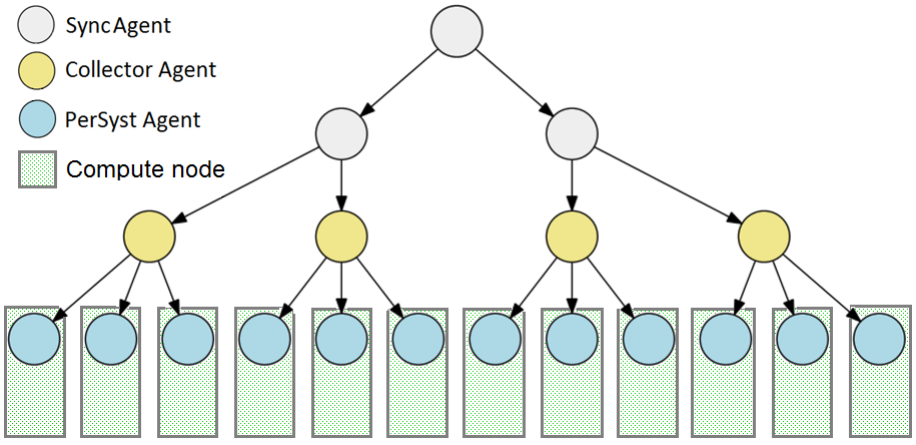


Figure 4.1: Agent Hierarchy

The software is designed to have one layer of PerSyst agents, one layer of Collector agents which manage the PerSyst agents, and there is at least one SyncAgent as the frontend. The term

middle layers will be used in this chapter for all the agents which are between the frontend and the PerSyst agents. For systems in the range of hundreds of thousands of cores, it is necessary to include more layers of SyncAgents in the middle layers of the hierarchy (as shown in Figure 4.1).

4.2 Agent Functionality

The main functionality of the PerSyst Tool is to analyze, filter, collect, and aggregate performance data. The aggregation uses a fixed number of quantiles for all cores in a job. It is known that aggregation of subsets of quantiles is only possible in special cases, refer to Section 5, and can only be done using estimations of the accumulated distributed frequency of each subset. Thus, two types of aggregations are used: estimation of quantiles or exact calculation. The SyncAgents only perform estimation of quantiles, while the Collector Agents and PerSyst Agents perform exact calculation of quantiles. Detailed functionality regarding quantile aggregation is provided in Chapter 5.

All types of agents are daemonized[1] and implement a continuously running loop that can react to inbound communication as well as to initiate communication with other agents. Agents are uniquely identifiable by the pair hostname-port. However, to simplify the handling, the agents receive a unique number, hereafter the tag.

4.2.1 SyncAgent

The SyncAgent can be used as the frontend[2] of the entire tree. It can also be used in the subsequent layers of the tree hierarchy below the frontend, as shown in Figure 4.1. The SyncAgent can be set with a flag to be the frontend. This agent triggers the measuring cycle and controls the agent tree. The cycle control for defining the measuring and idles times is held at this agent.

The frontend itself does not perform any measurements and should be released on another node which is not monitored to avoid further interference with running applications. The frontend reads at initialization the configuration files. The topology file, a file with the agent tree layout and network information of each agent, is also read at the frontend such that no other agent is required to read a file. The reading of a file in a file system from thousands of processes trying to connect to the same device is not scalable. When the frontend reads the configuration file it can also load the topology file. The hosts where each agent are to be distributed can be configured in the topology file. This file is in XML format and contains the parent-child relations of the agent network, additional information that will enable the communication between agents (a port if a fixed port is needed), and an agent identification number.

Once the topology file is read and the next layer of child-agents are spawned, the agent will

[1] Daemonized means that the process changes its unix session id in oder to be a child process from pid 1 (the root process).

[2] The term frontend is used as described by Gerndt et al [39], and refers to the root agent of the agent tree.

send the topology information of interest to a child agent and not the network information of the entire agent tree. Thus, a child agent will only know its own sub-tree; in this way reducing the memory needed at each agent. This is a process that repeats itself in a top-down fashion such that every time the sub-tree is started and configured downwards the topology information (parent-child relations) that is transmitted becomes smaller. Every agent has a registry of the connection details (port, host, and tag) of all agents in their sub-tree. Thus, it is required that agents at the top of the tree get enough memory space and should not reside in compute nodes where running applications may need the available memory space.
The agent start up is also left out of the framework and can be tailored for each type of agent. Agent start up is only performed by the SyncAgent (in both cases, when acting as a frontend or when placed in the middle layers), and by the Collector Agents. The possibilities to place an agent are varied. Examples are:

- Placement via an ssh command.
- Start up through the xinetd superserver [115].

While the first variant requires a configuration of environment variables at the moment of placing the agent, the latter depends upon a system wide configuration at all nodes of the xinetd servers.

When job information is available on the node were the frontend is running, the interface can be implemented to collect the job information at the SyncAgent since it has the capability of transmitting this information. The job information is split and sent to the child agents. A sub-tree of the agent hierarchy monitors only a subset of jobs. Thus, the splitting of the job information is done by sending only the pertinent information which the sub-tree requires. The receiving agent will only have the information of its sub-tree, such that the memory footprint of all the agents is kept negligible.

The distribution of the properties from each job to the Collector agents is referred to as *job balancing*: the amount of performance data from the monitored jobs that each Collector receives. The load that reaches the Collectors is distributed evenly to avoid having a bottleneck in one Collector. The implementation of the job balancing algorithm can be done at the frontend. It is recommended to do so if job information is available at this point. If this is the case, the SyncAgent calculates the *route* on the agent tree where the performance data of each job will be sent. Details on how the job balancing is implemented and how this route is defined are provided in Chapter 5.

Subsequent layers of SyncAgents send the synchronization commands and other information. In such a case only some core functionality is used to achieve this tasks. When a SyncAgent is placed in the middle layers and receives job information of the parent agent, it will split the job information to send it to its child agents. Layers of synchronization agents which do not act as frontend will have the cycle control inactivated as they only forward the commands of the frontend.

4.2.2 Collector Agent

The Collector Agent has as its main functionalities the propagation of the measuring command and receive the incoming properties in order to aggregate them. At the beginning of the execution, the Collector Agent receives the topology information of its sub-tree and spawns the child agents that are under its control. No topology information of the agent tree is sent to the leaves of the tree except the network information concerning the Collector that will receive the properties. The Collector Agent forwards the measuring commands to the PerSyst Agents. In the usual procedure, the PerSyst Agent will send the Properties as a response to the measuring command to another Collector or to the parent Collector.

A top-down control of agents, by sending the command through the tree structure of the agents, can be done just like other hierarchical tools. However, the collection of data from the bottom-up was readjusted in order to try to avoid quantile estimation (see Chapter 5 on quantile estimation) of subsets as opposed to quantile calculation to avoid further data quality degradation. The Collectors receive, in most of the cases, the entire property information of a job. If this is the case, they perform the aggregation and can perform the output according to the chosen implementation. At the end of the measuring cycle the Collector processes the properties to quantiles and averages them.

4.2.3 PerSyst Agent

The PerSyst Agent is the measuring agent at the compute nodes. They listen permanently at a given port (dynamically set or a fixed configured port) for instructions from parent agents before proceeding to measure their local nodes. When the measurement command is received, the PerSyst Agent has a preparation phase prior to measurement. If the batch scheduler has the job information available at the node, this information can be parsed at the level of the PerSyst Agent on this preparation phase. Otherwise, the agents can be configured to pass the job information and the PerSyst Agents receive the information during this phase in order to avoid massive requests to a file system. This information includes the network information of the recipient Collector who will receive the performance data.
The hardware events used by all properties are collected and passed to the measuring tool. The measuring tool is implemented outside of the framework and performs the measurement of the devices. Once the measurement is done, the Strategy will go through the tree of properties that will perform the actual analysis. The properties which are candidates to be reported are then sent to the recipient Collector.

The amount of information that the PerSyst Agent receives from the Collector Agent is kept minimal to avoid using unnecessary memory space. The PerSyst Agent has, therefore, no registry of the agent tree. In addition to the measuring strategy, the data sent by the Collector Agent includes the communication information (the port and host) of the recipient Collector

within the *route* information. The route can also specify whether the agent itself can aggregate the properties and perform the output. If the PerSyst Agent has to send its data, it tries to send it whithin the monitoring interval with a maximum of three trials to ensure that the message arrives. After a cycle is completed, the PerSyst Agent will delete this information and await the measurement command for the next timestamp. In the next cycle, the PerSyst Agent can receive a different host and port to send the performance data. The only information kept is the host and port of the parent. When a Collector has to be restarted because of failure, the Collector information is sent to all the children agents; the Collector network information is the only persisting information in a PerSyst Agent.
The measurement tools (such as LIKWID [113] and pfmon [3]) can be called from this agent either by invoking the tool through a system call, or internally with a library call. When both are available the choice is clear. Parsing the output data of a program invocation as opposed to a library call is more expensive and time consuming.

The configuration of the severities and severity formulae can be done at the level of the PerSyst Agent by reading a file. However, this feature is only available for clusters where reading a file from all the agents is not a problem and does not congest the entire system. For larger clusters, like petaflop systems, this feature can be turned off at compile time. Alternatively, for petaflop systems, the files can live locally at each compute node and no congestion is created.

4.3 Communication Interaction

The measurement commands of the agent system is orchestrated by a single frontend agent. The commands are broadcasted downwards through the tree until the leaves (the PerSyst Agents) are reached. The pairwise parent-to-child relationship is always kept for the transmission of the measurement command as well as for configuring commands sent for the preparation of an agent. A fundamental difference from other tools that gather data in the same way is the bottom-up communication in the agent tree of the properties. The relationship of child to parent is changed in order to gather the properties of one job centrally to one collector—when possible. Unless the data load surpasses the collection capacity of a Collector, the performance data of a job will be collected at only one Collector.

The agent communication consists of several commands that are processed by triggering a determined action or simply receiving information. The commands from any parent agent to any child agent are:

- `CONFIGURATION`: a command to configure an agent. This command is accompanied with its corresponding transfer object. The object contains the configuration to initialize all the agents. All agents react to this command, except the frontend, whose configuration is read from a file. This command is sent only at start up and it may include configuration details like running on test mode or production mode; it may contain a severity configuration file to be read, a topology file, cycle times, and debugging features that may be turned on.

- `TOPOLOGY_TRANS`: a command that is accompanied with the sending of topological information. This topological information contains the relations of the agent tree with network information. This command is only used at the initialization of the agent network or to recover from failure when the tree is rebuilt.

- `TERMINATE`: a command to terminate the agents. It is triggered by the frontend and sent to all the processes using the network such that after broadcasting to the children agents the process will terminate. The PerSyst Agent terminates by freeing its memory without doing any other action prior to termination. This command is used to remove the entire PerSyst Tool agents from the HPC system. A signal can be captured in the frontend agent which will propagate termination to the tree network including itself and thus, it allows for a scalable termination of the agent tree.

- `HEARTBEAT`: a command used by the heartbeat system in all agents, except the frontend, to check if the process should continue running or not. If a heartbeat is transmitted the receiving agent will renew its time-out to terminate so it will continue running. In case the heartbeat is not received, the agent activates a shorter time-out used as a marginal time. Within this marginal time the agent still waits for a belated sent message. At failure of receiving such a message the agent will terminate. The heartbeat system enables the termination of the entire agent tree when the frontend terminates due to failure.

- `JOB_DISTR`: a command associated with a transfer data structure that contains the job information and recipient Collector's network information (port and host). The performance data of a job is assigned a *route* that is communicated downwards to the leaves of the tree. This object contains the route information: the network information of the first receiving agent as well as the agent which centrally collects the entire job performance data. Only the job information pertaining to the PerSyst Agents in the sub-tree of the agent which will receive the job information are sent such that the message is broken into smaller messages as it is transmitted down the network hierarchy. If the message is too large it is split into manageable parts and sent in several messages.

- `START_MEASURING`: a command that starts a measurement. It is propagated by the parent agents to their child agents and is ultimately received by the PerSyst Agents. There is an associated data transfer with this command which is the measuring timestamp and the strategy identifier. The strategy identifier allows the tool to easily change strategies from one measuring interval to another one. This feature has not been used, but was left to allow extensions of the tool. The measurement is started right after this command is received.

The command from the PerSyst agents to the Collector agents is `GROUP_PROPERTY_TRANS`: a command associated for the PerSyst agent to forward the properties to a Collector. This command is accompanied with a transfer data structure that contains the property data. The properties are grouped with a message size that is controlled by the `HPCSystem` abstract class's

implementation (see Chapter 6 for more details on the HPCSystem abstract class). Once the group reaches the desired number of properties to be sent, the group is sent by the framework to an assigned Collector. The properties are bundled together (instead of single messages each with one property) in order to optimize the transfer of properties.

The commands from any child agent to parent child agent are:

- REGISTER_AGENT_PORT: is a command associated with a transfer data structure that contains the agent port. At start up, every child agent will dynamically select a port to listen to and send a command with the corresponding transfer object to register its agent port to the parent.
- REGISTER_AGENT_HOSTPORT and REGISTER_MASTERHOSTPORT: are commands to register network information. Alternatively to the previous command, the REGISTER_AGENT_HOST PORT and the REGISTER_MASTERHOSTPORT commands can be used when the ports are fixed. This can be specified in the topology file. This variant is also used when an xinetd super server is used to initialize the PerSyst Agent and/or a Collector Agent.

The communications commands between SyncAgents and Collector agents are:

- REQ_QUANTILES_TRANS: a command used for requesting the quantiles to the child agents. This command will be transmitted until it reaches the Collectors. If the Collectors have aggregated an entire job they will not pass any quantile information to their parent, instead the command will trigger the implementation for the output of aggregated information.
- QUANTILE_TRANS: the response command triggered by the REQ_QUANTILES_TRANS command. The response to the parent agent will be the corresponding transfer object that has the aggregated quantile information.
- COLL_INF: the command associated with a transfer data structure that contains the communication addressing of the collector agents.

All of the agents are servers which are listening to commands and have the capability to send and receive packets to each other using the predefined binary format of Common Data Representation (CDR). This binary format is used to pack the data stream into small messages that will be sent over the network tree. The binary serialization saves the data without any markup or meta information. A custom format provides flexibility by providing insertion and extraction operators for basic types. This representation is normally smaller than the equivalent in XML representation resulting in a faster de-marshalling and serialization.

4.4 Cycles and Time Control

The measuring cycle is divided into two phases: a measuring and analysis phase and an idle phase where the PerSyst Agents require almost no CPU utilization.

Table 4.1: Times for cycle control

Total cycle time (TC)	600 seconds
Measuring and analysis phase	120 seconds
Idle Phase	480 seconds
Tool basic measurement time	10 seconds

In the measuring and analysis phase there is also enough time for the PerSyst Agents to send the data. This phase is triggered by the frontend agent given that the frontend synchronizes and controls all the other agents. Several third party tools require some initialization time and, from experience, it was found that some measuring tools may crash. A slack time, to allow retrial of measurements, is included in this phase. The basic unit of measurement for tools which uses time is, therefore, considerably smaller than the measuring phase to allow for at least repeating the measurement and sending the data over the network. However, the basic unit of measurement is large enough to calculate the value per second (typically in the range of 1 to 20 seconds) of the measured metric before it is used for calculations by the properties.

The idle phase is used by the middle layers and the frontend for aggregation and collection tasks. Typically, agents in the middle layers are not deployed in the compute nodes but on other nodes such that the compute nodes are not loaded with monitoring computations. After the post processing of the properties the agents use a fraction of the idle phase to store the data. The storage means is implemented ad hoc for the system (see Chapter 6). Usually, buffering into files is sufficient and scales well since the information per job is already aggregated. Also, writing into the files is an action which is done at different times. Another task included in the idle phase is the failure recovery. An example of cycle times is shown on Table 4.1.

The PerSyst Tool can be configured to a precision of seconds for the measuring cycle, however, this is not recommended. Measuring every 10 minutes proved to be sufficient to capture the performance over time of the running codes; doing it more often strengthens the quality of the data but has as a downside that the PerSyst Agent needs to use the resources more frequently. Any performance collection tool will have some impact on the overall performance; doing the collection on a spaced interval of minutes makes this impact negligible.
The cycle administration is implemented as time-outs with the schedule timer facility in the ACE library. Each time-out triggers the next phase to be communicated. From experience, it is better to always start the cycle when the modulo operation evaluates to zero with the Unix time stamp (time elapsed since 1^{rst} of January, 1970). The reason behind this is that some delays in raising the time-out alarm may happen and the next cycle can be adjusted. Given that the delay to the next cycle time is known (it is determined by the modulo and the Unix timestamp), the timeout can be adjusted. More over, the measurements are done at an expected timestamp and are not dependent on the start up of the PerSyst Tool; so the time that measurements are carried out can easily be calculated and other measurements outside

PerSyst can be coordinated. If UT is the Unix Timestamp, the time to start (TS) at the very beginning is defined as:

$$TS \equiv TC - UT \pmod{TC} \tag{4.1}$$

Where TC is the total cycle time, see Table 4.1. Upcoming events will be corrected to this cycle, if necessary, such that if a delay arises due to processing time the next scheduler is set with less time.

4.5 Failure Recovery

Failure recovery is very important in continuously running daemons. In a tree structure design of several layers of agents, a failure in a middle layer component implies a failure in the entire sub-tree. In comparison to a non scalable structure of one centralized daemon which controls the rest, this is a clear disadvantage. In such a structure, if a daemon dies, only the daemon's monitored resources would be lost. However, the necessity of scaling bounds the monitoring tool to keep a tree structure with more than two layers of agents and thus, to have a recovery mechanism which takes place immediately after a fault. The fault recovery has been designed such that there is no interruption in the other PerSyst Tool components during the repair process. At most, the information of a monitoring interval from the affected agents is lost.
There are several sanity checks built into the framework to ensure that all components are running and are not faulty. Except for the frontend, the rest of the components are checked with a system of heartbeats. A heartbeat flag is set when the system performs any type of communication. This flag is checked at cyclical intervals. When no heartbeat was received (the flag is false) and there is an expired time-out, the agent will set another shorter time-out. After expiration of the second time-out the agent will terminate. The first time-out that was set is synchronized with the cycle times; the second time-out is only waiting for a belated message of the completed cycle. In case the message is received within the second time-out, the next heartbeat is set such that it is synchronized again with the cycle times. Agents which have released other children agents control whether the children are still up and running with a similar heartbeat registry system. The agents which are not responding are replaced by their parent agent. A component failure will trigger the recovery mechanism on the next cycle.

There are several methods for the book keeping and reestablishment of missing communication links or agents. The available approaches [8, 59], however, do not adapt to the communication characteristics of how the performance data is collected with the PerSyst Tool. The network information of the agents where this performance data is transmitted is contained in the *route* (see Section 4.3). Thus, redirecting of the performance data traveling through this route may result in having the performance data of a job being split, and having its output at more than one agent. The job would then appeared duplicated or the record of the monitoring data would be replaced by the latest agent (depending on the specific implementation). A simple approach has been taken, which recovers the missing link or agent after one timestamp: the topology of the middle layers of the agent tree allows to be changed dynamically at the runtime

of the PerSyst Tool. The parent agent of and agent which runs on a faulty node, can change the host where the new sub-tree will be running. The PerSyst Agents belonging to the affected sub-tree are then placed at each compute node, where they were originally assigned (except the faulty nodes). Thanks to the heartbeat system the agents which don't have a communication link with the parent agent will eventually terminate. Only agents in the middle layers can be reallocated: there is no sense on reallocating a PerSyst agent of a faulty node; the performance data of the node will be lost.

The PerSyst Tool performs a check to detect error and faults in order to know when it is necessary to terminate the agent. This includes the checking of a timestamp with the current time. The PerSyst Agent makes a consistency check when it receives a timestamp, if the received timestamp is outside a range of permitted slack different from the current time, the PerSyst Agent will stop its reactor, make a logged output so the discrepancy is known, and the daemonized task will terminate. This time range is defined as 90% of the measuring time to give enough time for correction when the error log message is detected but still before the completion of a measurement. A global cluster time setting is therefore required, however, it is not necessary to have exact clock times. The parent Collector will eventually generate the missing agent. It was found that a node having severe but unknown problems was presenting anomalies, in such cases trying to perform measurements usually result, from experience, in unusable data.
A faulty process may come from an external source, for instance using another tool for extracting information. This can happen when using a blocking invocation to an external program (e.g., via the popen function in C/C++), or an unforeseen failure within a library call. For non-critical exceptions there are in-built time-outs to stop the faulty process. If there is enough time, repeat trials are conducted. If the subsequent trial fails, the measurement for this cycle may be lost but the next monitoring cycle will continue trying to perform the measurement. The Collector agent will not process the partially received information from the faulty process.

Error logging is an integral part of any continuously running software. The monitoring tool has a logging infrastructure which allows each agent to write errors and logging messages in its own file. Logging must also be scalable. The solution to make logging scalable is to use a *representation system* where only assigned agents are allowed to write log files. The amount of logging has to be decided outside of the framework, i.e. the specific implementation for an HPC system. Another alternative was considered which collects logging messages without doing an output; the output is only done if an error occurs [5]. This solution was deemed to be inappropriate due to three main reasons. Firstly, the crash can occur before any output is printed. Secondly, given that the agents are constantly exchanging information the information output in one agent may contribute to determine an error in another one. With a representation system there is the possibility to keep these logging relations among the chosen agents. Finally, if there is a generalized system problem that will reflect in error messages within the agents, the agents will produce an amount of logging that will clog the system. A representation system

has as a downside that not all anomalies will be captured. To alleviate this downside, a handful of selected priority errors have been defined; these will be printed into the log files in any case. The file destined for logging will not only contain the id, or *tag* of the logging agent but also the timestamp when the agent was initialized. This proved to be useful to indicate that an agent had crashed and restarted by only taking a look at the amount of log files. Of course, a cleaning policy is needed to remove old log files from the logging directory.

5

Statistical Aggregation of Performance Data

In this chapter a detailed description is given on how aggregation and collection of performance data is performed in the context of a tree hierarchy (refer to Section 4.1 for more details on the agent tree.). Section 5.1 surveys the use of descriptive statistics to be used for performance data. Section 5.2 describes the method used to estimate percentiles. Section 5.3 describes the algorithms to optimize the collection of data and to avoid quantile estimation as opposed to quantile calculation.

5.1 Aggregation Using Quantiles

Monitoring of time series data implies the growth of data over time. Data is required for at least the entire life-span of a supercomputer such that the development of one application can be compared. This enables the observation of the evolution of a code with respect to performance when run with different parameters, number of cores, and data input sets. A supercomputer with 100,000 cores with, for example, 40 collectible metrics will generate 16MiB in one time point. If metrics are collected once per second, the total data collected is approximately 1.4TB per day. While tape storage provides a cheaper solution for storing such an amount of data per day, a more expensive alternative, like disc storage, is required for a faster querying of jobs. The volumes of data should be kept as small as possible to avoid such a steep growth of the data. This can be achieved in two ways: firstly by reducing the frequency in the collection and secondly by reducing the incremental amounts of collected data. This section deals with the latter method.

The tool's current approach stores only relevant aggregated information which has the advantage of reducing the volume of data not only for a permanent storage but also at collection time. The aggregation of the data is a compromise between reduction in its quality and a more cost-effective data storage with less disturbance at the monitoring time. The quality of the data has to be good enough to detect applications with bottlenecks. The analysis and selection of properties, done at the time of the measurement, will narrow down the studied population and this, in turn, counteracts the reduction in the quality of data.

Descriptive statistics offer a wide variety of methods to aggregate and present data. An intuitive solution is to register the frequency of values into bins or classes; this allows the depiction of the distribution with a histogram. Several drawbacks have been identified when using histogram bins or classes. There are two possibilities, to define either fixed bins a priori, or to use adaptive equidistant classes. The drawbacks for each possibility are:

1. Defining fixed bins before carrying out the measurement has two main disadvantages: Firstly, the ranges are different for different variables, so each range for each variable type has to be stored additionally with its corresponding frequency. For instance, if the architecture is capable of doing four flops in one cycle, storing properties for the number of cycles will have a different range than storing floating point operations. A normalization of all values would complicate the comparison of two properties if the person using this information does not have the architectural specifications in mind. Secondly, if the observations reside in fewer classes than the ones available the quality of the descriptive statistics is diminished. So using the entire possible range from a given population puts at risk loosing the level of detail.

2. If adaptive equidistant classes that adapt to the observed minimum and maximum are used, the aforementioned risk of reduced quality may be overcome if the population resides close together. However, for every measurement, the newly computed classes must be calculated and at least the minimum, and maximum must be stored along with all the recorded frequencies for every measurment interval. Every time a query takes place the bins need to be recalculated. Alternatively, every range the bin covers can be store along with its frequency doubling the storage space of the database.

The need of synthesizing data efficiently has been resolved largely by storing a fixed amount of quantiles. For practical purposes the definition and implications of using percentiles will be used in this chapter, other subsets (for example quintiles, quartiles, or deciles) can be adapted to the definitions and usage. The standard definition of the kth percentile P_k [34, 71] is the value within the range of x, called x_k, which divides the data set into two groups. The fraction of the observation specified by the percentile falls below and its complement falls above x_k. Thus, it is necessary to obtain the cumulative frequency of the variate x, hereafter *cdf* (also known as cumulative distribution function) to calculate any given percentile. To calculate the kth percentile of a distribution, P_k, the value of x_k which corresponds to the cdf of $\frac{Nk}{100}$ is taken, where N is the sample size. When $\frac{Nk}{100}$ results to be a non integral, the linear interpolation of the cdf between the value corresponding to the cdf, $\lfloor\frac{Nk}{100}\rfloor$, and the next value corresponding to the cdf, $(\lfloor\frac{Nk}{100}\rfloor + 1)$, is calculated.
As an example of this definition, take the 20th percentile. This is a value x such that 20% of the samples are smaller or equal to x.

Using equidistant percentiles makes it possible to describe the distribution of a population

via feature reduction of the variate x. Seen from another perspective this is equivalent to calculating non-equidistant bins which have as a common criteria a fixed number of elements between each percentile. The advantage is that the stored data becomes agnostic if it relates only to a fixed number of percentiles that will always be stored rather than having to store the range values for each data set. Percentiles adapt to their observed minimum, i.e. percentile zero, and to their maximum, i.e. percentile one hundred, without risking the loss of the desired level of detail.
At the moment of writing this document, resource managers and applications with the capability of varying the number of cores during runtime were not available for production, they are only ongoing research [109]. In any case, by collecting the percentiles the data structures should be capable of handling this situation.

The one drawback found with the usage of percentiles is that performing meta-aggregation of percentiles is not always possible using the standard definition.

Definition Let $P_a = \{p_i^a\}$ with $0 \leq i \leq 100, i \in \mathbb{Z}$ be the percentiles of a population A with size N_a and $P_b = \left\{p_j^b\right\}$ with $0 \leq j \leq 100, j \in \mathbb{Z}$ be the percentiles of a population B with size N_b, then the *meta-aggregation of the percentiles* of P_a and P_b is the calculation of percentiles of the population C where $C = A \cup B$. This definition can be applied to more than two populations.

The leaves of the tree topology calculate percentiles on their disjoint populations at the tree-leaves. As soon as these percentiles move to the next layer of the network with fewer agents, percentiles from percentiles are calculated. In the case of meta-aggregation of averages, the calculations are exact when population sizes are known, but not in the case of percentiles.

This can be verified by finding at least one generic constellation where percentiles can't be calculated. Let A and B be populations whose only known information are their percentiles and the population sizes, with $P_A = \{p_i^a\}$ and $0 \leq i \leq 100, i \in \mathbb{Z}$ being the percentiles for population A, and $P_b = \left\{p_j^b\right\}$ with $0 \leq j \leq 100, j \in \mathbb{Z}$ the percentiles for population B, and with population sizes N_a and N_b respectively. Additionally, the following constellation $p_0^a < p_0^b < p_1^a < p_1^b < p_2^a$ holds. The information available to try to calculate P_c, the multiset of percentiles of C with $C = A \cup B$, is P_a, P_b, and their population sizes N_a and N_b.

Definition $E_X(p_i, p_j)$ is defined to be the number of elements between the percentiles p_i and p_j, where $0 \leq i \leq j \leq 100$ and $i, j \in \mathbb{Z}$ for the *cdf* of X.

In order to try to find where p_1^c lies both $E_C(p_0^a, p_1^a)$ and $E_C(p_0^a, p_1^b)$ are calculated.
$E_C(p_0^a, p_1^a) = N_a/100 + N_b/100 - x_1$ where x_1 is an unknown number of elements.
$E_C(p_0^a, p_1^b) = N_a/100 + N_b/100 + (N_a/100 - x_2) = 2N_a/100 + N_b/100 - x_2$ where x_2 is an unknown number of elements.
For $p_1^a < p_2^b$ to be true, then it must hold that $0 < x_1 \leq N_b/100$ and for $p_1^b < p_2^a$ to be true, then it must hold that $0 < x_2 \leq N_a/100$. From this it follows that $E_C(p_0^a, p_1^a) < (N_a + N_b)/100 \leq$

$E_C(p_0^a, p_1^b)$.
By definition, the percentile at $k = 1$, p_1^c is the element at $(N_a + N_b)/100$ of the cfd. Thus, the value is p_1^c has to be in $p_1^a < p_1^c \leq p_1^b$ (given that $E_C(p_0^a, p_1^a) < E_C(p_0^a, p_1^c) \leq E_C(p_0^a, p_1^b)$).
The only possibility that p_1^c can be precisely calculated is that $p_1^c \in P_a \cup P_b$. For this to be true, it should hold that $x_2 = N_a/100$ and $p_1^c = p_1^b$, however, there is no information available about x_2 except that $0 < x_2 \leq N_a/100$, therefore p_1^c can not be determined from the given information.

This also applies to a subset of the percentiles (like deciles, quintiles, etc). Fortunately, percentiles—as well as any percentiles subset—can be estimated in these cases.

A feature of monitoring systems with tree topologies is that they can be configured to perform meta-aggregations. If percentiles are used, estimations are required when an application requires to use the entire tree, i.e. to apply meta-aggregation of percentiles. There are two dimensions which can be aggregated given that an application runs over time with a fixed number of cores. In order to see the performance variations in time, only the number of cores per timestamp are aggregated.

5.2 Percentile Estimation

In case percentile estimation is required the percentiles per Collector per Job are collected within a SyncAgent. These percentiles are collected and estimated at each common parent of the Collectors. The population of the properties from a Collector and a job are estimated [49]. For example, take

$$P_1 = \{p_0^1, p_1^1, ..., p_{100}^1\} \tag{5.1}$$

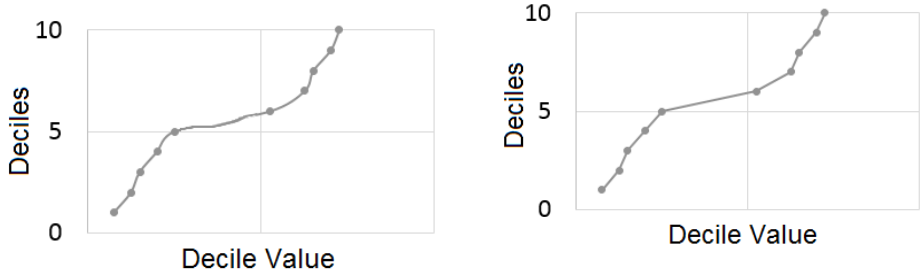
as the percentiles from Collector 1 (C_1) and

$$P_2 = \{p_0^2, p_1^2, ..., p_{100}^2\} \tag{5.2}$$

as the percentiles from Collector 2 (C_2). Both P_1 and P_2 belong to the same job such that the new percentiles need to be estimated from both of them. Given that a distribution is not known a priori, the entire set of observations from P_1 and from P_2 is estimated assuming a uniform random distribution between the percentiles. As seen in Figure 5.1, a uniform distribution (not randomized) assumes that the data between two deciles is uniformly increasing and curves in the cdf are replaced with a line joining two deciles. While randomization is necessary for a small number of elements, for a large number of elements the randomization will tend to reproduce uniformity. Thus, no matter the amount of the elements that have to be recreated, they are produced randomly.

The percentile values themselves do not need to be changed, they are part of the newly recreated set, for example:

$$S_1 = \{p_0^1, r_1^1, r_2^1, ..., p_1^1, r_n^1 ..., p_{100}^1\} \tag{5.3}$$



(a) Real distribution with deciles shown.

(b) Estimation using uniform distribution between deciles.

Figure 5.1: Approximation of a population with uniform distribution

and

$$S_2 = \{p_0^2, r_1^2, r_2^2, ..., p_1^2, r_n^2 ..., p_{100}^2\} \tag{5.4}$$

where r are the random values and S_1 and S_2 are the recreated sets. The new estimated set is then $S = S_1 \cup S_2$. The random values are produced in such a way that they lie within the range of two neighboring percentiles thus the value r_i, lies between $p_k \leq r_i \leq p_{k+1}$. The number of random values $R(k, k+1)$ between two neighbouring percentiles, k and $k+1$, where $k \geq 1$ is

$$R(k, k+1) = \frac{N_o}{n_p} - 1 \tag{5.5}$$

where N_o is the total number of observations and n_p is the number of percentiles (example: $n_p = 100$ when all percentiles are used, and $n_p = 10$ when only deciles are used). This formula applies except for the first interval, given that the minimum (considered to be the percentile zero) is in this range, there is one less random value to produce:

$$R(0, 1) = N_o/n_p - 2 \tag{5.6}$$

Both sets S_1 and S_2 are grouped together and they form the estimated observations of the collectors $C_1 \cup C_2$. The cdf is calculated from S, the estimated population. The percentiles are then determined from the estimated population. Analogously, this method can be applied to more than two sets (i.e. percentiles coming from more than two Collectors). Once all the estimated sets are joined together an estimated but complete population is obtained whose cdf can be determined as well as its global percentiles.

Example Problem: Estimate a new set of deciles from Collector a and b given the following data:
$D_a = \{2, 40, 45, 60, 65, 67, 78, 90, 100, 113, 130\}$ with a number of observations $N_a = 40$, and
$D_b = \{8, 29, 40, 65, 80, 87, 92, 97, 106, 112, 134\}$ with a number of observations $N_b = 20$.

Solution: To estimate the deciles from both Collector a and b, the first step is to recreate their populations S_a and S_b.
To recreate S_a the required amount of random numbers must be produced between each decile. The amount of random numbers between d_0 and d_1 is calculated with Formula 5.6.

$$R_a(0,1) = N_a/10 - 2 = 2$$

The required amount of random numbers between any other pair of deciles d_k and d_{k+1}, where $k \geq 1$, is calculated with Formula 5.5:

$$R_b(k,k+1) = N_b/10 - 1 = 3$$

Thus, the recreated set will have the elements:
$S_a = \{$ 2, r_1^a, r_2^a, 40, r_3^a, r_4^a, r_5^a, 45, r_6^a, r_7^a, r_8^a, 60, ..., 113, $r_{27}^a, r_{28}^a, r_{29}^a$, 130 $\}$
with $d_0 \leq r_1^a$,$r_2^a \leq d_1$; $d_1 \leq r_3^a, r_4^a, r_5^a \leq d_2$; and so on.
Similarly, S_b is recreated by calculating the required amount of random numbers and generating the cdf. The amount of random numbers between d_0 and d_1 is calculated with Formula 5.6:

$$R_b(0,1) = N_b/10 - 2 = 0$$

ie. no random numbers between d_0 and d_1 are needed.
Between any other pair of deciles $k, k+1$, where $k \geq 1$, the number of generated random numbers is:

$$R_b(k,k+1) = N_b/10 - 1 = 1$$

Thus the recreated set will have the elements:
$S_b = \{$ 8, 29, r_1^b, 40, r_2^b, 65, ..., 106, r_9^b, 130 $\}$
with $d_1 \leq r_1^b \leq d_2$; $d_2 \leq r_2^b \leq d_3$; and so on.
All random numbers are created within their ranges with a uniform random distribution.
Having calculated S_a and S_b the complete estimated population S can be calculated $(S_a \cup S_b)$:
$S = \{$ 2, r_1^a, r_2^a, 40, r_3^a, r_4^a, r_5^a, 45, $r_6^a, r7^a, r_8^a$, 60, ..., 113, $r_{27}^a, r_{28}^a, r_{29}^a$, 130, 8, 29, r_1^b, 40, r_2^b, 65, ..., 106, r_9^b, 130 $\}$ from which the deciles can be estimated.

5.3 Data Collection

The data collection starts at the source, i.e. the PerSyst Agent. The agent obtains the raw measured information of the complete compute node and stores it internally for processing and analysis in properties. Then, it will either send the properties upward in the agent tree or it will write the final results into the assigned database or file system. The response of the PerSyst agents is not necessarily directed to their Collector parent [44]. The responses can be remapped so that the properties of a job can be collected centrally, avoiding percentile estimation (the top-down control of agents was kept, by sending the command through the tree structure of the agents, just like other hierarchical tools). This mapping of the properties of the jobs to collectors is determined at the SyncAgent which acts as the frontend. A *balancing of jobs* is

made in which the load of the PerSyst agents is assigned to Collectors for data collection, as described in Algorithm 1. The three basic ideas of this algorithm are:

1. Allocate the properties of the biggest jobs first (by sorting the jobs by their load).
2. Find the Collectors with minimum assigned load (by sorting the Collectors by load).
3. From all these Collectors assign the collection to the Collectors with the closest tree distance to the PerSyst Agents.

Algorithm 1 Algorithm to distribute job's properties to collectors.

Require:

$$\left\lceil \frac{\sum_J l_j}{l_{max}} \right\rceil \leq |C| \tag{5.7}$$

Where C is the set of collectors and l_j is the load from job j and J is the set of all jobs at a measurement interval. l_{max} is the maximum load a Collector can take.

```
COMMENT: A job j is running on different nodes monitored by the set of agents A_j.
COMMENT: Initialize the load in all collectors in C:
for all l_c do
    l_c ← 0
end for
SORT jobs J in descending order of load l_j
COMMENT: Assign loads from jobs to collectors:
for all j ∈ J do
    if |A_j| = 1 then
        MARK j to be processed directly at a ∈ A_j
    else
        DistributeLoadOnCollectors(C, l_j, A_j)
    end if
end for
```

When the job size fits exactly in one compute node, the job is processed locally and stored from this compute node. If the system runs entirely on one-node sized jobs, the multiple accesses to the storage device can create a bottleneck. Thus, if these requests are exceeding the limits imposed by the storage medium, these one-node jobs are sent through the network tree (for clarity, this sanity check and the corresponging action are not shown in Algorithm 1).

l_j and l_{max} are called loads and the terms represent the amount of properties of a job l_j or the maximum amount of properties a Collector can take l_{max}. Given that the balancing of jobs algorithm is done before the measurement of the jobs, the total amount of properties are not known, so it is assumed that the maximum amount of properties will be transmitted. The algorithm can be implemented such that the loads represent the properties of a node or another domain which may be more convenient (core, hyperthread).

Algorithm 2 shows how the assignment of properties to the Collectors is performed. The idea is to use the tree structure only when it is needed, otherwise to aggregate with exact calculations and store information as quickly and as closest to the source as possible. It has four main steps. Firstly, the number of collectors needed(n_c) for one job is determined. Secondly, the most suitable Collectors are found with the `FindBestCollector` Algorithm. Thirdly, the loads of child agents of these Collectors which also belong to the job (if any) are assigned. Finally, the rest of the load of the job is assigned.

Note that the placement of the rest of the load (Lines 22 to 25 in Algorithm 2) allow for a load greater than l_{max} for a Collector. In practice, one-node jobs are processed at the PerSyst Agent leaving Collectors with more room to take load up to l_{max}. In theory, the algorithm requires a check in Lines 22 to 25 that compares the load in the Collectors against l_{max}. If this load is surpassed, another Collector from the set of collectors C^* can be acquired with the `FindBestCollector` algorithm. However, the downside to this variant is that the properties of these jobs will be distributed over more Collectors and the percentile estimation is needed. Alternatively, the mix of job sizes (load) should be taken into account to choose a higher number of Collectors suitable for distributing the load.

Definition The tree distance of two tree nodes (leaves or nodes) has been defined to be the longest distance in terms of edges joining the nodes where the data is transmitted, such that this data is collected centrally at the root of the smallest sub-tree which contains the two tree nodes.

Figure 5.2 shows examples of how the tree distance is calculated. The tree distance between node 13 and node 15 is two, given that the data has to be transmitted through two edges at most to reach the destination node. The tree distance between node 12 and node 15 is one, and the tree distance between node 15 and node 30 is three.

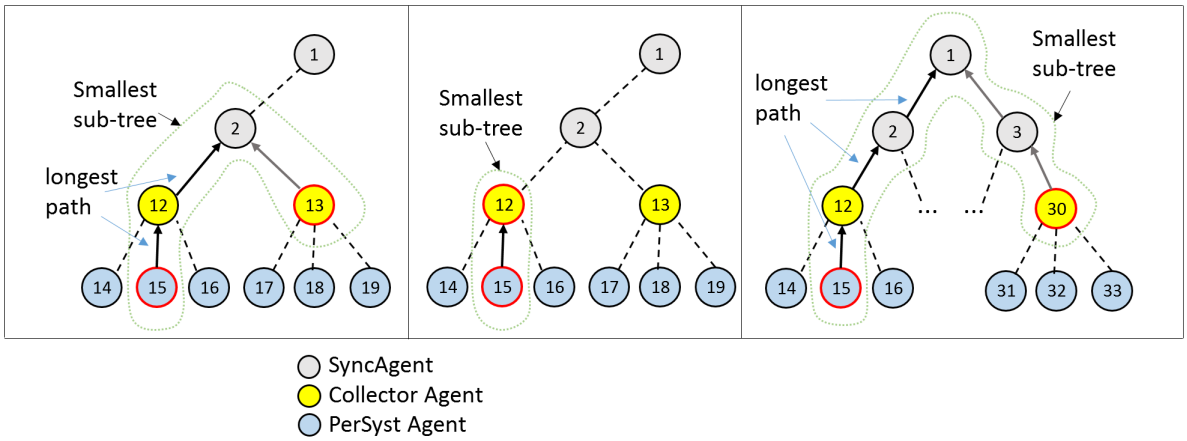


Figure 5.2: Examples of the tree distances between nodes

Algorithm 3, `FindBestCollector`, finds the Collector with the minimum assigned load (l_c in Algorithm 1). When minima are found the algorithm considers also the tree distance of a

Algorithm 2 Algorithm to distribute performance data load to several collectors.

1: Algorithm DistributeLoadOnCollectors(C,l_j, A_j)
2:
$$n_c \leftarrow \left\lceil \frac{l_j}{l_{max}} \right\rceil \tag{5.8}$$
3: COMMENT: n_c is the number of Collectors that will receive the load from all agents A_j.
4: iter $\leftarrow 0$
5: $A'_j \leftarrow A_j$ COMMENT: A'_j is a temporary variable for agents of a job.
6: $C^* \leftarrow C$ COMMENT: C^* is a temporary variable for all the collectors.
7: **while** iter $< n_c$ **do**
8: $c \leftarrow$ FindBestCollector(C^*, A'_j)
9: INSERT c in C'
10: REMOVE c from C^*
11: COMMENT: Removing c from C^* avoids having the same Collector in the next iteration.
12: COMMENT: Allocate load of child agents of c (A^c_j) in l_c until l_{max} is reached:
13: **for all** $a \in A^c_j$ **do**
14: **if** $l_c + l_a \leq l_{max}$ **then**
15: $l_c \leftarrow l_a + l_c$
16: REMOVE agent a from A'_j
17: **end if**
18: **end for**
19: $iter \leftarrow iter + 1$
20: **end while**
21: COMMENT: Place the rest of the load on the first n_c Collectors found:
22: **for all** $a \in A'_j$ **do**
23: $c \leftarrow$ FindBestCollector(C', A_j)
24: $l_c \leftarrow l_a$
25: **end for**

Collector and a PerSyst Agent so the properties will be sent to their closest Collector. Note that the algorithm always works with the original configuration of the tree agent.

The closest Collector to a job is defined as the Collector with the minimum total distance of itself to all the PerSyst Agent nodes where the job is running.

$$d_{C_{min}} = \min_k (\sum_{i=0}^{N} d_{ik}) \tag{5.9}$$

where d_{ik} is the tree distance between i (ith PerSyst Agent) and k (kth Collector Agent) for all the N PerSyst Agents that are measuring a job, and C_{min} is the Collector with the smallest tree distance ($d_{C_{min}}$). By performing such a distribution, the algorithm guarantees that the normal parent-child relations are used as much as possible. This avoids sending additional Collector information to the PerSyst Agents more than necessary.

Algorithm 3 Algorithm to find Collector with minimum load and minimum tree distance.

```
1: Algorithm FindBestCollector(C, A)
2: COMMENT: Returns Collector with minimum load and minimum tree distance given the
   set of Agents A.
3: C' ← all c with minimum load.   COMMENT: C' = {c ∈ C|l_c = l_min}
4: if |C'| > 1 then
5:   for all c ∈ C' do
6:     d_c ← 0   COMMENT d_c is the accumulated tree distance from c to all a ∈ A.
7:     for all a ∈ A do
8:       d_c ← d_c+ TreeDistance(c, a)
9:     end for
10:  end for
11:  SEARCH for c the Collector with min(d_c)
12:  COMMENT: c is the collector with minimum distance. If |min(d_c)| > 1, i.e. more than
     one Collector, only the first one is returned.
13:  RETURN c
14: else
15:  COMMENT: c ∈ C' with minimum load.
16:  RETURN c
17: end if
```

The algorithm that calculates the tree distance is not shown but has a time complexity of $\mathrm{O}\left(log(n)\right)$ as it reduces to a tree search. The complexity of Algorithm 1, including the other calls, is therefore $\mathrm{O}\left(n^2 log(n)\right)$ where n is the number of the measuring agents.

For jobs where $l_j \leq l_{max}$, the maximum load a Collector can take, it is only necessary to use one Collector and not the entire tree structure for extracting and collecting data. For convenience these jobs will be defined as *medium sized* jobs, i.e. jobs whose load $l_j \leq l_{max}$. When $l_j > l_{max}$ the job size is handled with quantile estimation and use the tree partially to fit the collection in the lowest possible number of collectors to extract the information, these jobs are called for convenience *big* jobs.

The last remaining task is to calculate the common collection node for the big jobs among the SyncAgents of an entire job. The original parent-child relations of SyncAgents to Collectors and among Collectors are used to propagate the property data. Only the PerSyst Agents may transmit property data to other Collectors different from their parent Collector. Thus, the SyncAgent responsible for big jobs can be determined with tree operations. Once the loads are assigned and the final destination is defined (PerSyst Agent, Collector, or SyncAgent). The *route* of the properties for each job is defined (See the `JOB_DISTR` command in Section 4.3) and propagated downwards to the tree, via the original tree topology. The measurement, analysis, and retrieval of properties via the assigned routes takes place. An example of the retrieval of

properties is shown in Figure 5.3. One-node jobs finish their collection at the PerSyst agent in charge of monitoring its node without requiring percentile estimations.
The second location is to collect the properties at the Collector level. A medium sized job running in more than one node will be centrally sent to one Collector, thus, the Collector can perform the precise calculation of percentiles without using estimations. This is the case for Jobs 1, 2, and 3 in Figure 5.3 which are processed in Collector Agent 1, 2, and 3 respectively.
The third location is at a SyncAgent. The properties from big jobs require more than one Collector and, thus, part of the tree topology to extract the data. Unfortunately, the estimation of percentiles can not be avoided when using a SyncAgent to aggregate. By using aggregation the communication is reduced as the data propagates to the tree top. A job which runs on the entire machine requires that the final collection of properties takes place at the frontend.

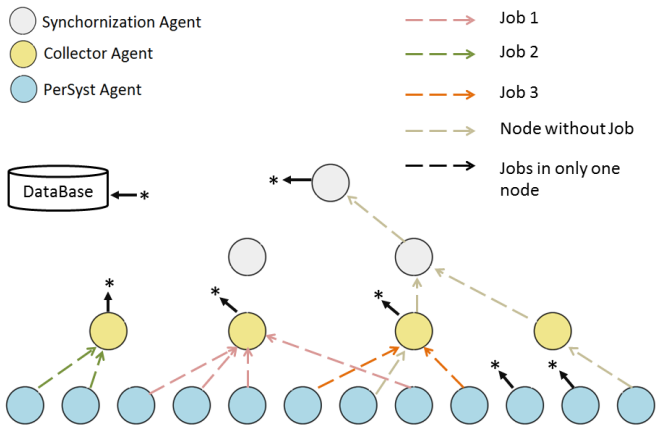


Figure 5.3: Example of retrieval of Properties

The storage of results in the file system or a database is distributed over time. Firstly, one-node jobs are stored; directly after the analysis step. Secondly, medium sized jobs are aggregated and stored by the Collector. Of course parallel storage takes place among the active Collectors. Finally, the layers of SyncAgents will store their job information (these are the big jobs).
The output of all the agents (to a parallel file system, or directly into the database, or other devices) is already aggregated. The bigger the job is, the higher the percentage of data reduction. Take as an example the usage of deciles: only ten quantifiers and the minimum, amounts to eleven data points per job that have to be stored. For a job with 4096 Cores, more than 99% data reduction is done. Thus, the final aggregated information which would be written on the parallel file system or inserted directly into a database is not a bottleneck itself.

6 Portability and Adaptability

In this chapter, the description of how the portability and adaptability of the PerSyst Tool is described. The software architecture of the tool and of each agent is discussed in the following section. The software engineering of the communication is described Section 6.2.

6.1 Framework and Abstract Classes

The software was designed as a framework to allow for portability and adaptability of the PerSyst Tool. The communication is handled within the framework and the communication commands activate via callback the specific implementations.
All of the system components, i.e. the SyncAgent, the Collector Agent, and the PerSyst Agent, have an interface which can be used with a tailored implementation.
The framework of the PerSyst Agent triggers the measurement command, enables the evaluation of the strategies (which in turn evaluates the properties), and passes the properties for communication to the Collectors. Several factors made it necessary to allow for specific implementations for the monitoring tasks. At the level of the PerSyst Agent, the strategies and the properties are architecture dependent. The set of measuring tools can be different among supercomputing systems and there are other tasks related to the measurement tool's functionality which depend on the HPC system's architecture or software. Decoupling is done with the use of four main abstract classes—the `Property`, the `Strategy`, the `MeasurementTool`, and the `HPCSystem`—to have a distinct encapsulation for the specific operating system, processor architecture, and available tools for measurement. Figure 6.1 shows the interaction among the interfaces. The `HPCSystem` interface allows the `Tool` to measure the raw data and pass it to the `Strategy`. The `Strategy` evaluates the `Property` and the `Property` is passed to the communication layer to be sent through the network.

There are two additional abstract classes: one at the level of the Collector Agent and the other at the level of the SyncAgent. They also enable the implementation of tasks related to the tool's functionality which depend on the HPC system's architecture or software. The interfaces of the tool are further described below. In the following sections the *JobID* refers to a unique job number which is provided by the batch systems.

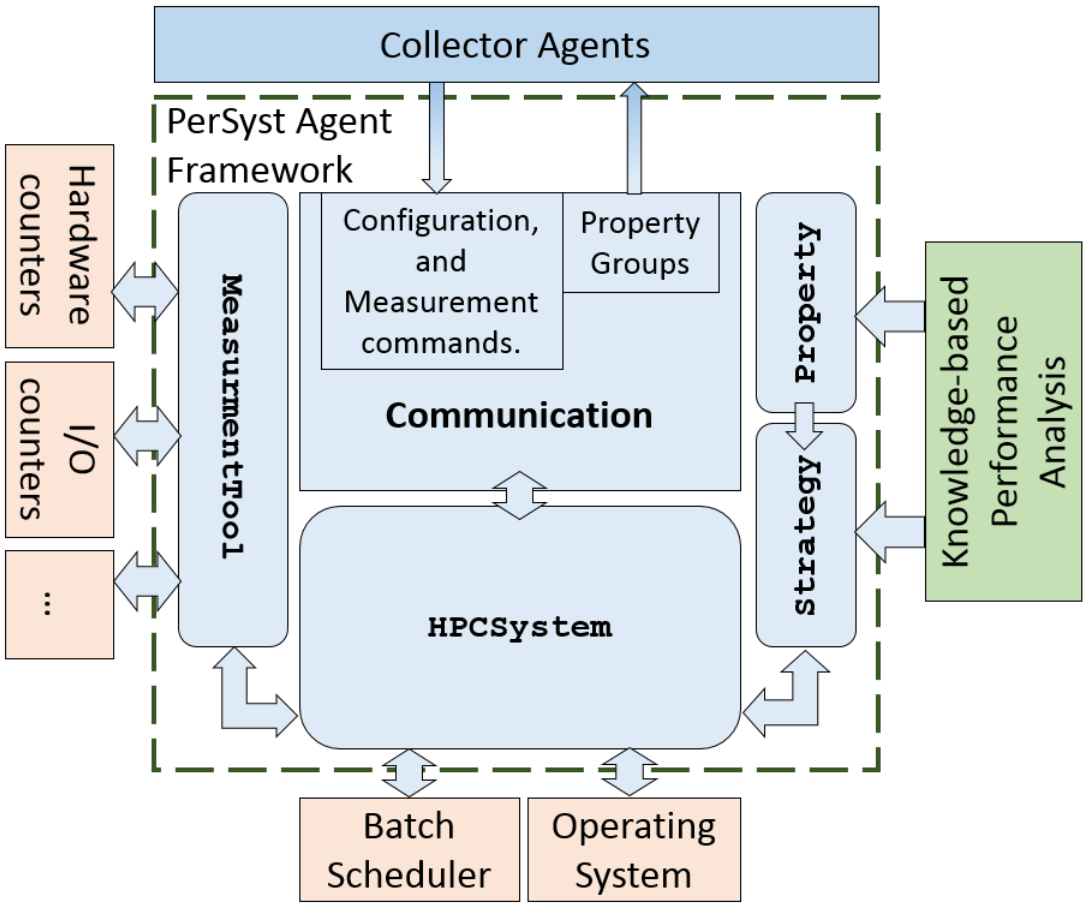


Figure 6.1: PerSyst Agent Framework

6.1.1 MeasurementTool

The `MeasurementTool` abstract class allows for the ad hoc implementation of tool which measures the raw hardware counters or other raw monitoring data. This abstract class, shown in Listing 6.1, is used at the PerSyst Agent. The `MeasurementTool` abstract class leverages third-party tools or APIs with systemwide[1] measurement capabilities. An invocation of an external tool via a system call can be done in the delegated class of the `MeasurementTool` interface. Other possibilities include parsing system files or library calls to an API that performs the measurements. The only requisite is that the implementation of the `measure` method, see line 5 of the `MeasurementTool` interface, inserts the event counters which correspond to each device id into the internal database of the PerSyst Agent. The device refers to the measured hardware domain. For instance: a given core, a hardware thread, or a compute node. Every device is given an id in order to handle the measurements in an internal database (`PerformanceDataBase` object is Line 5).

Listing 6.1: Abstract Class Tool

```
1  class MeasurementTool {
   public:
     MeasurementTool();
     virtual ~MeasurementTool();
5    virtual int measure(PerformanceDataBase &inOutPdb,
       int endTime, int basicUnitTime)=0;
     virtual bool forkTool()=0;
   };
```

In the specific implementation it is determined whether a fork for the tool is necessary (with process forking). Forking can be controlled by returning true in the `forkTool` method (line 7), then the framework will create (via a system call) a copy of the current process (as a child process); otherwise the measurement and the strategy will be evaluated sequentially. The forked child process is independent and can be internally parallelized with threads in case the measurement takes too long and a speed-up is required. Even if the PerSyst Agent is forked, the unix operating system will optimize the usage of the new child process which has been forked by avoiding the duplication of constant values from the process. Figure 6.2 shows the forking of three tools and one of the tools is parallelized using threads. The first three parallel processes represent a forked process for each measuring tool (tool 1, tool 2, and tool 3), while a tool itself may be further parallelized with the usage of threads (these are in Figure 6.2 thread 1 and thread 2; both threads from tool 1).

6.1.2 Property

The classes that derive from the `Property` abstract class hold the implementation of formulae and thresholds to calculate the property value and severity. This abstract class is used at the

[1] Systemwide measurement in this context refers to being able to measure a shared memory node within one process.

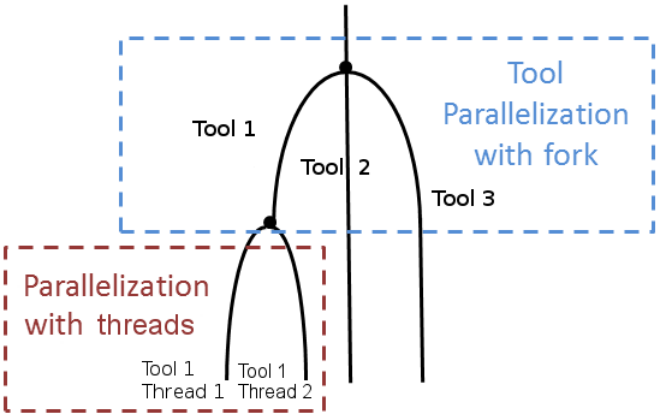


Figure 6.2: Tool parallelization within PerSyst Agent.

PerSyst Agent. Section 3 has detailed information on the `Property` abstract class. The metrics used in the `Property` will be passed to the `Tool` interface for measurement. The `Property` has a property evaluation method which is used to calculate the property value based on the hardware counters or other raw monitored data (where the analysis is done) with a threshold and a severity. The implementation of each Property defines the unique property `id`.
To avoid object generation, a class which inherits from `Property` is instantiated only once and evaluated against all device ids. The property data to be communicated is grouped into *property transfer groups* in order to optimize the communication. Instead of sending one message per Property, these are bundled together to minimize the number of messages.

6.1.3 Strategy

The classes derived from the `Strategy` abstract class hold the properties in tailored tree-like structures of properties, with each tree having a root property. This abstract class, used at the PerSyst Agent, is shown in Listing 6.2. More than one tree node is allowed in the `Strategy`; they can be specified as an array of root properties as shown in line 10. The strategy trees are used to refine from general properties to more specific ones when the general properties indicate a poor performance. The strategy tree itself can be specified by defining parent to child relations, as shown in the member variable in line 9. Each strategy may vary according to the underlying micro-architecture and the used properties. Virtual functions which are not pure have a default implementation that will be used if they are not overwritten. These include refining the trees, retrieving all metrics from the strategy tree, and calculating properties through the strategy trees (Lines 13 through 18 and Line 23).

Listing 6.2: Abstract Class Strategy

```
class Strategy {
public:
  Strategy();
4   virtual ~Strategy();
```

```
    virtual void getAllMetricsInGroup(int group,
      set<string> & metrics);

protected:
    map<Property*,vector<Property*> > parentChildMap;
    vector<Property*> rootProperties;
    int numberOfMeasurementGroups;

    virtual void getMetricsFromTree(Property * node,
      set<string> &metrics);
    virtual void getMetricsFromProperty(Property * prop,
      set<string> &metrics);
    virtual void calculateProperty(PerSystAgent * agent,
      PerformanceDataBase &hwcdata, int dev_id, Property * prop);

public:
    virtual string getStrategyName()=0;
    virtual void getAllMetrics(set<string> & metrics);
    virtual void calculateProperties(PerSystAgent * agent,
      PerformanceDataBase &hwcdata, int dev_id);

};
```

6.1.4 HPCSystem

The `HPCSystem` abstract class provides a means to delegate tasks unique to the underlying HPC system. This abstract class, shown in Listing 6.3, is used by the PerSyst Agent. It typically implements parsers to information needed for measurement activities. The software design decouples the main functions for monitoring from the underlying specific implementation required. However, it is unavoidable to have constraints. An assumption has been made, for example, that the location of information sources is available either at the compute nodes or at the frontend. Usually, this is the case as requests to a batch scheduler allows for the attainment of this information anywhere on a supercomputer. If the source of this information is located at the frontend, the information can be broadcasted until it reaches the PerSyst Agents at each measuring timestamp. The framework has restrictions in time and location regarding the sources of information; which still allows a variety of ad hoc implementations. These restrictions are a compromise between having a lightweight tool with enough options to be adapted to clusters with similar characteristics or having a tool which can fully adapt to any cluster but is no longer lightweight. The `Strategy` which defines the property tree, can access the metrics which will be measured. The `HPCSystem` abstract class can pass these metrics to the `MeasurementTool`. Implementations which can be adapted are:

- The constants which are specific to an HPC system. For example: the maximal size of the number of properties per transfer such that the amount sent is optimized for the system

architecture (Line 14).

- The reading of the underlying hardware topology. The term hardware topology refers to the layout of sockets, cores, and hardware threads, as well as *uncore devices*. Uncore devices refer to the shared domain among cores within the processor.
- Strategies and Measurement tools used (Line 11).
- The properties and strategies themselves which are not generic to an architecture. This is due to the specific hardware counters and monitoring units of each architecture.
- The format of the information that relates a job with where it runs. This includes the codification of the CPU identifier (Line 13).
- The source of the job information. The performance data is correlated to the job and this information can be requested in the compute node (directly through the batch scheduler or via the local virtual file system) or it can be transmitted via the network for every measurement (Line 16). The job information can be reset at each measurement (Line 17).
- Preparation activities for a next measurement (Line 18). For example: cleaning files.
- The communication details of the recipient agent (Line 21).
- The data structure of the information coming from the batch scheduler/resource manager.

Listing 6.3: Abstract Class HPCSystem

```
class HPCSystem {
protected:
  PerSystDirs * perSystDirs;
public:
  HPCSystem();
  virtual ~HPCSystem();
  void setPerSystDirs(PerSystDirs * perSystDirs);
  virtual void storeResults(int agentTag,
    map<JobID,JobIDResult*> & propResults,
    int measurementTimeStamp)=0;
  virtual map<Strategy *, MeasurementTool *> getStrategies(
    int forkStrategy)=0;
  virtual int getCPUKey(int tag, int cpu)=0;
  virtual int getNumberOfPropertiesPerTransfer()=0;
  virtual int getCodedTagFromHost()=0;
  virtual void insertJobInfo(JobRouteTrans &toinsert)=0;
  virtual void resetForNextMeasurement()=0;
  virtual void prepareMeasurement(map<int, string> &devid2jobID,
    map<int, pair<int, int> > &devIdRoute,
    bool &measurementOff)=0;
  virtual CommAgent& getAgentToSendTo(string &jobid)=0;
```

```
22    virtual void killOtherAgents()=0;
      virtual void changeDebugLevel()=0;
};
```

6.1.5 HPCSystemCollector

The `HPCSystemCollector` is an abstract class used by the Collector to delegate methods which are specific to the node where the collector is running. The Collector has a lighter framework architecture compared to the framework of the PerSyst agent as it already handles properties as C *structs* with only one struct type definition[2]. This struct for properties includes the property id, property value, and severity. Structs are preferred over classes as an optimization in order to avoid the creation of C++ object virtual tables. Specific implementations include:

- Implementing the agent placement differently, see Listing 6.4 at Line 5.
- Making an output to a database or file system, see Line 8.
- Checking whether a node is not in operation, see Line 7. This method requires interaction with the batch scheduler.
- Implementing any cyclic actions additional to performance monitoring, see Line 12.
- Sending a signal to another agent so the agent terminates execution, see Line 13.

Listing 6.4: Abstract Class HPCSystemCollector

```
class HPCSystemCollector {
public:
  HPCSystemCollector();
4   virtual ~HPCSystemCollector();
    virtual bool releaseAgent(Node * child_info,
      Node &own_info, int timetowait)=0;
    virtual bool nodeisDown(string node)=0;
8   virtual void storeResults(int agentTag,
      map<JobID, JobIDResult*> &propResults,
      int measurementTimeStamp)=0;
    void setPersystDirs(PerSystDirs * persystdirs);
12  virtual void timeToMeasureAction()=0;
    virtual void killOtherAgents()=0;
    virtual void changeDebugLevel(int tag)=0;

16 protected:
  PerSystDirs * perSystDir;
};
```

[2]As described in the Property abstract class, each property is codified using a class which inherits from the `Property` abstract class.

6.1.6 HPCSystemSync

The `HPCSystemSync` is an abstract class used by the SyncAgent to delegate methods which are specific to the node where the SyncAgent is running. The SyncAgent has a lighter framework architecture compared to the framework of the PerSyst Agent. It defines only the `HPCSystemSync` abstract class shown in Listing 6.5. Similar to the `HPCSystem` abstract class, this interface provides a means, at the level of the SyncAgent, to call tailored implementations and tasks unique to the underlying HPC System. Specific implementations include:

- Implementing output methods to store the data in a database or in a file system, see Line 7 of Listing 6.5.
- Implementing agent placement, see Line 5. If the child agent has been reallocated to a new host, the new possible host can be codified in the specific implementation of this method.
- Implementing any cyclic actions additional to performance monitoring, see Line 10.

Listing 6.5: Abstract Class HPCSystemSync

```
class HPCSystemSync {
2 public:
    HPCSystemSync();
    virtual ~HPCSystemSync();
    virtual bool startAgent(Node * child, AgentArgs & agentargs,
6     int timetowait)=0;
    virtual void storeResults(QuantileEstimator &properties,
      QuantileEstimator &severities, map<string, int> &jobsToFinalDest,
      set<int> &propids, Node * myNode,  int measurementTimeStamp)=0;
10  virtual void timetoMeasureAction(
      map<int, vector<JobRouteTrans *> > & transfers)=0;
    virtual void initialize()=0;
    void setPerSystDirs(PerSyst Dird * perSystDir);
14 protected:
    PerSyst Dird * perSystDir;
};
```

6.2 Communication

The socket based communication is implemented using the ACE (Adaptive Communication Environment) framework [96,99]. ACE is an object oriented framework designed to simplify the programming of interprocess communication, threading, and memory management [56,98,100]. It also serves as an API to general operating system services. The ACE software is designed to be extensible and avoids falling into common pitfalls when dealing with low-level communication libraries. All these qualities make it highly portable and ideal for new software development.

The PerSyst Tool framework uses, in all the agents, the Reactor [97] implementation of the ACE library. The Reactor allows concurrent requests to be handled by an agent. The service requests involve not only communication from other agents but also handling signals and time outs. The Reactor is responsible for taking these requests and dispatching the appropriate services.

7

Context of Evaluation and Results

The validation, results, and performance measurements of the PerSyst Tool for the architectures Westmere-EX and Sandy Bridge-EP are provided in this chapter. These are provided for three large HPC architectures: HLRB-II, SuperMIG, and SuperMUC. SuperMUC is the largest and has the most recent architectures of the three systems. Thus, the bulk of the results is provided for the largest system, SuperMUC, where the tool was deployed. The three systems (the context of evaluation) are described in the following section. On the rest of this chapter only the references to the environment which was used is given at each validation or experiment.

7.1 Context of Evaluation

The evaluation of the PerSyst Tool has been done in three different systems, each with a different architecture. The performance validation has been conducted in SuperMIG (Westmere-EX architecture) and SuperMUC (Sandy Bridge-EP architecture). Both systems were integrated into a larger system with one batch scheduler running for both of them. The three systems are:

HLRB-II: The SGI Altix 4700 teraflop system[1] was the first supercomputer where the PerSyst Tool was deployed. The HLRB-II had 9,728 Intel Itanium 2 processors (Montecito Dual Core) divided into 19 nodes (called *partitions*). The partitions were shared among users and had a large shared memory. This machine was connected to a NAS file parallel system. The PBS batch scheduler [118] was managing the resources and the queuing system.

SuperMIG/fat nodes: The Intel's 'The Intelligent Cluster' based on IBM BladeCenter HX5 was used for the second deployment of the tool. This system has Westmere-EX (Xeon E7-4870) processors with 8,200 cores in total divided into 205 nodes, ie. 40 cores per node. All the nodes comprise one island. The batch scheduler which distributes the jobs on this system is the LoadLeveler from IBM and is configured to allow users a dedicated use of the compute nodes, thus, a compute node can't be shared. There are four job classes[2]; each allowing submissions

[1] This machine was retired from operation.

[2] The batch scheduler schedules jobs according to the job class which in turn depends on the job size. A job class allows a range of job sizes which are typically sent to the a sector of the system (for example: two islands are dedicated to the job class "small".)

with a different range of job sizes.

SuperMUC/thin nodes: The IBM X Series Cluster system, SuperMUC, is based on Intel Sandy Bridge-EP (Xeon E5-2680) and Mellanox FDR-10 Infiniband technology. SuperMUC comprises 18 thin-node Islands with 516 nodes each (8256 cores in total in one island). Each node has 2 Sandy Bridge-EP Intel Xeon E5-2680 processors (also known as *packages*), with a total of 16 cores per node (a Sandy Bridge-EP package has 8 cores). Each individual island is connected internally via a FDR10 fully non-blocking infiniband network. Among islands, the high speed interconnect enables a bi-directional bi-section bandwidth ratio of 4:1 (pruned network). SuperMUC has, thus, 9,216 nodes with a total of 148,608 cores in the thin island. The batch scheduler is configured to allow users to use one compute node for a job, thus, a compute node can't be shared.

Integration of SuperMIG into SuperMUC: Both systems were integrated and have a common batch scheduler and interconnect network. There are eight job classes; each allowing submissions with a different range of job sizes for the Westmere-EX nodes (also known as fat-nodes) and for the Sandy Bridge-EP nodes (also known as thin-nodes). The batch scheduler is configured to allow the users to use one compute node for a job. Simultaneous Multithreading (SMT) is activated and users are encouraged to use only one hardware thread per core to run their applications which leaves the second hardware thread for administrative tasks (such as those performed by the OS, the batch scheduler, and monitoring tools).

The IBM General Parallel File System (GPFS) which is connected to both the thin-node and the fat-node islands, has 10 PB of capacity and an aggregated throughput of 200 GB/s. Disk storage subsystems built by DDN were configured for a striping factor of 8 MiB, which is the amount of data written onto each disk before moving to the next disk. The striping block can be split into sub-blocks which are 1/32 of the striping factor; this is the smallest allocation to a single file [73].

7.2 Portability

The PerSyst Tool has been ported to three systems. A more detailed description of the porting of the tool is given for the SuperMUC system where most of this work focuses on.
The tool was initially deployed in HLRB-II with one Collector (which did the synchronization) and 19 PerSyst Agents, one at each partition (equivalent to a node). In this system, the tool was interfacing with the *pfmon* [3] tool from HP which was parallelized with four threads in order to read the performance metrics. Other tools used were *SAR* [33], and *top* [46], both from the Linux OS.
Secondly, the deployment on SuperMIG was achieved with one SyncAgent, 9 Collectors, and 205 PerSyst Agents. In this machine, the tool interfaced with: LIKWID for the performance measurements; the loadleveler from IBM; the *SAR*; and the */proc* virtual file system from the

Linux OS.
Finally, the tool was ported to the SuperMUC petaflop system with one SyncAgent as frontend, 12 SyncAgents in the middle layer, 216 Collectors, and 9216 PerSyst Agents (one at each compute node). In this machine, the tool interfaced with: LIKWID, the loadleveler from IBM, the *SAR* and the */proc* virtual file system from the Linux OS, the *mmpmon* [51] tool from the GPFS, and the *perfquery* tool [92].

The PerSyst Tool runs currently in production mode in SuperMUC; thin and fat nodes now comprising one large supercomputer system. The tool was configured to run as one instance (one tree of agents with one frontend) on both parts of the system and measuring two different architectures (two different types of PerSyst agents; one at each type of architecture).

Table 7.1 shows the measuring tools used in SuperMUC.

Tool	Monitored Events
LIKWID	Hardware Events
mmpmon	I/O GPFS Events
perfquery	Network Events
/proc/meminfo	Memory Usage
SAR	System Activity Report Events

Table 7.1: Measuring tools used

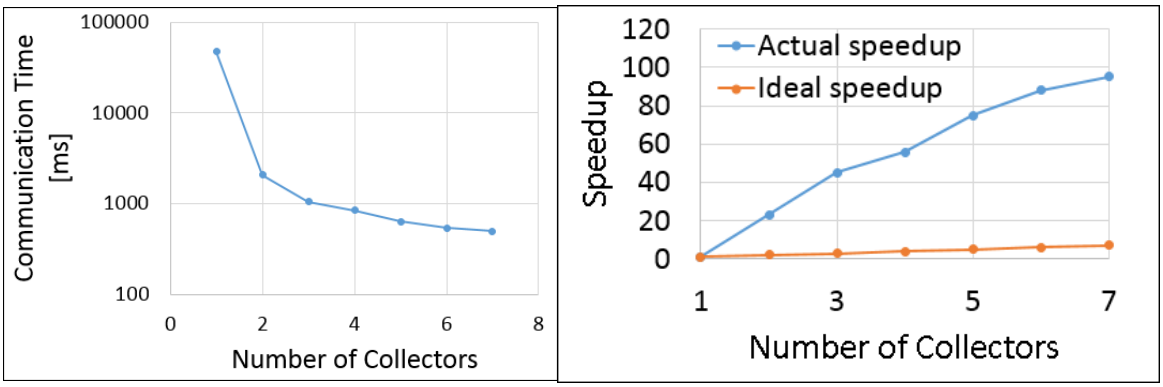
The tools in the fat nodes used are the same as with SuperMUC/thin-nodes (*mmpmon* and *perfquery*). The *mmpmon* [51] tool is provided by the GPFS to read certain events, including read and write bandwidth, number of read and write operations, and operations requesting metadata. The infiniband interconnect provides event measurements via the *perfquery* [92] tool. It provides transmitted/received data and number of transmitted/received packets.

Regarding the experience of deploying a tool in a large HPC architecture, it is recommended to start with part of the system at first; then tune the fanout, and finally scale the tool. In SuperMUC, for instance, the deployment can be done first in three islands in order to calculate the number of agents needed. Furthermore, some sort of fault tolerance approach must be used to keep the tool running over more than one year without management intervention. Finally, logging is required to debug the system. If logging is done centrally (in one directory) a representation system with only a few agents has to be selected to achieve scalability.

7.3 Scalability Tests

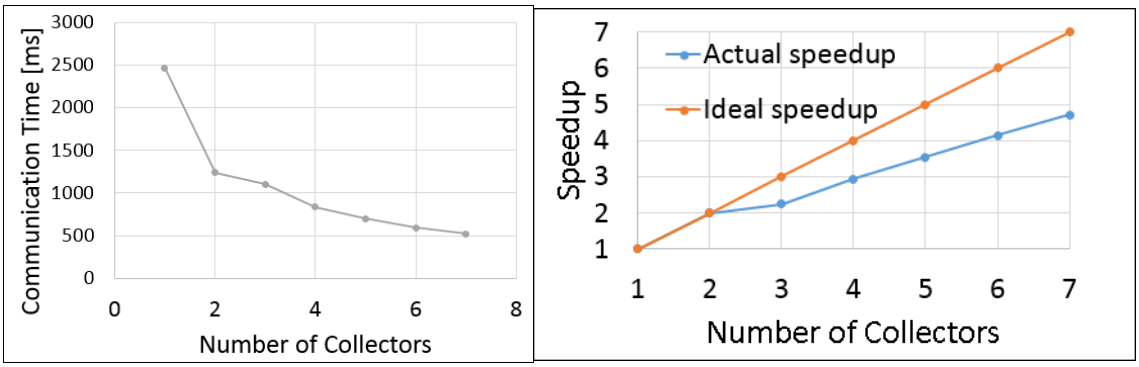
Scalability tests were carried out on the fat-nodes with seven different configurations and the time taken to transmit a command and the job information was recorded. Every configuration had one SyncAgent (as a frontend) and 206 PerSyst Agents. The only variation was the number of Collectors (1 to 7 Collectors were used). Seven measurements were taken and averaged for every configuration except one. The configuration with one Collector Agent (a SyncAgent, a Collector Agent, and 206 PerSyst Agents) was not stable and it was only possible to take

one successful measurement. The agents received late heartbeats from the parent agents and terminated, with the use of only one Collector. The time taken for the job information transmission on all the branches of the tree was recorded and is shown in Figure 7.1a. The number of messages varies with the number of jobs. On average, there are 26 jobs in one point in time in SuperMIG. SyncAgents have to split the job information on average into 3 messages. The Collectors usually send one message with the job information to all the SyncAgents (206 messages for all the PerSyst Agent). The time taken to transmit a command was recorded for all these configurations and the results are shown in Figure 7.2a. There is only one command message per agent. In both cases the measurements were done by instrumenting at the agent which initiates the communication the time taken to send the job information or commands to its child agents.



(a) Time plot. **(b)** Speedup plot.

Figure 7.1: Scalability for job information transmission



(a) Time plot. **(b)** Speedup plot.

Figure 7.2: Scalability of command transmission

Figure 7.1b shows the scalability results with a better speedup than the expected 'ideal' speedup. This is due to the instability of the tool when only using one Collector. The scalability plot (see Figure 7.2b) shows that the command transmission does not scale as well as the ideal speedup. However, the scalability still remains linear. Scalability tests were not fully performed with all configurations in SuperMUC. Having the PerSyst Tool fail with only one Collector on

a systemwide basis will disturb the running applications. Due to the heartbeat system (refer to Section 4.5), the failing agents terminate due to late heartbeats from their parent. The parent agent will eventually replace them. Thus, the placement of agents and their termination will occur in a cycle. Table 7.2 shows the scalability results for three configurations. For all configurations there was one additional SyncAgent acting as the frontend. The 21.5 PerSyst Agents per Collector is an average. Half of the collectors will have 22 PerSyst Agents while the other half only 21.

Tree Configuration				Communication time		
Configuration Id	Number of SyncAgents	Collectors per SyncAgent	PerSyst Agents per Collector	Frontend to SyncAgents [ms]	SyncAgent to Collectors [ms]	Collector Agent to PerSyst Agents [ms]
A	12	36	21.5	108	472	396
B	12	18	43	108	180	631
C	12	9	86	108	107	2658

Table 7.2: Scalability of the tool in SuperMUC.

Table 7.3 shows the aggregated results with the total time a command is communicated downward on the agent tree (total time of propagation) and the total number of agents. These results indicate that the best fanout of the tree is configuration B, which has the lowest total time and the second to lowest total amount of agents.

Configuration Id	Total time of propagation [ms]	Total number of Agents
A	984	9733
B	925	9517
C	2881	9409

Table 7.3: Total time and number of Agents.

The measurements of the data collection include property processing time. The collection of data is not necessarily simultaneous for all agents. Moreover, performance data is not communicated throughout the entire agent tree, but through optimized routes. Thus, only the top-down scalability has been tested. These results show that the agent tree is necessary to enable the scaling of the tool to larger systems.

7.4 Results of the Transport System in SuperMUC

The PerSyst Tool runs in production mode on SuperMUC, analyzing automatically more than 10,000 application runs per month. The fanout of the tree hierarchy consists of one SyncAgent

as a frontend, 12 SyncAgents as a middle layer, 216 Collectors, and 9288 PerSyst Agents at every compute node. The 13 SyncAgents were placed at an external node which is used for administrative tasks. Six Collectors per island were placed having each 43 PerSyst Agents (child agents). Parent-child relations among Collectors and PerSyst Agents were placed in the same island. Thus, the tree agent topology exploited the faster interconnects with these placements. The tool was configured to run and aggregate using deciles (ten percentiles) and at time intervals of ten minutes with the cycle times shown in Table 7.4.

Total cycle time (TC)	600 seconds
Measuring and analysis phase	120 seconds
Idle Phase	480 seconds
Tool basic measurement time	10 seconds

Table 7.4: Times for cycle control

The following sections provide results on the influence of different aspects of the transport system on the performance data.

7.4.1 Optimized Routes

The amount of performance data (property data grouped per job) that an agent type (PerSyst, Collector, or SyncAgent) aggregates and stores was recorded. The average amount of ten observations (at ten different measuring intervals) of property data grouped per job at each agent type is shown in Table 7.5. The collection per job at a single point for quantile aggregation is done only at the Collector Agent level and at the PerSyst Agent level. On average, in one measuring interval, 91% of the performance data of a job is processed centrally at one node. Thus, the estimation of quantiles (as opposed to exact calculations) is on average 91% circumvented.

Tree level	Average amount of property data (Grouped per job)	Percentage
Frontend	4.8	2.59%
SyncAgents	11.8	6.37%
Collector Agents	157.5	85%
PerSyst Agents	11.2	6.04%
Total number of jobs	185.3	100%

Table 7.5: Distribution of jobs in agent tree.

The tree network can be used fully when the performance data of all the jobs is transmitted to the frontend via the connecting tree nodes. The alternative is to try to collect this data at selected nodes with the job load balancing algorithm (described in Section 5.3) and perform

the output when the job has been collected in its entirety. In order to quantify the usage of the agent tree with both methods, the number of nodes where the performance data of a job was being sent was added for all jobs. Table 7.6 shows these figures.

Retrieving method	Average number of nodes used
Data transmitted through established topology connections until frontend	905.81
Data transmitted to selected nodes with job load balancing algorithm	217.59

Table 7.6: Usage of the topology network for 58 measurements taken during a week.

The results show that the usage of the topology nodes is more than a factor of four better compared to the traditional bottom-up retrieval of job information. Optimizations on the collection of data further ensure the scalability of the tool.

7.4.2 Collection Times

For this experiment all the properties were sent to the collector (no property filtering was applied, as described in Chapter 3). At the level of the PerSyst Agent, three different sets of properties are sent: one for the hardware events, one for the SAR, I/O and network events, and one set for the memory consumption. Nine islands were measured and the average of the collection time per PerSyst Agent was taken. Given that there are 43 PerSyst Agents per collector, the incoming amount of data at a collecting point was considered.

Properties	IO, SAR, and Network	Memory Usage	Hardware counter
Bytes	87294	6708	110424
Average time [s]	0.14	0.02	0.85
Used bandwidth [MiB/s]	0.56	0.31	0.12

Table 7.7: Collection time in Collector Agents.

The collection times include property processing, as the properties evaluated and packed into groups. As soon as a group reaches a certain size, they are transmitted. The reason why the bandwidths are low, are due to the asynchronous collection (including processing and grouping time) of properties.

7.4.3 Reduction of Information

To check the reduction of information due to the use of properties, and quantile aggregation; the theoretical amount of all the events was calculated and compared to the collected properties.

- There are a total of 36 events collected for the different devices of SuperMUC. These break down into 22 hardware events, 1 memory event, 6 I/O events, 3 SAR events, and 4 infiniband network events. From the hardware counter events, the 2 memory bandwidth events are collectible per node, and the power events each are collected per package (two packages per node) or per DRAM. The 6 I/O events and the 4 network events are collected per node. If all events were to be collected raw, without any transformation, filtering, or reduction, the amount of data would sum up to 6.7 million data items, see Table 7.8. This is equivalent to 25MiB in floating point with single precision.

- There are 34 properties for SuperMUC. An evaluation of how many jobs were running at a time (on a monitoring interval) resulted on an average of 142 out of 10 observations in different days. The information collected on each job is 11 deciles (10 plus the minimum), the average, and the number of observations: 13 data items. Assuming all had a severity and using the average quantity of jobs that run at one time on the supercomputer, the total amount of data items collected are 142 jobs times 13 data items times 34 properties $= 62,764$. This is equivalent to 245KiB of data in floating point with single precision.

62,764 data items represents only 0.94% of the 6.7 million raw hardware counters. Thus, the reduction of information is of 99.06%.

Event	Domain	Amount of events	Total in SuperMUC
Hardware counter events	Hyperthread	18	5,349,888
	Node	2	18,432
	Socket (2 sockets in a node)	2	36,864
IO events	Node	6	55,269
Network events	Node	4	36,864
Virtual Memory	Hyperthread	1	287,216
SAR events	Hyperthread	3	891,648
		Total amount	6,676,181 Events

Table 7.8: Amount of collected data in SuperMUC without reduction techniques.

The reduction of information due to the selection of thresholds was measured by comparing the number of records generated in a week with the full collection of properties with the records of another week with thresholds being set. In the case of SuperMUC, there were 4,027,187 records with all the property information in a week, and 2,346,044 records with property selection. Thus, the information was reduced by approximately 42%.

7.4.4 Negligible Interference on Running Jobs

This validation tests whether the interference of production applications caused by the PerSyst Tool is statistically negligible. A statistical equivalence test was carried out as described in [117]

to study the interference of the PerSyst Tool. The application used for this test solves a heat propagation equation with the Jacobi method (with an arithmetic intensive kernel) in a two dimensional plane. The validation is performed by comparing the runtime in different nodes without the PerSyst Tool (Group A) and the runtime of the same application in different nodes with the tool running in the background (Group B). The distributional assumption is that the runtime is approximately normally distributed for both groups with mean values μ_A and μ_B and an unknown common variance σ^2.
The evaluation is done with a 1% maximum tolerated deviation of μ_A and μ_B (both 1% less than the lower limit, l, and 1% more than the upper limit, u) and with 20 observations to estimate both μ_A and μ_B.

$$\text{Group } A(n_A = 20) : \bar{A} = 659.871s, S_A = 4.433$$
$$\text{Group } B(n_B = 20) : \bar{B} = 659.880s, S_B = 3.126$$
$$l = 653.271, u = 666.469$$

Note that $\bar{A}$ and $\bar{B}$ are the average values of A and B respectively; S_A, and S_B are the standard deviation of A and B respectively; and n_A and n_B are the number of observations for Group A and Group B respectively. The null hypothesis is that Group B is not equivalent to Group A. The confidence limits for μ_A have been calculated for one sided confidence level of 99% with the central t-distribution:

$$\mu_A - C_l = 662.035, \mu_A + C_u = 662.706$$

Given that $l < \mu_A - C_l$ and $\mu_A + C_u < u$ the null hypothesis can be rejected and the decision is in favor of the equivalence. Thus, the interference caused by the PerSyst Tool on runnning applications is negligible.

7.5 Quality of Quantile Estimation

The tool uses quantile estimation at the SyncAgent layers of the tree. At these layers the performance data received are the quantile aggregation of the child nodes. This section evaluates the quality of the quantile estimation. In order to evaluate the quality of estimating quantiles 1,000 random normally distributed numbers were used. The random values were divided into ten groups (i.e. 100 elements in each group). The grouping was performed without ordering the random numbers: they were grouped in a round robin fashion as they were generated. The meta-aggregation of deciles of the ten groups was done as well as the aggregation of the original array containing the 1,000 random values. The results are shown in Figure 7.3. It can be seen that the quality of the distribution is preserved by the estimation.

7.6 Validations of Performance Measurements

In this section, the validation of the performance measurements are presented. These validations are not exhaustive for both architectures presented (Westmere-EX and Sandy Bridge-EP)

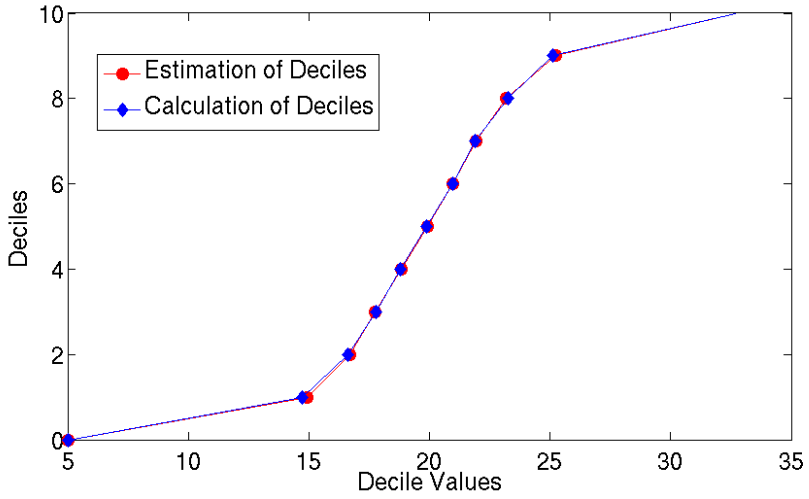


Figure 7.3: Evaluation of Quantile Estimation

but most of the important metrics have been covered. The measurements were done with direct instrumentation of the application and using the same measurement library as in the PerSyst tool. Not every single metric was validated especially if it is known that they produce erroneous results (for example: expensive instructions count in Westmere-EX, and flop count in Sandy Bridge-EP). The validation of energy consumption is not presented in this document, an extensive validation has already been done [47, 83]. All of the tests that were conducted with SMT enabled[3].

7.6.1 Flops

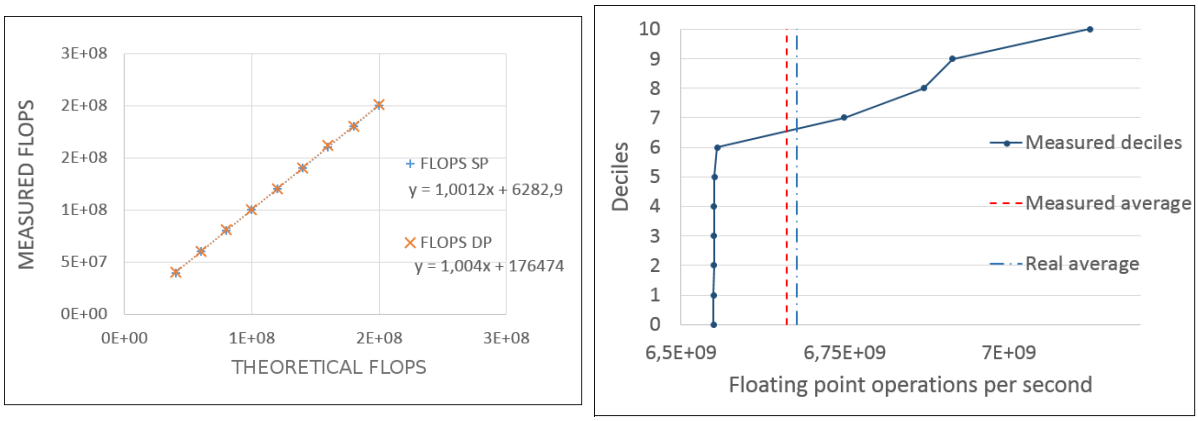
The validation of flops has only been done for the Westmere-EX architecture directly by measurements done with the PerSyst Tool. Others [19, 111] have already uncovered the problem of over counting of flops in the Sandy Bridge-EP architecture. Although the measurements have been implemented as an estimation of the real floating operations, it will not be covered, nor validated, in this section. Figure 7.4a shows a validation without the PerSyst Tool, of the single precision and double precision triad at different array sizes (See Listing 7.1). The interface to LIKWID (used by the PerSyst Tool) was used instead to directly measure the triad. The amount of real flops/s against the measured flops is shown in Figure 7.4 and the linear fits show that both slopes are close to one (both fits rendered $R^2 = 1$, a perfect fit). Figure 7.4b shows the flops measured on an application; *pmatmul*. *pmatmul* is a matrix multiplication where the exact amount of flops is known and printed out for testing purposes. The experiment was set with 4 fat nodes (160 cores) and the average of the flops/s within the measured lapse is shown in Figure 7.4 as the actual average. Measured average and deciles are also shown.
The results with a mixed amount of single and double precision are very similar to those shown in graph Figure 7.4a. Listing 7.1 was modified to have a and b arrays declared as double (double precision), and scalar and c was declared as float (single precision). The fit produced a slope

[3]Due to special operation conditions, the SMT can not be turned off.

of approximately one with $R^2 = 1$ (the linear regression was $y = 1,001x + 34673$, where y is measured and x is real.)

Listing 7.1: Triad used for measuring flops

```
for (j=0; j<STREAM_ARRAY_SIZE; j++)
    a[j] = b[j]+scalar*c[j];
```



(a) Validation without the PerSyst framework

(b) Validation with the PerSyst Tool

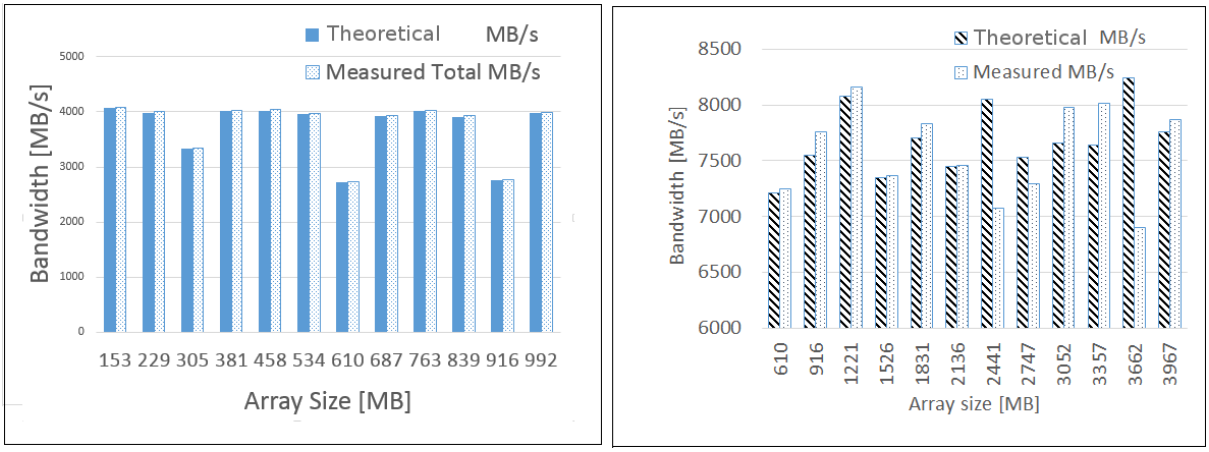
Figure 7.4: Validation of Flops/s in Westmere-EX.

7.6.2 Memory Bandwidth

The highest memory bandwidths can be determined with the STREAM benchmark [70]. This benchmark provides four different types of operations: copying an array from one to another, multiplying an array with a scalar, adding two arrays, and the triad (which consists of scaling an array and adding it to another one). The triad operation was used with different array sizes and measured with the measuring interface to LIKWID. Figure 7.5 shows the results of the total bytes divided by the time the STREAM benchmark took to perform the triad operation (shown as theoretical MB/s). The figure also shows the counter based measurements (shown as Measured MB/s). These measurements are shown for both architectures, Westmere-EX and Sandy Bridge-EP. In the case of the Westmere-EX, all of the measurements were under a 1.1% error. For the Sandy Bridge-EP architecture, the average error over all measurements is 4%, with only two measurements significantly above the theoretical measurement (array size 2441 MB with 12% error and 3662 MB with 16%), otherwise all of the measurements were under 5% error.

7.6.3 L3 Bandwidth

The STREAM benchmark was used to evaluate the L3 bandwidth. This experiment was done by using the triad (see Listing 7.1) and by changing the array size of the three vectors. The interface to LIKWID was used to validate this metric. The total theoretical amount of data



(a) Memory bandwidth measurements in the Westmere-EX architecture.

(b) Memory bandwidth measurements in Sandy Bridge-EP architecture

Figure 7.5: Memory Bandwidth with STREAM and the LIKWID interface.

loaded to the L2 cache was calculated (array size * 3 arrays * size of float). The total measured data is calculated with 64 * (L2_LINES_IN_ALL + L2_LINES_OUT_DEMAND_DIRTY) (the same as the property value, see Section 3.3.16 without dividing by time). The results are shown in Figure 7.6, the error between the theoretical amount of data loaded to L2 and the measured values remains at about 5% for all measurements. This can be seen in the graph and explains why the higher the array size, the more the gap separates. The over-counting of a constant 5%

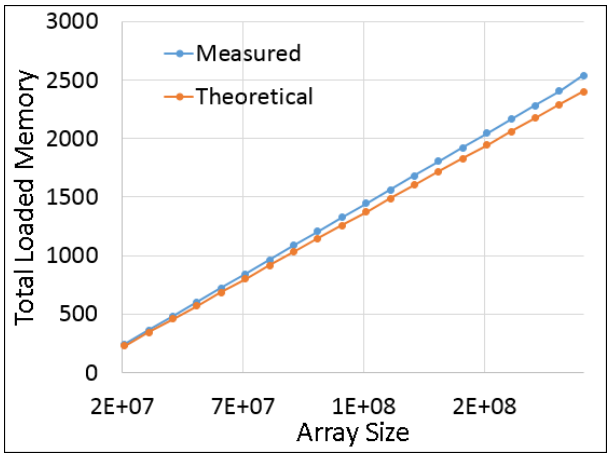


Figure 7.6: Validation of L3 Bandwidth in Sandy Bridge-EP

of the data loaded is not considered critical and therefore useful for detecting L3 bandwidth related bottlenecks.

7.6.4 Instruction Count

The validation of instructions was performed with the kernel shown in Listing 7.2.

Listing 7.2: Kernel for measuring number of instructions

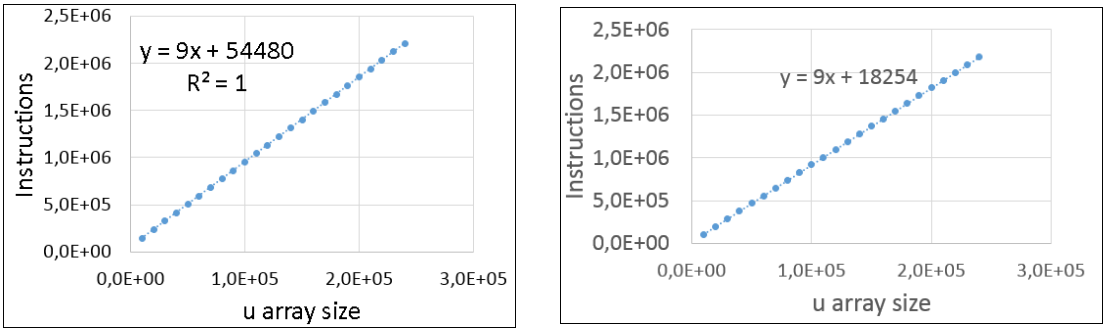
```
for (j = 0; j < size; j++) {
    u[j] = 0.05 * u[j] + j;
}
```

The corresponding assembly language (Listing 7.3) shows that there are nine instructions per iteration. Thus, the number of instructions at runtime increases by a factor of nine with the array size (iterating from 0 to $size - 1$).

Listing 7.3: Assembler code for measuring the number of instructions

```
.L3:
        cvtsi2sd        %eax, %xmm1     # j, tmp67
        addl    $1, %eax        #, j
        movapd  %xmm2, %xmm0    # tmp69, tmp65
        mulsd   (%rdi), %xmm0   #* ivtmp.42, tmp65
        addsd   %xmm1, %xmm0    # tmp67, tmp65
        movsd   %xmm0, (%rdi)   # tmp65,* ivtmp.42
        addq    $8, %rdi        #, ivtmp.42
        cmpl    %eax, %esi      # j, size
        jg      .L3     #,
```

Measurements on the amount of instructions were made in order to determine whether the instruction count is correctly measured. Figure 7.7 shows the measurements with different array sizes (array u in Listing 7.2) for both architectures: Westmere-EX (Figure 7.7a), and Sandy Bridge-EP (Figure 7.7b). The linear regression resulted in a slope of exactly nine instructions per iteration (see assembly code 7.3) for both architectures, as expected. The coefficient of determination equals one—a perfect fit—in both cases ($R^2 = 1$).



(a) Instruction count validation in Westmere-EX **(b)** Instruction count validation in Sandy Bridge-EP

Figure 7.7: Validation of instruction count using the LIKWID interface of the PerSyst Tool.

7.6.5 Load Imbalance Properties

The validation of the imbalance properties was done with the aid of the APART Test Suite [37], hereafter ATS. ATS was designed to generate OpenMP or MPI communication patterns, with the intention of creating common bottlenecks that could be tested by automatic performance

tools. Figure 7.8 shows the block distribution used (named as b2 in ATS), with the first half of the MPI tasks performing 10% of the work that the second half of the tasks were performing, i.e. low = 0.1 and high = 1. The pattern used creates a load imbalance at an `mpi_barrier` (the `imbalance_at_mpi_barrier` function of the ATS was used).

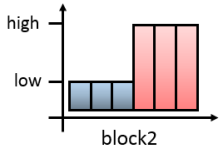


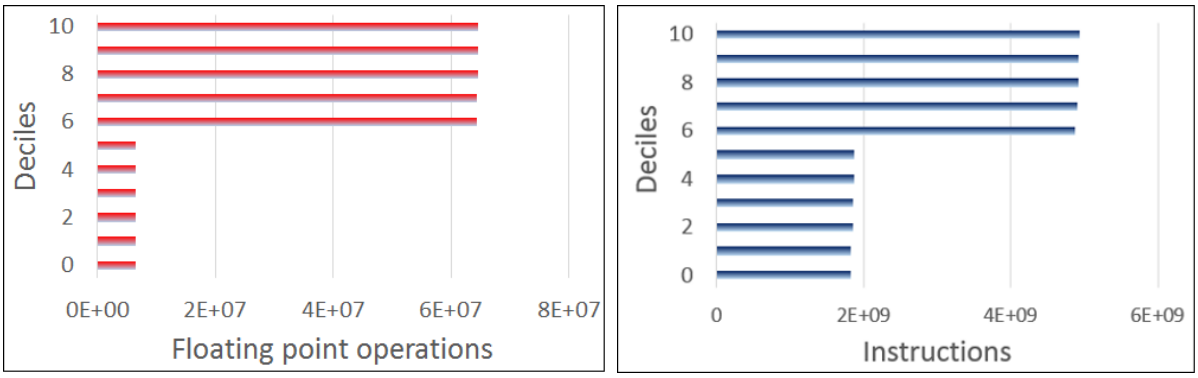
Figure 7.8: ATS load imbalance distribution pattern used to create and test the load imbalance properties.

The ATS function was called one thousand times in a for-loop to ensure that it lasted long enough to be measured by a cycle of the PerSyst Tool. Two lines in the ATS were changed in order to emulate codes with floating point operation-intensive kernels. The `do_work()` function which runs a for-loop exchanging integral values at random positions was changed so that it runs longer. The arrays were declared as floats and a floating point operation was introduced, a division by (i+10), as illustrated in Listing 7.4.

Listing 7.4: ATS do_work kernel modifications

```
for (i=0; i<N; ++i) {
  arrA[myrand() % ARR_MAX] =
    arrB[myrand() % ARR_MAX]/(double)(i+10);
}
```

To validate the intra-node performance property, the number of floating point operations in the Westmere-EX architecture and the number of instructions in the Sandy Bridge-EP architecture were measured for the modified ATS code and the imbalance performance pattern as previously described (See Listing 7.4). For the Westmere-EX architecture the job was set up to run in one fat node of SuperMUC with 40 cores. The property evaluates the difference of the maximum value and the minimum value, which was 58 MFlop/s for this job. Figure 7.9a shows the deciles measured and aggregated by the PerSyst Tool. The first five deciles are about 10% of the upper five deciles, which corresponds to the setting chosen for the ATS test (on average it was a ratio of 1:9.997 of the first median with respect to the second median). For the Sandy Bridge-EP architecture, the job was set up to run in one thin node of SuperMUC with 16 cores. The instruction count shows a marked difference for the first median (0th to 5th decile) compared to the second median (6th to 10th decile) as shown in Figure 7.9. However, the ratio 1:10 is not preserved due to executed instructions outside from the kernel are included in the measurements. Floating point operations were not used on Sandy Bridge-EP for this property due to over-counting of the events. The property evaluates the difference between the maximum value with the minimum value which was 3.67 Giga Instructions in this case.



(a) Westmere-EX architecture **(b)** Sandy Bridge-EP architecture

Figure 7.9: Validation of intra-node imbalance

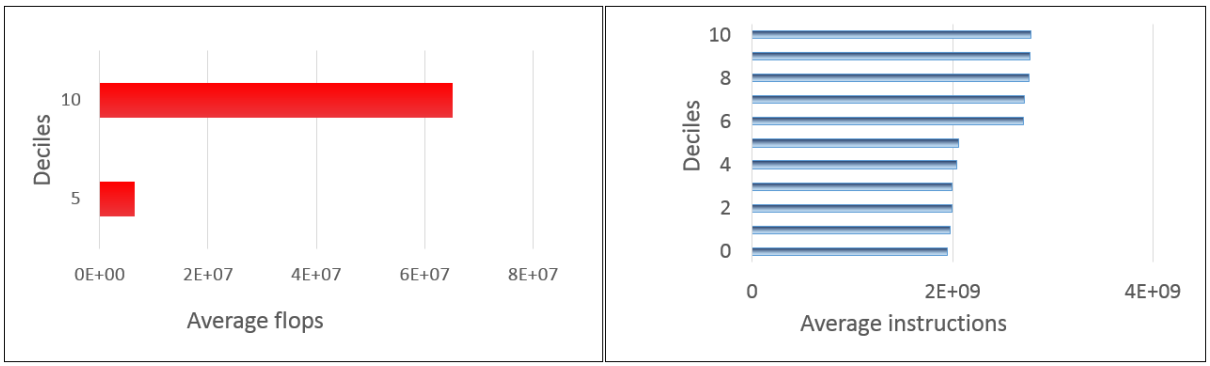
The inter-node imbalance measured by the PerSyst Tool properties was also validated using the ATS code which was used for the intra-node imbalance. The difference in the experiment is that two fat nodes (Westmere-EX) were used to validate the inter-node imbalance. On the other hand, 20 thin nodes (Sandy Bridge-EP architecture) of SuperMUC were used for the job to validate the internode imbalance property that uses the instruction count. Figure 7.10a shows the measured flops/s and Figure 7.10b shows the instructions retired in average at each node. This information is shown as deciles given that the PerSyst Tool extracts aggregated data. Given that two nodes will repeat the first five deciles (and also the last five with the same value), only the 5th and 10th deciles are shown. The total average of flops calculated by the internode imbalance property was the same as the average calculated by the flop count (avg = 35MFlops). Likewise, the total average of instructions calculated by the internode imbalance property was the same as the average calculated by the instruction count (avg = 2.374 GInstructions). From Figure 7.10b it is possible to determine that there is load imbalance, however, it does not correspond to the proportions of 0.1:1 that was emulated with the ATS test suite, while Figure 7.10a does preserve this ratio. Thus, it can be concluded that using the flop count to evaluate the imbalance property is more accurate than with the instruction count on flop-intensive applications.

The experiment was repeated by using the flop count (as opposed to using retired instructions) for inter-node imbalance in the Sandy Bridge-EP architecture. The ratio of the imbalanced tasks preserves the 10% which was set on the ATS code. Figure 7.11 shows the measurement of one monitoring interval.

In conclusion, a more precise detection of load imbalance can be done by using flops on flop intensive applications.

7.6.6 Expensive Instructions Property

The expensive instruction validation was carried out with a kernel with four divisions in a loop (see Listing 7.5). The kernel was compiled with the -O0 flag to avoid compiler optimizations. The total number of divisions was $4 \cdot (n-1)$, where n is the number of iterations loops should



(a) Westmere-EX architecture **(b)** Sandy Bridge-EP architecture

Figure 7.10: Validation of inter-node imbalance property

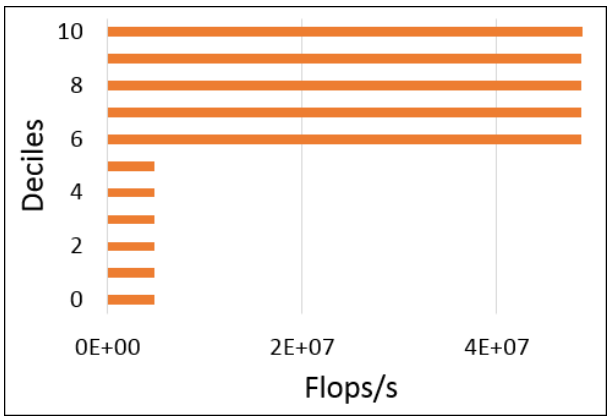


Figure 7.11: Validation of inter-node imbalance for Sandy Bridge-EP using floating point operations.

run (variable `n` in Listing 7.5). Figure 7.12 illustrates the results of counting the expensive instructions; the graph shows how expensive instructions are strongly correlated to the total number of divisions performed in the kernel. The expensive instruction event counter is about 15 times the real number of divisions. The measurements were only carried out in the Sandy Bridge-EP architecture, given that the event may produce inaccurate results when SMT is enabled [4].

Listing 7.5: Kernel for expensive instructions

```
for (int i = 1; i < n - 1; i++) {
  for (int j = 1; j < n - 1; j++) {
    u[i*n+j] = u[i*n+j-1]/1.333 + u[i*n+j+1]/3.421
      + u[(i-1)*n+j]/9.786+u[(i+1)*n+j]/0.236;
  }
}
```

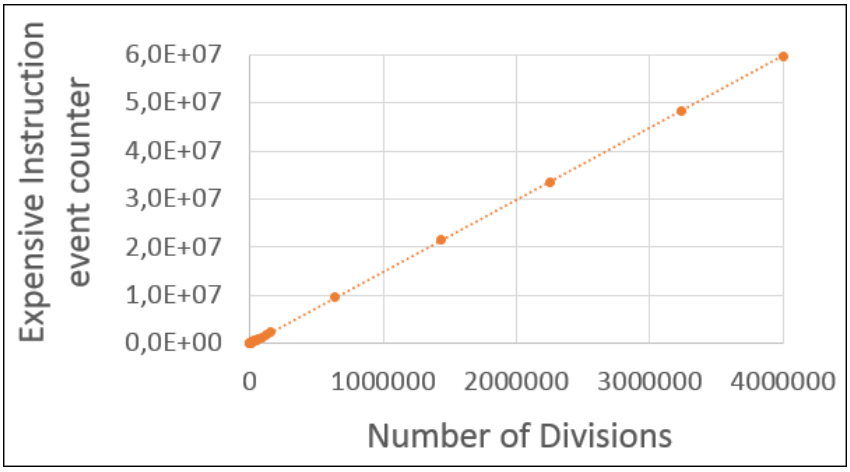


Figure 7.12: Expensive instruction validation with the LIKWID interface

7.6.7 Loads To Stores Ratio Property

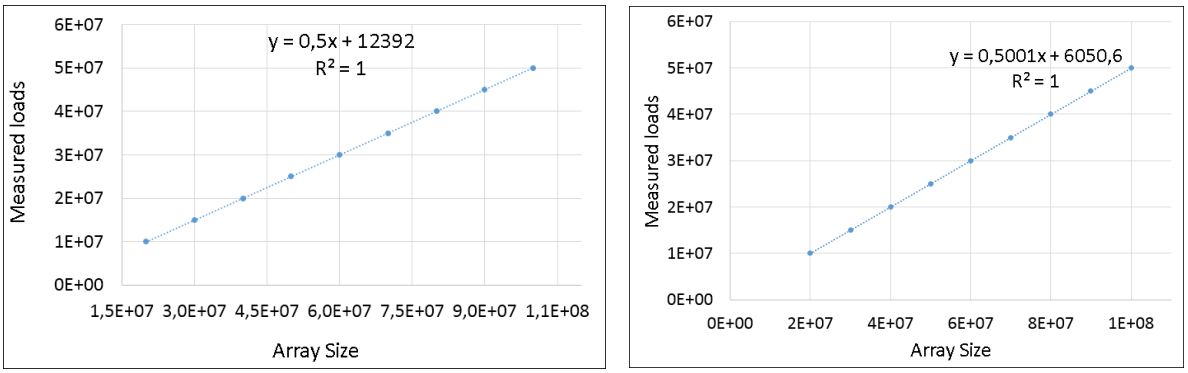
Loads and stores are validated separately with the assumption that their ratio will produce correct results if they produce precise counts. The triad in the STREAM benchmark was used and compiled with `O3`. The `O3` optimization compiled the code to SSE instructions (*movaps*, *addps*, *mulps*) which optimize the code to execute four loop iterations in one cycle. Listing 7.6 shows the assembly code of the triad function which was compiled with an array size of 20,000,000 floats.

Listing 7.6: Assembler code for measuring number of instructions

```
.LFB35:
        movaps  .LC0(%rip), %xmm1       #, tmp68
        xorl    %eax, %eax      # ivtmp.65
```

```
        .p2align 4,,10
        .p2align 3
.L2:
        movaps  %xmm1, %xmm0    # tmp68, tmp65
        mulps   c(%rax), %xmm0  #, tmp65
        addps   b(%rax), %xmm0  #, tmp65
        movaps  %xmm0, a(%rax)  # tmp65,
        addq    $16, %rax       #, ivtmp.65
        cmpq    $80000000, %rax         #, ivtmp.65
        jne     .L2     #,
        rep
        ret
.LFE35:
```

Figure 7.13 shows the array sizes and the corresponding measurements of loads in the Westmere-EX (Figure 7.13a) and the Sandy Bridge-EP (Figure 7.13b) architecture. Note that only one load every two iterations is done with the vectorized instructions, hence the slope of the line is 0.5 in both architectures. The regression shows a perfect fit for the array sizes with respect to the measured loads in both architectures. Thus, the loads were measured correctly.

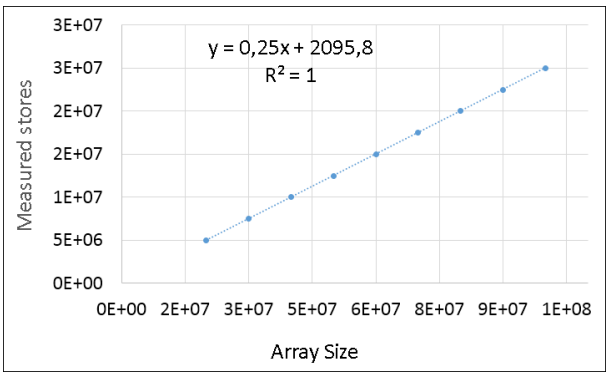


(a) Validation of loads in Westmere-EX **(b)** Validation of loads in Sandy Bridge-EP

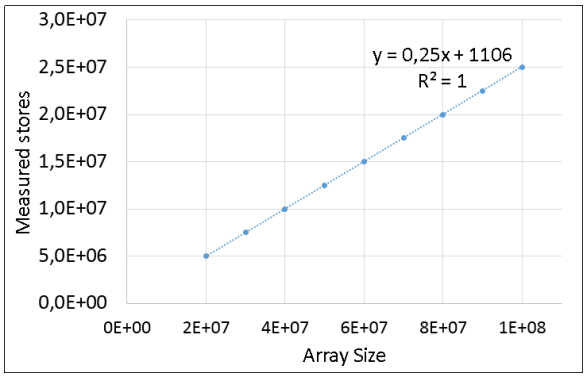
Figure 7.13: Validation of loads using the LIKWID interface with the PerSyst Tool

The same code and compilation options were used to measure the stores. Figure 7.14 shows the array sizes and the corresponding measurements of stores in the Westmere-EX (Figure 7.14a) and the Sandy Birdge-EP (Figure 7.14b) architecture. One store is done every four iterations due to the vectorized instructions, hence the slope of the line is 0.25 in both architectures. The regression shows a perfect fit for the array sizes with respect to the measured stores in both architectures. Thus, the store count was validated successfully.

Given that regressions applied to the measurement data fitted with a coefficient of determination equals one ($R^2 = 1$), the stores and loads were measured correctly. This, in turn, validates the loads to store ratio.



(a) Validation of stores in Westmere-EX

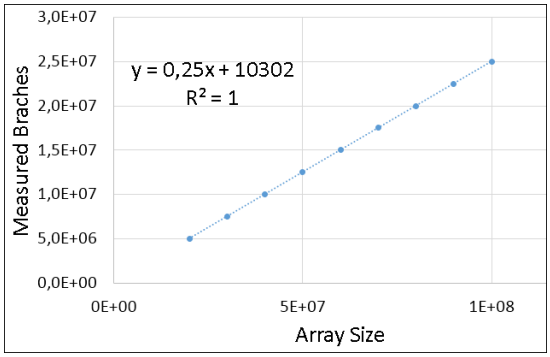


(b) Validation of stores in Sandy Bridge-EP

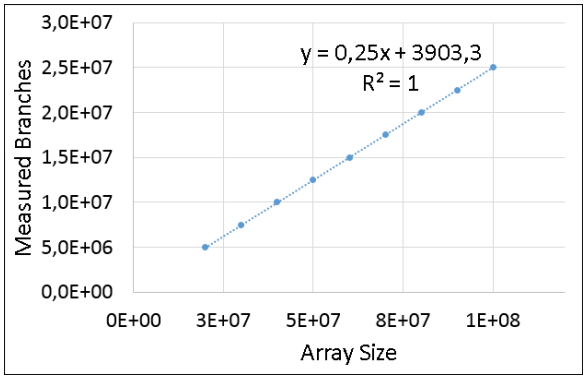
Figure 7.14: Validation of stores using the LIKWID interface with the PerSyst Tool

7.6.8 Branches

The triad in the STREAM benchmark was used and compiled with the -O3 flag to measure the branches. The `O3` optimization produced SSE instructions (*movaps*, *addps*, *mulps*) which optimize the code to execute four loop iterations in one cycle (see Listing 7.6). Figure 7.15 shows the compilation with different array sizes and the corresponding measurements of the number of branches for the Westmere-EX (Figure 7.15a) and the Sandy Bridge-EP (Figure 7.15b) architecture. The vectorization allowed four loop iterations to be done in one cycle. Thus, there is only one branch every four iterations. This is why the slope of the line shown in Figures 7.15a and 7.15b is 0.25. The regression was done with a perfect fit ($R^2 = 1$) in both architectures. Thus, the number of branches counted are reliable.



(a) Validation of branches in Westmere-EX



(b) Validation of branches in Sandy Bridge-EP

Figure 7.15: Validation of branches using the LIKWID interface with the PerSyst Tool

7.6.9 Branch Misspredictions

A benchmark provided by [48] was used to generate branch misspredicions. Arrays are initialized randomly with equal probability of being either 1 or -1 as shown in Listing 7.7, line 4. The four branches are in lines: 14, 17, 19, and 24. From these four branches it is expected that

approximately one branch at each inner iteration is misspredicted with a probability of 50% (the branch at line 19). The assumption is that all probabilities are equal. If $P(A)$ is the probability of misspredicted (prediction) ($P(A')$ is the probability of a good prediction) and $P(B)$ is the probability of having `c[i] < 0` (therefore, $P(B')$ is the probability of having a `c[i] >= 0`), then $P(A/B) = P(A'/B) = P(A/B') = P(A'/B')$. Thus, the probability of having misspredicted branches ($P(A)$) is 0.5 for one branch. If one out of four branches has a probability of being misspredicted, the ratio of misspredicted branches to the total branches should be roughly 0.125 (i.e. half the number of branches of the loop are misspredicted while the total amount of branches is 4 times the number of loops $(0.5 * L)/(4 * L) = 0.125$, where L is the number of inner loops).

Listing 7.7: Benchmark for branch misspredictions

```
int main(){
  //...
  for(int i=0; i<size; i++)
    a[i]=b[i]=c[i]=d[i]=rand()/double(RAND_MAX)*2.0-1.0;
  //...
  //start measuring
    do_triad(a,b,c,d, size, size);
  //stop measuring
  //...
}

  void do_triad(double *a, double *b, double *c,
                double *d, int size, int niter){
  for(int j=0; j<niter; j++){
#pragma novector
#pragma vector temporal
    for(int i=0; i<size; i++) {
      //Branch with 50% probability of being negative
      if(c[i]<0.0)
        a[i]=b[i]-c[i]*d[i];
      else
        a[i]=b[i]+c[i]*d[i];
    }
    if(a[size>>1]<-10.0){
      dummy(a,b,c,d);
    }
  }
}
```

The results show that the misspredicted rate converges to 0.2 (almost 62% more misspredictions as expected). In all likelihood this means that the behaviour of the predictor is not as expected. A second possibility is that the misspredictions are overcounted. In order to calculate the theoretical ratio of misses to total branches, the behaviour of the predictor must be

known. By removing any other branch in Listing 7.7 (for example a *for* loop or the *while* loop) the number of misspredicted branches drops to almost zero. It could not be verified that the measured hardware counters are reliable.

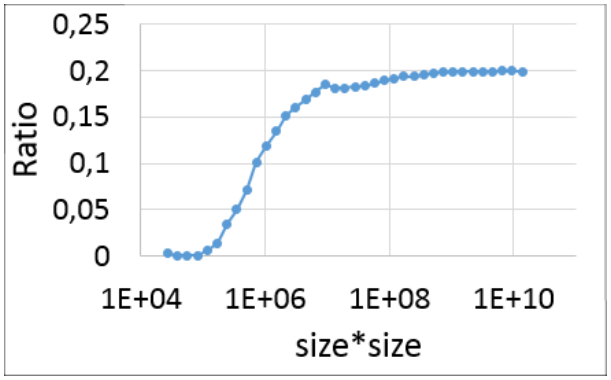


Figure 7.16: Validation of misspredicted instructions for Sandy Bridge-EP

7.6.10 Core Frequency Property

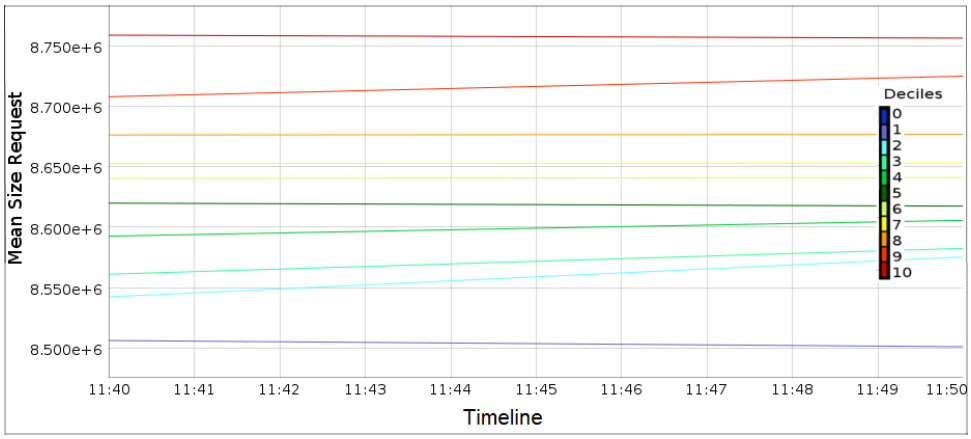
The frequency can be changed via the *CPUfreq* subsystem [23] provided by the Linux Kernel (2.6 or above). In order to have a constant frequency, the *userspace* governor was used. This governor, or policy for setting the frequency, allows the setting of a constant frequency. Running a *sleep* may put the processor on an idle state (also known as C states) even at *userspace* [50,87]. Thus, the measurements were carried out with a matrix-matrix multiplication with 16 MPI tasks. Every MPI task was running a matrix multiplication and there was no communication involved. All of the measured frequencies matched the set *userspace* frequency, when the frequency was set to all the cores in the processor as shown in Table 7.9. However, when setting two or more different frequencies in a core of the same processor, only the highest frequency was measured for all the cores.

Frequency set GHz	Measured Frequency GHz
1.5	1.5
1.7	1.7
1.8	1.8
2.1	2.1
2.3	2.3
2.5	2.5
2.6	2.6
2.7	2.7

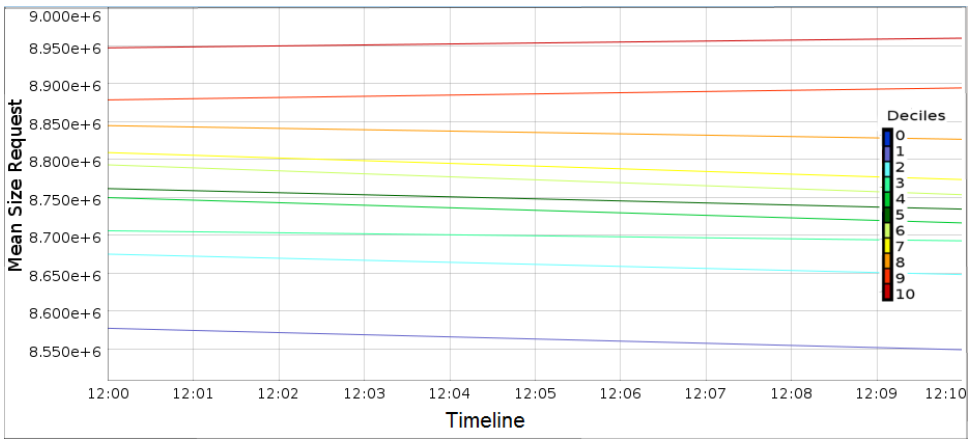
Table 7.9: Set frequency compared to measured frequency.

7.6.11 I/O Bytes per Operation

The IOR benchmark [104] was used with the HDF5 library and instrumented with Darshan [20] to obtain the size of the requests. The benchmark was configured to write and read files with transfer sizes of 8MiB (access sizes) with one file of 32TiB (32GiB from each process using 1024 processes).
The benchmark was set to perform first write operations and then the read operations.
The typical size detected by Darshan are access sizes of 8,388,608 for reading and writing. The results measured directly with the PerSyst Tool are shown in Figure 7.17. The graphs show in the horizontal axis the timeline with the first measurements being write operations (Figure 7.17a), and the second measurements being read operations (Figure 7.17b). Both graphs also show the aggregated measures with ten percentiles (deciles) of access sizes on the vertical axis with each line representing a decile. The quantiles measured show a range of values of [8.5MiB, 8.76MiB] for writing and [8.55MiB, 8.95MiB] for reading. These results correspond to an error of [1.34%, 4.32%] for writing and [1.9%, 6.7%] for reading.



(a) I/O Written Bytes per Write Operation

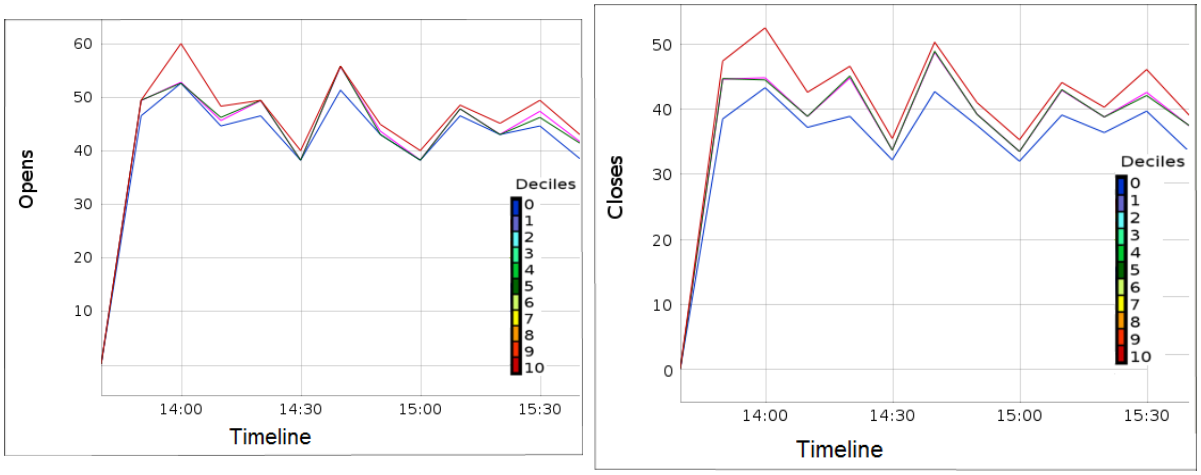


(b) I/O Read Bytes per Read Operation

Figure 7.17: Validation of I/O Bytes per operation

7.6.12 I/O Opens and Closes

To validate the I/O Opens and Closes the IOR benchmark [104] was used. The benchmark was instrumented with Darshan [20] and was configured to write and read files with transfer sizes of 8MB (access sizes) with files of 16GB and using 1024 processes. Darshan reports a total of $20,480,000$ open operations. Given that all open operations are paired with a close operation, the same amount of closed operations were performed. The 1024 processes are measured by the mmpmon [51] in groups of 16 (16 cores per node). Thus, the total amount of opens (and closes) per node amounts to 320,000. The job had a total duration of 7,523 seconds, which means that roughly 43.5 opens (and thus 43.5 closes) per second were performed. The results measured directly with the PerSyst Tool are shown in Figure 7.18. The outliers were removed (first measurement at 13:50) as the opens and closes started after the measurement. The average of opens and closes among nodes and time (total average) obtained from the measurements of the tool was 46.3 opens and 41.0 closes. These results have only 6.4% and 5.9% error to the expected value for opens and closes respectively.



(a) I/O Open Operations

(b) I/O Close Operations

Figure 7.18: Validation of I/O per open and per close operation

7.6.13 Memory Usage Property

The memory usage was validated with the STREAM benchmark by using different array sizes. The first five runs were done with arrays in the stack, while the last three were done with the array allocated on the heap (allocating more than 2.9GiB on the stack was not permitted by the compiler). By reading the */proc/meminfo* virtual file of the operating system, the tool can find out the usage of memory for the entire compute node. This not only includes the user processes (application processes) but also the memory of the root processes and the batch scheduler. Thus, to design the experiment the same amount of increment of memory were used per run, expecting to have a proportional increment of the total memory (it was assumed that the root

processes' memory remains constant). Figure 7.19 shows the result with a directly proportional increase in the measured memory with respect to the memory increase in each run (the slope is roughly 1 and $R^2 = 0.99$). The increase in memory for each run was of 228.9 MiB (from 2059.9 MiB for the first experiment and 3662.1 MiB the last experiment).

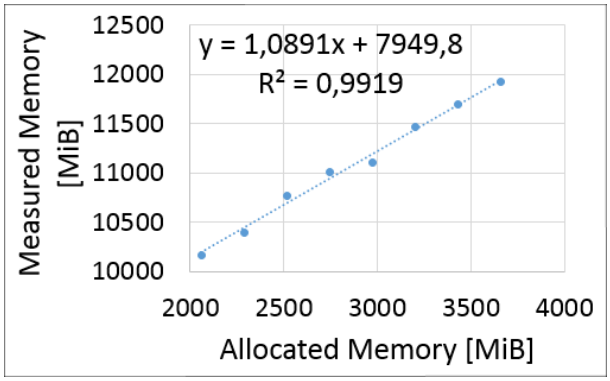


Figure 7.19: Validation of memory usage

—

The foregoing sections provided a validation of individual metrics and properties. Most of the individual metrics were validated, therefore, the combinations of them are also valid. It can be concluded that load imbalance in arithmetic intensive operations can best be detected with floating point operations, even if the architecture over-counts the events of floating point operations. In general, the property value formulae could render false positives for intra-node imbalance. For example: a job does not use the entire node for calculations due to memory requirements will report load imbalance (some cores are idle). In this case the memory usage property will help sort out the false positive. Another case is when an increasing memory usage does not necessarily indicate a memory leak. These two aren't bottlenecks but will be reported as such. The performance expert must compare all information available to sort out false positives. However, it is inevitable to have them with the current scheme. This is why only recommendations can be given and the tool aims at a preliminary detection of bottlenecks as opposed to a detailed analysis. A detailed analysis with instrumentation to applications which are considered inefficient is needed to establish the real causes or to discard potential false positives.

In order to validate branch misspredictions it is necessary to emulate the predictor to calculate the theoretical misspredicted branches. A similar problem applies when trying to validate the L3 cache hits and L3 cache misses. The hardware prefetcher makes it difficult to estimate the theoretical cache hits and cache misses (even when trying to randomize the access to an array).

Some of the properties can be enhanced to detect the bottlenecks more precisely. For example: the intra-node load imbalance property can be enhanced by adding a term into the

equation which does not only measure the difference between floating point operations among cores. When most of the cores are idling the severity should be greater than the severity of only one core idling. The idea is to integrate into the severity a measure of the number of idling cores with respect to the total number of cores in the node.

7.7 Selection of Thresholds

Thresholds are used to decide whether a property has a severity or not. They are selected according to one of the following heuristics:

- A threshold can be based on the hardware characteristics and expert knowledge. Keyword: *Hardware*.
- A threshold can be based on a benchmark. Keyword: *Benchmark*.
- A threshold can be chosen at the point where the performance does not significantly change when improving the property value. In this case, the definition of what is significant is based on the decision of the performance expert. Keyword: *Performance*.
- A threshold can be chosen based on statistical data. Keyword: *Statistics*.

Note that the keywords are used in the following tables to describe how the thresholds were set.

Memory Bandwidth Threshold: The threshold to split the memory bound from the compute bound codes can be determined by saturating a compute node with memory accesses. This threshold is, thus, more related to the hardware characteristics. STREAM was used to calculate the threshold. The bandwidth average value of all the STREAM kernels (copy of an array to another array, adding of two array, scaling of an array, and the triad) when a node is saturated was used. This value was divided by the number of cores in the node to obtain the threshold. Table 7.10 shows the threshold values for each architecture. The threshold to decide whether

	Bandwidth per node	Bandwidth per core	Used heuristic
Westmere-EX	64 GB/s	1.625 GB/s	*Benchmark*
Sandy Bridge-EP	65 GB/s	4.023 GB/s	*Benchmark*

Table 7.10: Average memory bandwidth from STREAM

the code is compute bound is if the memory bandwidth is less than 1.625 GB/s or 4.023 GB/s for Westmere-EX and Sandy Bridge-EP respectively.

I/O Thresholds: Selecting the threshold for the mean read request size and mean write request size has been done by comparing the performance of different request sizes. The technical report with results using the same GPFS system at the Leibniz Supercomputing centre [73] shows the best performance when request sizes are between 2MB and 32MB. Therefore, the

threshold has been chosen to 2MB. Table 7.11 shows the I/O related thresholds with the heuristic used for each property threshold.

Property	Threshold value	Used heuristic
IO Read Bytes	1.1 GiB/s	*Statistics*
IO Written Bytes	1.1 MiB/s	*Statistics*
Mean Size IO Reads	2 MiB	*Performance*
Mean Size IO Write	2 MiB	*Performance*

Table 7.11: I/O Thresholds

If the I/O read or write bandwidth is below the given threasholds, the corresponding mean size of of the I/O operations will be checked. I/O operations with less than 2MiB will produce a recommendation to consolidate the I/O requests.

Other Thresholds: Table 7.12 shows the threshold values for each property. The values determined by using statistical data (keyword *statistics*) were taken from the Sandy Bridge-EP architecture, by gathering the values of one week of measurements and taking the value with the Pareto rule: the 80th percentile is taken as a threshold and any value below is not considered a bottleneck for increasing severity with increasing property value. On the other hand, the 20th percentile is taken as a threshold for increasing severity with decreasing property value.

Property	Threshold value	Used heuristic
Flops	0.15*Peak Flops	*Hardware*
Instructions	0.25 * Peak Instructions	*Hardware*
Core Frequency	(Max. frequency - Min. frequency)*0.7 + Min. frequency	*Hardware*
Memory Usage	0.5 * Max. memory	*Hardware*
CPI	1.6	*Statistics*
Expensive Instructions	$1.7 \cdot 10^8$	*Statistics*
Vectorized Flops	0.01	*Statistics*
AVX to SSE Ratio	0.013	*Statistics*
SP to DP Ratio	0.035	*Statistics*
Intra Node Imbalance	0.5*Peak Instructions	*Hardware*
L3 misses to Instructions Ratio	0.0003	*Statistics*
L3 Cycles Ratio	0.03	*Statistics*
L3 Bandwidth	1.43 GiB/s	*Statistics*
L3 Hits to Misses	0.35	*Statistics*
Loads To Misses Ratio	124	*Statistics*
Loads To Stores Ratio	4	*Statistics*
Mean Size Transmitted Packets	256	*Hardware*
Mean Size Received Packets	256	*Hardware*
Transmitted Bytes	100MB	*Statistics*
Received Bytes	100MB	*Statistics*
SAR User %	80%	*Statistics*
SAR System %	20%	*Statistics*
SAR IO Wait %	1%	*Statistics*

Table 7.12: Thresholds and how they are selected

7.8 Use cases of the PerSyst Tool

The PerSyst Tool analyzes automatically on SuperMUC more than 10,000 application runs per month. This section provides results of the collected data from the automatic analyzes. The two types of uses are demonstrated within this section. Firstly, the detection of bottlenecks of an application. Secondly, a comprehensive view of three properties is given. Appendix A shows a possible visualization of the property value and severity of the collected properties.

7.8.1 Detection of bottlenecks

In this section the detection of inefficient use of SuperMUC is presented with a real application. An application run with 2048 cores, i.e. 128 nodes, was examined. It can be seen in Figure 7.20 that the percentiles 60, 70, 80, 90, and 100 can be visually grouped together, while the rest of the percentiles have 0 flops/s. This means that about half of the cores were idle. The intra-node imbalance was also severe (severity was most of the time close to one).

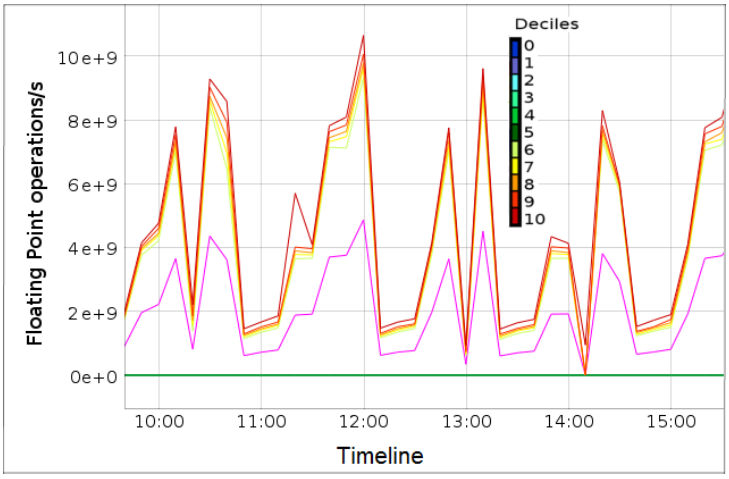


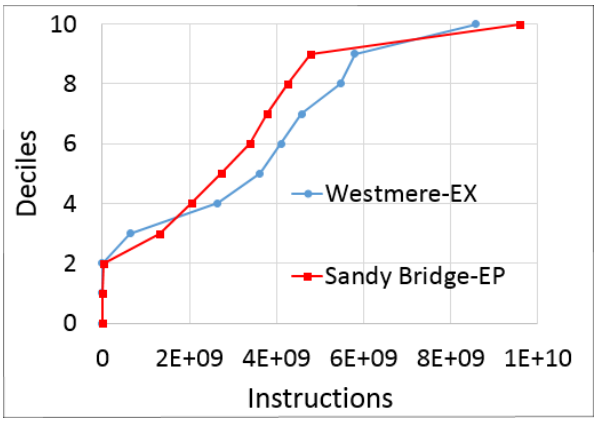
Figure 7.20: Application with bottleneck: Floating point operations/s

Using less than 16 cores per node can be justified if the memory per core is higher than 1.5GB. The tool detected that the memory usage was about 1.7GB per active core, but not as twice as much (3GB/Core), such that the user did not require to use 8 Cores/Node. The user was advised to redistribute the job using 12 cores per node (or more) if possible. The application now runs with about 67% of the resources, with only 1376 cores (i.e. 86 nodes).

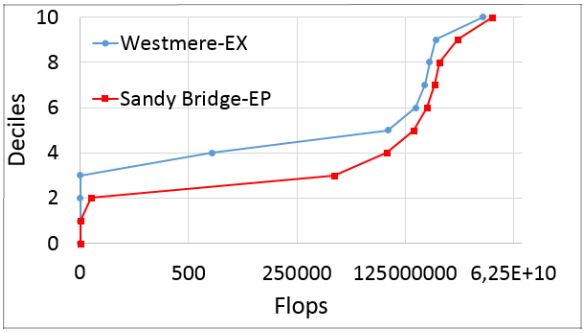
7.8.2 Comprehensive View of System Usage

The collected data can be processed in order to obtain a distribution of a given property in a period of time. The data for floating point operations per second, instructions per second, and memory bandwidth for all the measurements in ten days were collected. All of the values are per core. The deciles were estimated from all the jobs which were measured and the average was precisely calculated (as opposed to estimated) and the data included the measurement of

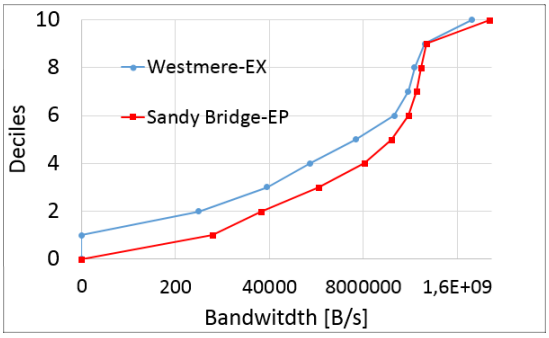
'empty' nodes, nodes which were not running a job. The results are shown in Figures 7.21a through 7.21c. The decile estimation was done using the method described in Chapter 5.



(a) Instructions distribution



(b) Flops distribution



(c) Memory bandwidth usage distribution

Figure 7.21: Comprehensive view of system usage

Exact averages for these three properties are shown in Table 7.13. The averages come from the same set of data used for estimating the deciles.

70% of the Westmere-EX cores run instructions in the order of GB, while 80% of the Sandy Bridge-EP cores count more than 80%, both types of cores have averages in the order of giga instructions. In the case of the flops, however, up to 10% are Giga flops per core, while 20% (or less) of the cores run with Giga flops. The averages are in the order of 100 MFlops for both architectures. Most of the measurements are not showing a significant memory bandwidth usage, only 10% or less are considered memory bound.

	Sandy Bridge-EP	Westmere-EX
Giga Instructions	2.54	3.04
Giga Flops	0.92	0.30
Memory Bandwidth GB/s	0.10	0.07

Table 7.13: Average Values per Core of Usage for the SuperMUC Processor Architectures

8

Conclusions and Outlook

This thesis describes concepts and methods for systemwide monitoring of large scale HPC architectures. These concepts and methods have been implemented in the PerSyst Tool which was extensively evaluated on a petaflop system, SuperMUC. The results show that the concepts presented allow for a scalable, low overhead monitoring which also provides a preliminary analysis of user codes. The optimization cycle identified starts with a preliminary detection of bottlenecks in user codes which are being run in a supercomputer. The tool supports this optimization cycle without requiring user training and without requiring the users to perform additional actions to monitor the performance of their codes. Recommendations given for the bottlenecks found allow to guide the detailed analysis and the optimization.

The main concepts developed are the on-line analysis of application runs adapted to systemwide monitoring without instrumentation. This on-line analysis is achieved with codified expert knowledge in strategy maps which are designed to reveal bottlenecks in an application. Scalability is achieved with a hierarchical distributed software architecture: a tree of agents which can operate autonomously and run continuously to measure, analyze, filter, and collect performance data. The tree is designed to optimize the collection route and minimize the usage of the network interconnect. Several techniques have been applied to reduce the volume of performance data. Firstly, perforamance data are collected or discarded based on the strategy maps. Thus, only the useful data are kept. Secondly, descriptive qualities of performance data are retained by using quantiles which largely reduces the raw data.

The following aspects of the presented concepts were evaluated in detail. The scalability of the agent tree as well as the the optimizations performed on the transport system were assessed, showing the difference between using the entire agent tree or parts of it. The amount of data reduction due to the usage of quantiles and due to selection of jobs was quantified. The interference of the tool with running applications was evaluated. The result was that the tool exerts a negligible interference. The quality of the method used for quantile estimation was verified with a trial population. The validation of the performance measurements and analysis was done and thresholds were determined. Finally, examples of use were provided such as bottleneck detection and a comprehensive view of system performance.

The following sections provide the conclusions and outlook for the different concepts, methods, and algorithms which were applied and their evaluation.

8.1 Conclusions

The PerSyst Tool's agent tree is a highly scalable solution. The tool is scalable in all the objective areas: synchronized measurement, tool management, and accessing of data. The scalability of a synchronous measurement was achieved with the tree hierarchy. The tool management is feasible with a single configuration file, the start of one process (the frontend), and terminating the agent tree can also be achieved only through the frontend. Scalability for extracting the data was achieved by novel approaches to reduce information: storage of properties as opposed to raw counters; selection of performance data that detect inefficiencies; and aggregation using quantiles.

The tool was successfully ported to three systems, including the integration of two of them, having one agent tree deal with two different types of architectures and runs in operation mode in a petaflop system. The framework allows for the parallelization among measurment tools and the measurement within a single measurment tool. This is useful when the number of cores makes the measurement slow down, a parallel measurement in groups of cores is available.
Several tools and interfaces have been adapted to the PerSyst Tools framework including two different batch schedulers. Thus, the framework is extensible and suitable to allow for a wide variation of ad hoc implementations, including implementations for heterogeneous systems.
The deployers of the tools are typically system administrators or performance experts in supercomputing centres that need to collect performance data in large HPC architectures. These users require to implement the specific functionalities related to the system, define properties, and calibrate the thresholds. An estimate of the effort required to set the tool in production mode is of one month work for one person. Even though this appears to require much effort, this is done only once and only one person requires to learn the tool's interface. The advantage to this is that all of the users that send jobs to a supercomputer can benefit from the collected performance data; without requiring any user training.

The tool runs with a negligible interference on the running jobs and is, thus, suitable for permanent systemwide collection of data of large HPC systems. Monitoring without instrumentation greatly reduces the measurement overhead. The analytical codified knowledge without the use of instrumentation is sufficient for a preliminary detection of bottlenecks.

The definition of properties as specified by the APART Specification Language was modified for monitoring without instrumentation. The concept of the property value was introduced, which allows the tool to attain performance data at system level. This in turn, allows to obtain a comprehensive view of system performance and the changes of performance over time at a

system level.
The most outstanding advantage of the use of performance properties is that this leads to a easier and faster interpretation of results than reading the raw hardware counters (no further calculations on the data are needed). The revealing of inefficiencies is readily available by making direct requests to the properties database. On the other hand, severities allow to recognize the application with the worst bottlenecks. The severity relies on the threshold to evaluate a Property. Thus, choosing thresholds precisely will ensure the quality of the analysis. Thresholds were determined with four main heuristics. Firstly, with expert knowledge combined with knowledge on the hardware characteristics. Thresholds can also be based on benchmarks. Thirdly, by using a performance degradation policy. Finally, thresholds can be based on statistical data on measurements done to real applications.

As new technologies emerge and more complex micro-architectures are made available, the performance patterns become more complex. Subsequently, the strategy maps need to be updated or modified. The general guidelines on how to formulate strategy maps, however, have been given: limitations inherent to the code, optimizations that have the biggest impact on performance (i.e., bottlenecks with the highest latencies), and finally identification of the bottlenecks that typically arise more frequently. These guidelines were applied successfully to two architectures.

Measurements are performed synchronously and collection asynchronously. The advantage of a synchronous measurement is that it allows the comparison of measurements among properties and aggregation of measurements in a measurment interval. Synchronous measurements are important for the load imbalance property and a comprehensive systemwide overview. The use of quantiles (which can be seen as a data compression) and asynchronous collection allows the tool to perform an output directly to a database or file system.
Quantiles have proven to be effective in data reduction while still keeping the quality of the data. They reduce the information of large jobs into manageable amounts.
Estimation of quantiles that need to be used in the context of a tree agent hierarchy preserve also the quality of the data.
Due to the use of quantiles, two different kinds of aggregations are needed that produce exact calculations at certain nodes and another type of aggregation that estimates the new set of quantiles. In order to avoid meta-aggregation of quantiles, the transport system adapts to the jobs' topological placement in the supercomputer. Not only are the estimations largely avoided, but the extraction of data is optimized compared to the traditional extraction that uses the entire tree topology. This optimization reduces network traffic by processing of information as locally as possible. The optimized routes also exploit the topology of the tree agent. By having parent and child agents arrangements with the faster interconnects between them, the extraction of performance data will also use the faster interconnects. Using specific parts of the tree rather than its entirety for extracting the data implies that there will be less communication and less network traffic which then translates into less data loss.

8.2 Outlook

Future work includes integrating the tasks of the different agent types. If different agent types are combined into one agent, then the memory footprint will be optimized. A PerSyst Agent could also serve as a Collector for doing collecting tasks if a PerSyst Agent and a Collector Agent are placed at the same node.
The agent tree demonstrated to be scalable. However, the scalability into larger systems, in the order of exascale, will require further optimizations, especially in the communication times. Optimizations in the algorithms presented can be carried out in order to reduce the time complexity.
Extensions to the tool include a dedicated strategy map for every field of science. Given that the strategy to be measured is communicated with the measurement command, the tool could be easily adapted to support dedicated strategies. The idea behind specialized strategies for each application type is that they would be more reliable in revealing bottlenecks by preventing false positives. Moreover, online analysis can be used to apply performance steering in the user applications. The strategy can be changed at each measurement interval to adapt to the running applications and focus on one performance pattern when it is deemed as dominating. For example: the I/O strategies could be more detailed when the tool detects an I/O region, while other strategies are simply switched off.
In order to alleviate the effort of the system-specific implementations, a sample of delegates will be provided in which standard Linux tools will be used with its corresponding properties. These standard delegates will make it possible to deploy the tool by only setting a configuration file. This will make it possible to have standard measurements without job correlation with almost no effort for deployment.
Furthermore, the tool will be ported to future HPC architectures. With the deployment of novel multicore architectures it will require new analysis strategies. In order to make the results available to users, the visualization (see Appendix A) will be made available to the users.

A
Visualization

The visualization tool is presented in this Anex, showing the performance data of user applications. The data from the applications are from real production runs on the petaflop system SuperMUC at the Leibniz Supercomputing Centre. They were taken from the database of the PerSyst Tool. Figure A.1 shows the visualization of the severities of a list of applications (user data appears blurred). The severities are shown via a colour codification. Green, yellow, orange, red, and deep pink correspond to 0, 0.25, 0.5, 0.75, and 1 respectively. Like in a heat map, red colours represent a higher severity. White spaces mean that no properties were captured.

The severities are shown in a timeline for the time the job was running in Figure A.2. Every square in the timeline represents a measurment interval and has the same colour codification as in Figure A.1. The severities along with the timeline are presented for each property. The property names are listed on the left and the strategy hierarchy is shown with indentations. At the left of each property name a square with the average severity with respect to cores and time is shown. Recommendations are shown as a "hint" in the visualization.

Finally, Figure A.3 shows two graphs which allow for comparison of two properties in the timeline. In this case memory bandwidth an expensive instructions are shown. A coloured line represents a decile.

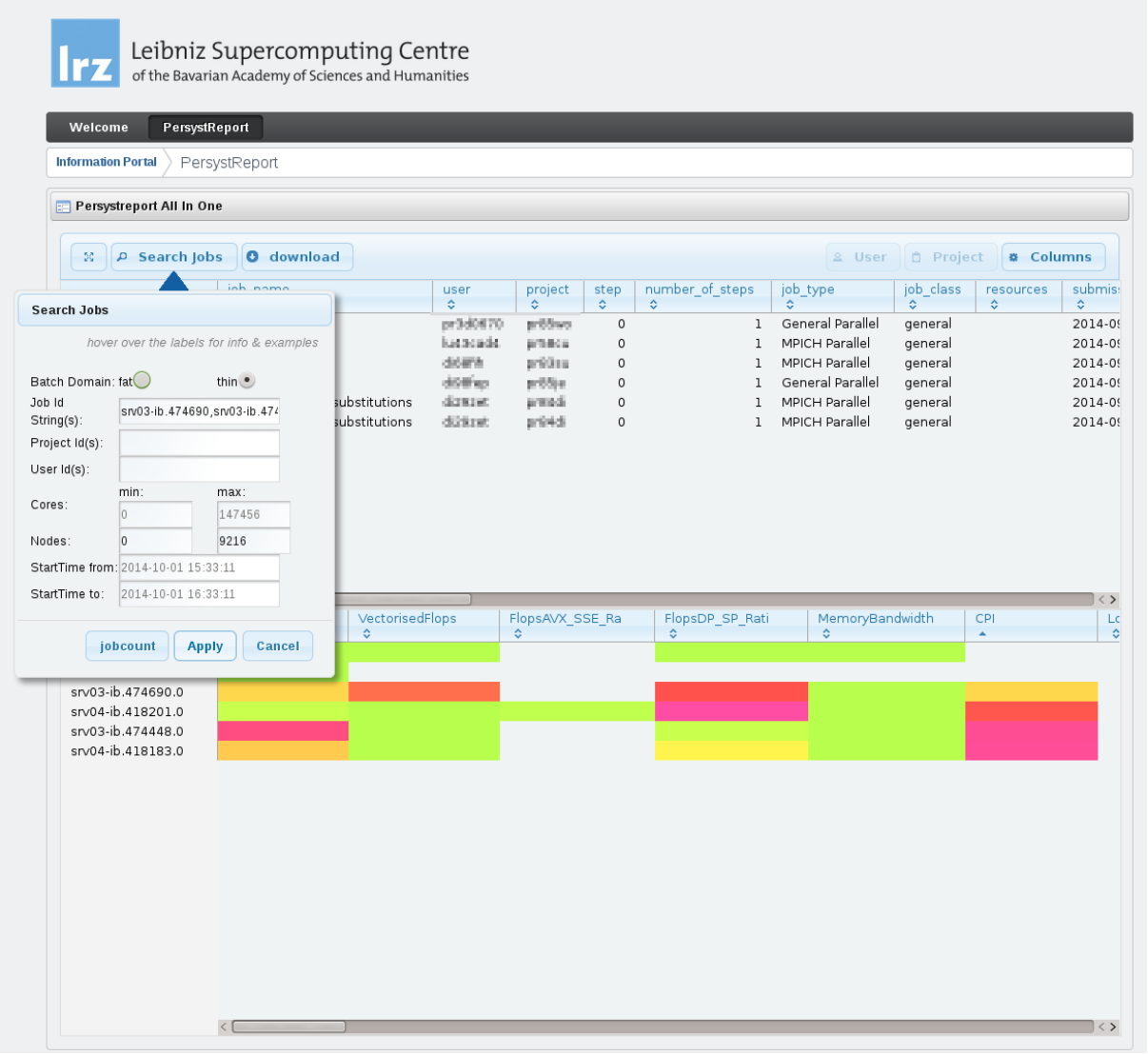


Figure A.1: View of Average Severity for Several Jobs

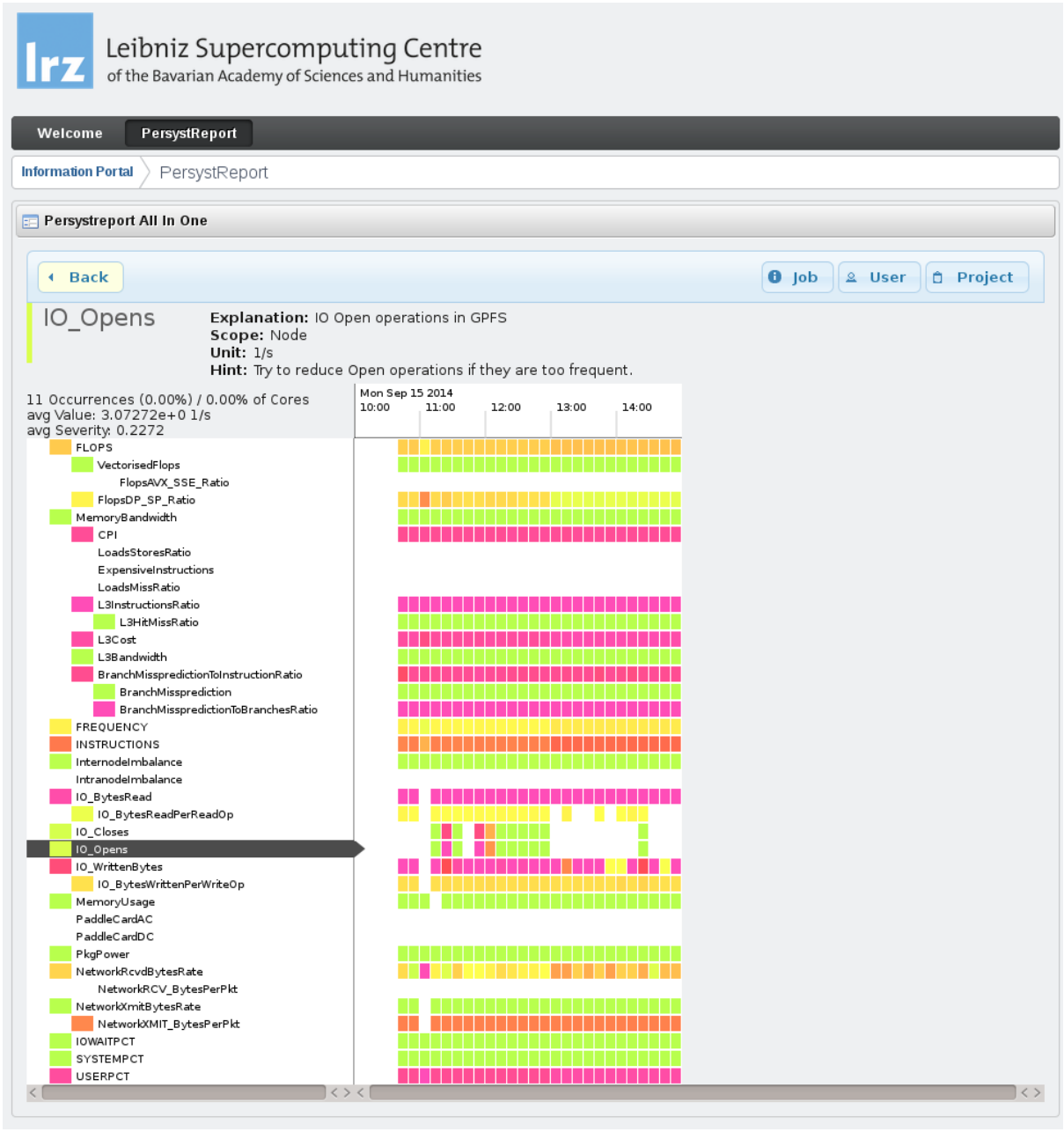


Figure A.2: View of Severity for a Single Job in a Timeline

Figure A.3: View of Property Value in a Timeline

B

Glossary

ACE	Adaptive Communication Environment
APART	Automatic Performance Analyisis: Real Tools
API	Application Programming Interface
ARITH.CYCLES_DIV_BUSY	Event counts instruction resulting from division operations.
ARITH.FPU_DIV_ACTIVE	Event counts instructions due to division.
ASL	APART Specification Language
ATS	APART Test Suite
AVX	Advanced Vector Extensions
BAdW	Bayerische Akademie der Wissenschaft
BR_INST_RETIRED.ALL_BRANCHES	Event measures the total amount of branches.
BR_MISP_RETIRED.ALL_BRANCHES	Event measures the mispredicted branches.
CAS_COUNT.RD	Loaded lines from memory per channel.
CAS_COUNT.WR	Written lines to memory per channel.
CDR	Common Data Representation
CmdReads	FVC_EV0_ BBOX.CMDS_READS
CPI	Clocks per Instructions
CPU	Central Processing Unit
CPU_CLK_UNHALTED.CORE	Event for the unhalted core cycles, i.e. the cycles where the core is active.
CPU_CLK_UNHALTED.REF	Event counts the unhalted reference cycles.
DLB	Dynamic Load Balancing
DP	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION event
DPCL	Dynamic Probe Class Library
DRAM	Dynamic Random Access Memory

FP_256_PACKED_SINGLE	Event counts AVX floating point operations which corresponds to the eight packed flops at single precision.
FP_256_PACKED_DOUBLE	Event counts AVX floating point operations which corresponds to four packed flops at double precision.
FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Event counts double precision flops.
FP_COMP_OPS_EXE.SSE_FP_PACKED	Event counts packed flops.
FP_COMP_OPS_EXE.SSE_FP_SCALAR	Event counts scalar flops.
FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Event counts single precision flops.
FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE	Event counts double precision floating point packed SSE operations.
FP_COMP_OPS_EXE_SSE_FP_PACKED_SINGLE	Event counts single precision floating point SSE operations.
FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE	Event counts the double precision floating point operations.
FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE	Event counts the single floating point operations.
FP_DP	FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE
FP_SP	FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE
FP_DP_{AVX}	FP_256_PACKED_DOUBLE
FP_SP_{AVX}	FP_256_PACKED_SINGLE
FVC_EV0_BBOX.CMDS_READS	Event for the read commands from memory
GCS	Gauss Centre for Supercomputing
GPFS	General Parallel File System
HPC	High Performance Computing
I/O	Input / Output
IA32	Intel 32 bit computer architecture
IA64	Intel 64 bit computer architecture
IMC	Integrated Memory Controller
IMT	In Memory Table
IMT_INSERTS_WR	Event for the write inserts registered at the In Memory Table.
ImtWrite	IMT_INSERTS_WR
INST_RETIRED.ANY	Retired instructions event
IPM	Integrated Performance Monitoring Tool
JobId	Job identification number
KONWIHR	Kompetenznetzwerk für Wissenschaftliches

	Höchstleistungsrechnen
L2_all	L2_LINES_IN.ANY
L2_dirty	L2_LINES_OUT.DEMAND_DIRTY
L2_LINES_IN.ANY	event counts all the lines loaded into the L2 cache.
L2_LINES_OUT.DEMAND_DIRTY	
	Event counts all the lines which have been evicted by demand.
L3	Level three cache
L3_Hits	MEM_LOAD_RETIRED.L3_UNSHARED_HIT
L3Hits	MEMLOAD_UOPS_RETIRED.LLC_HIT
L3_LAT_CACHE.MISS	L3 cache misses
L3_Misses	MEM_LOAD_RETIRED.L3_MISS
LLC	Last level cache
MEM_INST_RETIRED.LOADS	
	Event that counts loads from memory.
MEM_INST_RETIRED.STORES	
	Event counts stores to memory.
MEM_LOAD_RETIRED.L3_MISS	
	event counts the L3 cache misses.
MEM_LOAD_RETIRED.L3_UNSHARED_HIT	
	Event counts the L3 level cache hits.
MEM_UOP_RETIRED.LOADS	
	Event counts load instructions from memory.
MEM_UOP_RETIRED.STORES	
	Event counts store instructions to memory.
MemFree	Free memory in a node
MemLoad	MEM_INST_RETIRED.LOADS
MemStore	MEM_INST_RETIRED.STORES
MemTotal	Total available memory in a node
MPI	Message Passing Interface
MRNet	Multicast Reduction Network tool
MSR	Model-specific register
NUMA	Non-uniform Memory Access
OpenMP	Open Multiprocessing
Packed_DP	FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE
Packed_SP	FP_COMP_OPS_EXE_SSE_FP_PACKED_SINGLE
PAPI	Performance Application Programming Interface
POWER_PKG.WATT	RAPL counter that measures package power.
POWER_DRAM.WATT	RAPL counter that measures DRAM power.
PRACE	Partnership for Advanced Supercomputing in Europe
QPI	Quick Path Interconnect
QPI_RATE_STATUS	Rate at which QPI is transferring data.

RAPL	Running Average Power Limit
SAR	System Activity Report
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multithreading
SP	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION
SSE	Streaming SIMD Extension
SyncAgent	Synchronization Agent
TAU	Tuning and Analysis Utilities
TC	Total Cycle Time
TDP	Thermal Design Power
UOPS_EXECUTED.PORT015_STALL_CYCLES	
	Event count for the micro operations which have stalled
UOPS_RETIRED.ANY	Event counts for all the micro operations
UT	Unix Timestamp
XML	Extensible Markup Language

Bibliography

[1] *Introduction to Microarchitectural Optimization for Itanium 2 Processors. Reference Manual. Intel Corporation. 2002.*, 2002.

[2] *Dual-Core Update to the Intel Itanium 2 Processor. Reference Manual. For Software Development and Optimization. Revision 0.9. January 2006.*, 2006.

[3] pfmon tool. `www.hpl.hp.com/research/linux/perfmon/pfmon.php4`, December 2009.

[4] *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2. April 2011.*, 2011.

[5] All about monads. `http://monads.haskell.cz/html/writermonad.html`, August 2012.

[6] Counting floating point operations on intel sandy bridge and ivy bridge. `http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops`, November 2013.

[7] Ganglia monitoring system. `http://ganglia.info/`, 2012 January.

[8] Paulo Sérgio Almeida, Carlos Baquero, Martín Farach-Colton, Paulo Jesus, and Miguel A. Mosteiro. Fault-tolerant aggregation: Flow-updating meets mass-distribution. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 513–527, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Barton P. Miller Andrew R. Bernat. Anywhere, any-time binary instrumentation. *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2011.

[10] Axel Auweter, Arndt Bode, Matthias Brehm, Luigi Brochard, Nicolay Hammer, Herbert Huber, Raj Panda, Francois Thomas, and Torsten Wilde. A case study of energy aware scheduling on supermuc. In JulianMartin Kunkel, Thomas Ludwig, and HansWerner Meuer, editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 394–409. Springer International Publishing, 2014.

[11] Shajulin Benedict, Matthias Brehm, Michael Gerndt, Carla Guillen, Wolfram Hesse, and Ventsislav Petkov. Automatic performance analysis of large scale simulations. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 199–207, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] Erik Berg and Erik Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004)*, Austin, Texas, USA, March 2004.

[13] Andrew R. Bernat, Kevin Roundy, and Barton P. Miller. Efficient, sensitivity resistant binary instrumentation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 89–99, New York, NY, USA, 2011. ACM.

[14] Matthias Brehm. Personal communication, October 2011.

[15] Matthias Brehm. Personal communication, April 2012.

[16] Matthias Brehm, Sascha Haupt, and Richard Patra. Performance monitoring - a generic approach. Technical report, Leibniz-Rechenzentrum, 2006.

[17] Michael J. Brim, Luiz DeRose, Barton P. Miller, Ramya Olichandran, and Philip C. Roth. Mrnet: A scalable infrastructure for development of parallel tools and applications. *Cray User Group 2010 Proceedings*, 2010.

[18] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*. ACM, November 2013.

[19] John Mc Calpin. Notes on the mystery of hardware cache performance counters. `http://blogs.utexas.edu/jdm4372/2013/07/14/notes-on-the-mystery-of-hardware-cache-performance-counters/`, July 2013.

[20] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Robert Latham, Samuel Lang, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. In *27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011)*, 2011.

[21] ParTec Cluster Competence Center. Gridmonitor administrator's guide, September 2007.

[22] ParTec Cluster Competence Center. Parastation gridmonitor. software product detailed description., May 2011.

[23] IBM Corporation. The cpufreq governors. `http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaai/cpufreq%2FTheCPUFreqGovernors.htm`, February 2014.

[24] Intel Corporation. Intel xeon processor e5-2680. `http://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-(20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI)`, December 2013.

[25] Intel Corporation. Intel xeon processor e7-4870. `http://ark.intel.com/products/53579/Intel-Xeon-Processor-E7-4870-%2830M-Cache-2_40-GHz-6_40-GTs-Intel-QPI%29`, June 2013. Westmere EX specifications.

[26] Arnaldo de Melo. The new linux perf tools. In *17 International Linux System Technology Conference (Linux Kongress)*, September 2010.

[27] Jeff Dean. Latency numbers every programmer should know. `http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`, November 2012. Latency of computing components + history + perspective.

[28] Valgrind Developers. Valgrind. `http://valgrind.org/`, December 2012.

[29] Jack Dongarra, Allen D. Malony, Shirley Moore, Philip Mucci, and Sameer Shende. Performance instrumentation and measurement for terascale systems. In *International Conference on Computational Science*, pages 53–62, 2003.

[30] Garrett Drysdale, Antonio C. Valles, and Matt Gillespie. Performance insights to intel hyper-threading technology. `http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/`, November 2009.

[31] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *USENIX Winter*, pages 303–314, 1995.

[32] Thomas Fahringer, Michael Gerndt, Bernd Mohr, Felix Wolf, Graham Riley, and Jesper Larsson Trff. Knowledge specification for automatic performance analysis, August 2001.

[33] The Linux Foundation. System activity report. `http://www.softpanorama.org/Admin/Monitoring/sar.shtml`, January 2013.

[34] Ildiko E. Frank and Roberto Todeschini. *The data analysis handbook*, volume 14. Elsevier Science B.V., 1994.

[35] Karl Fürlinger and David Skinner. Capturing and visualizing event flow graphs of mpi applications. In *Workshop on Productivity and Performance (PROPER 2009) in conjunction with Euro-Par 2009*, August 2009.

[36] Karl Fürlinger, Nicholas J. Wright, and David Skinner. Performance analysis and workload characterization with ipm. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Parallel Tools Workshop*, pages 31–38. Springer, 2009.

[37] Michael Gerndt and Karl Fürlinger. Specification and detection of performance problems with asl: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(11):1451–1464, August 2007.

[38] Michael Gerndt and Karl Fürlinger. Automatic performance analysis with periscope. *Journal: Concurrency and Computation: Practice and Experience.Wiley InterScience. John Wiley & Sons, Ltd.*, 2009.

[39] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced techniques for performance analysis, parallel computing: Current & future issues of high-end computing. In *International Conference ParCo 2005*, volume 33, 2006. NIC Series ISBN 3-00-017352-8.

[40] Michael Gerndt and Edmond Kereku. Search strategies for automatic performance analysis tools. In *Euro-Par 2007*, volume LNCS 4641, pages 129–138, 2007.

[41] Michael Gerndt and Sebastian Strohhäcker. Distribution of analysis agents in periscope on altix 4700. In *Proceedings of ParCo*, 2007.

[42] Carla Guillen, Wolfram Hesse, and Matthias Brehm. A scalable monitoring tool using performance properties. *inSiDE - Innovatives Supercomputing in Deutschland*, 9, 2011.

[43] Carla Guillen, Wolfram Hesse, and Matthias Brehm. A new scalable monitoring tool using performance properties of hpc systems. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 51–60. Springer Berlin Heidelberg, 2012. 10.1007/978-3-642-24025-6.5.

[44] Carla Guillen, Wolfram Hesse, and Matthias Brehm. The persyst monitoring tool - A transport system for performance data using quantiles. In *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, volume 8806 of *Lecture Notes in Computer Science*, pages 363–374. Springer, 2014.

[45] Carla Guillen, Wolfram Hesse, Carmen Navarrete, Matthias Brehm, and Jan Treibig. A flexible framework for energy and performance analysis of highly parallel applications in a supercomputing centre. In *inSiDE Autumn 2013*, volume 11, 2013.

[46] Juergen Haas. Linux / unix command: top. `http://linux.about.com/od/commands/l/blcmdl1_top.htm`, January 2013.

[47] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 194–204, 2013.

[48] Georg Hager. Benchmark for measuring predictor performance.

[49] Claudia Hemmelmann. Personal communication, September 2012. Conversation with Dr. rer. hum. biol. Claudia Hemmelmann from the University of Luebeck, Institut for medical biometry and statistics.

[50] Jenifer Hopper. Reduce linux power consumption, part 1: The cpufreq subsystem. `http://www.ibm.com/developerworks/library/l-cpufreq-1/`, September 2009.

[51] IBM Corporation. *GPFS: Administration and Programming Reference.*, `http://publibz.boulder.ibm.com/epubs/pdf/a2314520.pdf` version 4 release 1 edition, 2014.

[52] Intel. *Intel Itanium 2 Processor Reference Manuel. For Software Development and Optimization. May 2004.*

[53] Intel. *Intel Xeon Processor E7 Family Uncore Performance Monitoring Programming Guide*, April 2011.

[54] Intel. *Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide.* Intel, March 2012.

[55] Emily R. Jacobson, Michael J. Brim, and Barton P. Miller. A lightweight library for building scalable tools. In Kristjn Jnasson, editor, *PARA (2)*, volume 7134 of *Lecture Notes in Computer Science*, pages 419–429. Springer, 2010.

[56] P. Jain and D. Schmidt. Dynamically Configuring Communication Services with the Service Configurator Pattern, 1997.

[57] Sverre Jarrp, Alfio Lazzaro, Julien Leduc, and Andrzej Nowak. Evaluation of the intel westmere-ex server processor. Technical report, CERN Openlab, July 2011.

[58] Sverre Jarrp, Alfio Lazzaro, Julien Leduc, and Andrzej Nowak. Evaluation of the intel 4 socket sandy bridge-ep server processor. Technical report, CERN Openlab, December 2012.

[59] Paulo Jesus, Carlos Baquero, and PauloSrgio Almeida. Fault-tolerant aggregation by flow updating. In Twittie Senivongse and Rui Oliveira, editors, *Distributed Applications and Interoperable Systems*, volume 5523 of *Lecture Notes in Computer Science*, pages 73–86. Springer Berlin Heidelberg, 2009.

[60] Rajiv Kapoor. Avoiding the cost of branch misprediction. `http://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction`, February 2009.

[61] Edmond Kereku and Michael Gerndt. The monitoring request interface (mri). *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, page 238, 2006.

[62] Andreas Knpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, MatthiasS. Mller, and WolfgangE. Nagel. The vampir performance analysis tool-set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[63] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Low overhead program monitoring and profiling. In Michael D. Ernst and Thomas P. Jensen, editors, *PASTE*, pages 28–34. ACM, 2005.

[64] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. Dip: A parallel program development environment. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 665–674, London, UK, UK, 1996. Springer-Verlag.

[65] Chee Wai Lee, Allen D. Malony, and Alan Morris. Taumon: scalable online performance data analysis in tau. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 493–499, Berlin, Heidelberg, 2011. Springer-Verlag.

[66] Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube, and Zoltán Szebenyi. Extending Scalascas analysis features. In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 115–126. Springer Berlin Heidelberg, 2013.

[67] Allen D. Malony and Daniel A. Reed. Models for performance perturbation analysis. *SIGPLAN Not.*, 26(12):15–25, December 1991.

[68] Allen D. Malony, Sameer S. Shende, Alan Morris, and Felix Wolf. Compensation of measurement overhead in parallel performance profiling. *International Journal of High Performance Computing Applications*, 21(2):174–194, May 2007. Special Issue *Selected Papers from the EuroPVM/MPI 2005 Conference*.

[69] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[70] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[71] William Mendenhall and Terry Sincich. *Statistics for engineering and the sciences*. Prentice-Hall International, Inc., 4th edition, 1995. ISBN 0-13-181017-0.

[72] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[73] Renato Miceli, Gilles Civario, Carla Guillen, Carmen Navarrete, and Matthias Brehm. Analysis on i/o bottlenecks for further implementation phases. Technical report, AutoTune (Automatic Online Tuning Consortium), 2014.

[74] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In Pekka Manninen and Per Öster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2013.

[75] Ronald G. Minnich. Supermon: high-performance monitoring for linux clusters. In *Proceedings of the 5th annual Linux Showcase & Conference - Volume 5*, pages 5–5, Berkeley, CA, USA, 2001. USENIX Association.

[76] Bernd Mohr, Vladimir Voevodin, Judit Gimenez, Erik Hagersten, Andreas Knuepfer, DmitryA. Nikitenko, Mats Nilsson, Harald Servat, Aamer Shah, Frank Winkler, Felix Wolf, and Ilya Zhukov. The hopsa workflow and tools. In Alexey Cheptsov, Steffen Brinkmann, Jos Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 127–146. Springer Berlin Heidelberg, 2013.

[77] Ryan Mooney, Kenneth P. Schmidt, R. Scott Studham, and Jarek Nieplocha. Nwperf: a system wide performance monitoring tool for large linux clusters. *Cluster Computing, IEEE International Conference on*, 0:379–389, 2004.

[78] P. J. Mucci. Dynaprof tool. http://web.eecs.utk.edu/~mucci/dynaprof/ and http://www.linuxclustersinstitute.org/conferences/archive/2000/PDF/Mucci.pdf, 2012 August.

[79] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[80] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian C. Arnold, and Barton P. Miller. A framework for scalable, parallel performance monitoring using tau and mrnet. *Concurr. Comput. : Pract. Exper.*, 22(6):720–735, April 2010.

[81] Aroon Nataraj, Matthew Sottile, Alan Morris, Allen Malony, and Sameer Shende. Tauoversupermon: Low-overhead online parallel performance monitoring. In *Proceedings Euro-Par 2007*, volume LNCS 4641, pages 85–96, 2007.

[82] Carmen Navarrete, Carla Guillen, Wolfram Hesse, and Matthias Brehm. Optimizing the energy-to-solution on sandybridge systems. In *inSiDE Autumn 2012*, volume 10, 2012.

[83] Carmen Navarrete, Carla Guillen, Wolfram Hesse, and Matthias Brehm. Autotuning the energy consumption. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE). Advances in Parallel Computing 25*. IOS Press, 2014.

[84] Tobias Oetiker. Round robin database tool. http://oss.oetiker.ch/rrdtool/, May 2012.

[85] Top 500 Organization. Development over time of application areas. http://www.top500.org/statistics/overtime/, March 2013.

[86] Top 500 Organization. Top500. supercomputer sites. http://www.top500.org/, August 2014.

[87] Venkatesh Pallipadi. cpuidle - do nothing, efficiently... 2007.

[88] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, 5th edtion edition, 2014.

[89] Xiao Qin, Hong Jiang, Adam Manzanares, Xiaojun Ruan, and Shu Yin. Dynamic load balancing for i/o-intensive applications on clusters. *Trans. Storage*, 5(3):9:1–9:38, November 2009.

[90] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[91] Luiz De Rose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The dynamic probe class library: An infrastucture for developing instrumentation for performance tools. In *IPDPS*, page 66. IEEE Computer Society, 2001.

[92] Hal Rosenstock. perfquery. http://linux.die.net/man/8/perfquery, October 2012.

[93] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *in: Proc. IEEE/ACM Supercomputing 03*, 2003.

[94] 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, pages 460–469, New York, NY, USA, 2009. ACM.

[95] Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler. Wide area cluster monitoring with ganglia. In *CLUSTER*, pages 289–. IEEE Computer Society, 2003.

[96] Douglas Schmidt and Stephen Huston. *C++ Network Programming Vol. 1: Mastering Complexity with ACE and Patterns*. Pearson Education, 2002.

[97] Douglas C. Schmidt. The reactor: An object-oriented wrapper for event-driven port monitoring and service demultiplexing (part 1 of 2). *I/O Multiplexing, C++ Report*, 1993.

[98] Douglas C. Schmidt. An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit. 1995.

[99] Douglas C Schmidt. The adaptive communication environment (ace). `http://www.cs.wustl.edu/~schmidt/ACE.html`, January 2013.

[100] Douglas C. Schmidt and Irfan Pyarali. The design and use of the ace reactor - an object-oriented framework for event demultiplexing.

[101] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Memory performance at reduced cpu clock speeds: An analysis of current x86_64 processors. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[102] Robert Schöne, Jan Treibig, Manuel Dolz, Carla Guillen, Carmen Navarrete, Michael Knobloch, and Barry Rountree. Tools and methods for measuring and tuning the energy efficiency of hpc systems. *AutoTune Special Edition*, 2014.

[103] Harald Servat, Germán Llort, Judit Giménez, and Jesús Labarta. Detailed performance analysis using coarse grain sampling. In *Proceedings of the 2009 International Conference on Parallel Processing*, Euro-Par'09, pages 185–198, Berlin, Heidelberg, 2010. Springer-Verlag.

[104] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press.

[105] Sameer Shende, AllenD. Malony, and Alan Morris. Workload characterization using the tau performance system. In Bo Kgstrm, Erik Elmroth, Jack Dongarra, and Jerzy Waniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 289–296. Springer Berlin Heidelberg, 2007.

[106] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.

[107] David Skinner. Performance monitoring of parallel scientific applications. *LBNL-5503*, 2005.

[108] Matthew J. Sottile and Ronald G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '02, pages 39–46, Washington, DC, USA, 2002. IEEE Computer Society.

[109] Jürgen Teich, Wolfgang Schröder-Preikschat, and Andreas Herkersdorf. Invasive computing - common terms and granularity of invasion. *CoRR*, abs/1304.6067, 2013.

[110] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[111] Jan Treibig. Intel sandy bridge and counting the flops. http://likwid-tools.blogspot.de/2012/02/intel-sandybridge-and-counting-flops.html, February 2012.

[112] Jan Treibig. Likwid. like i knew what i was doing. http://code.google.com/p/likwid/, February 2012.

[113] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. *CoRR*, abs/1004.4431, 2010.

[114] Jan Treibig, Georg Hager, and Gerhard Wellein. Best practices for hpm-assisted performance engineering on modern multicore processors. *CoRR*, abs/1206.3738, 2012.

[115] Tsirigotis. *xinetd Manual pages*, 1992.

[116] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Workshop on Binary Instrumentation and Application.*, 2007.

[117] Stefan Wellek and Maria Blettner. Establishing equivalence or non-inferiority in clinical trials—part 20 of a series on evaluation of scientific publications. *Dtsch Arztebl Int*, 109(41):674–679, 2012.

[118] PBS Works. Pbs batch scheduler. http://www.pbsworks.com/, July 2014.

[119] Brian J. N. Wylie, Bernd Mohr, and Felix Wolf. Holistic hardware counter performance analysis of parallel programs. In *Parallel Computing : Current & Future Issues of High-End Computing ; proceedings of the International Conference ParCo 2005 / ed.: G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, E. Zapata. - Jlich, FZJ, John von Neumann Institute for Computing, 2006. - (NIC series ; 33). - 3-00-017352-8. - S. 187 - 194*, 2006. Record converted from VDB: 12.11.2012.