TECHNISCHE UNIVERSITÄT WIEN

D I S S E R T A T I O N

Systematic Performance Analysis and Interpretation for Parallel and Distributed Programs with Aksum

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o. Univ.-Prof. Dipl. Ing. Dr. Thomas Fahringer

Institut für Informatik, Leopold-Franzens Universität Innsbruck

eingereicht an der Technischen Universität Wien

Fakultät für Informatik

von

Clóvis Seragiotto Júnior

Matrikelnummer: 0327222

Wien, am 12. Oktober 2005

Abstract

With applications growing more complex everyday, it becomes also more difficult to understand the interaction between their components and the factors that are responsible for loss of performance. An unmanageable number of variables may affect and degrade the efficiency of an application, making it necessary to resort to performance analysis tools in order to achieve even an acceptable level of performance.

Performance analysis tools rely on instrumentation and monitoring tools to perform measurements and collect data. Since two instrumentation tools seldom use the same protocol to communicate with other tools or to represent the performance data collected, integrating a performance analysis tool and an instrumentation/monitoring tool may require a reasonable amount of time, implying dependence on a specific tool and, in some cases, on a language or environment. In this dissertation, we propose a standard format for both representation of performance data and communication with instrumentation and monitoring tools. These formats are generic enough to allow their use with several programming languages and paradigms, covering not only the requirements of the current generation of performance tools, but also the capabilities of instrumentation and monitoring tools available today, besides being platform neutral.

We also developed a fast and powerful instrumentation and monitoring engine for Java that makes use of the formats we proposed. Our tool, called Twilight, is based on state-of-the-art technology for instrumenting and monitoring Java programs, allowing both source code and bytecode instrumentation. Twilight has a rich set of metrics; it is written entirely in Java and provides high-level support for dynamic bytecode instrumentation, presenting a view similar to the original source code that hides details about the structure of the bytecodes instrumented. In addition, Twilight has full support for distributed Java applications, providing useful performance metrics specific for distributed programs.

Nevertheless, an instrumentation and monitoring tool provides only the means to carry out the performance analysis of an application. It can easily generate an amount of data that cannot be fully analyzed even by the most skillful performance analyst, a situation that becomes far worse when one needs to compare the performance of several executions. For this reason, we also created a sophisticated and highly customizable performance analysis tool called Aksum, which allows one to generate several experiments for different input parameters, decides automatically which pieces of an application must be instrumented, and outputs a condensed yet significant analysis of the application's performance. By using JavaPSL, a Java-based language we developed for specification of performance problems, Aksum allows one to incorporate the definition of new problems that were not envisaged when Aksum was developed. Aksum conducts the performance analysis in a systematic way using an overhead classification system, and also interprets the performance data gathered by the instrumentation tool; Aksum's output is not simply a collection of charts or absolute numbers, but normalized values that allow the easy identification of problems that require immediate attention.

This dissertation also shows how the problem of performance analysis can be formalized using reinforcement learning techniques; such formalization, which we integrated into Aksum, can be used to justify many decisions taken by a tool for automatic performance analysis.

Finally, we conducted several experiments using real-world applications, which validated the ideas described in this work.

Acknowledgments

I express my gratitude to my advisor, Prof. Thomas Fahringer, who guided me until this dissertation reached its final shape.

I thank the members of the Aurora research project and the staff of the Institute of Scientific Computing for the help they provided over the last years.

Thanks are also due to the members of the APART research group, who helped me especially during the design of instrumentation interfaces.

Contents

1 Introduction

Technology advances constantly try to improve the performance of applications. However, complex interactions between hardware and software (operating system, libraries, and user application) commonly impose performance penalties that are difficult to detect and to analyze. A lack of in-depth knowledge about the technologies used and the abstractions they offer leads to underutilization of the hardware and software resources of modern computers. This is especially true for parallel architectures.

Historically, scientists have been responsible for testing innovative technology for parallel computers. Scientific computing is characterized by an almost exclusive focus on performance, which motivated the creation of new programming models in order to make it possible to obtain the most performance with the minimum effort in new computer architectures. Some of these models have offered powerful abstractions, hiding almost completely the underlying architecture from the programmer and leaving most of the optimization work to compilers. HPF [59], for example, achieved partial success, but even with support from industry and research, it has never become predominant, mainly because it is too constrained in its data distribution features and difficult to be supported by compilers and runtime systems, resulting in programs that suffer from performance problems out of user's control.

Although languages that emphasize productivity without sacrificing performance are still focus of research (like Fortress [2] and Chapel [15]), interfaces like MPI [50] became *de facto* standards, even though they represent "the assembler level of parallel programming for networks of computers" [93, 15]. Programming so close to the machine level can potentially result in very fast applications, but also requires a deep understanding of the computer architectures, how machines are connected, and how software and hardware interact. This led to generations of biologists, physicists, chemists, and many other researchers that have to spend a considerable amount of their time to study and program complex computer architectures; consequently, tools that help developers in analyzing the performance of their applications have grown in importance.

For many researchers, however, especially for those who only occasionally require parallel computing facilities, the cost of parallel computers is prohibitive. On the other hand, while users demand more computation power, studies have reported that computers commonly have long idle times [1, 55]. In this context, distributed computing represents a cost-effective alternative to dedicated parallel computers, delivering great performance by using CPU cycles of idle machines, in extreme cases even more than the most powerful parallel computers (SETI@home reports an average of 60 Teraflops [32]). This fact, allied to the inherent distribution of certain kinds of applications, the potential for incremental growth, and the possibility of sharing data and expensive peripherals, contributed to the popularity of distributed systems not only in research, but also in the industry.

Despite its young age, Java has gained widespread acceptance as a development language for distributed systems. Besides being secure, robust, portable, and architecture neutral, Java has a built-in support for threads and synchronization, achieving in some cases performance similar to C or C++ [83, 87]. Unfortunately, many abstractions provided in Java are misused or misunderstood and, once again, tools for the performance analysis of applications that use these abstractions became extremely important, since the hidden complexity may create complicated interactions that impose severe performance penalties.

The application's user (who sometimes is also the application developer) is the person who ultimately decides if the performance of the application is adequate or not. In order to be useful and profitable, each application, whether an earthquake simulator or a distributed chess game, has some performance requirements that must be achieved. The classical approach for performance analysis consists of several cycles of running an application with a monitoring tool, analyzing the data the tool generates, and modifying the application based on the analysis. This is a quite mechanical, time-consuming, and tedious process; automating it, even partially, can enormously accelerate the performance analysis. Nevertheless, the possibilities of automation in several steps of the analysis have been only partially or not explored. For instance, traditional performance analysis tools are unable to combine and relate performance data generated from multiple application executions with different input parameters, and visualization tools do not scale well for long-running applications or if there is large number of processors involved [92, 108], overwhelming the user with data and charts and leaving him or her their interpretation. Not unimportant is also the ability of a tool to adapt to new architectures or programming paradigms, and how easy it is to incorporate knowledge into the tool.

The main goal of this work is to automate the analysis step, providing the user with condensed yet useful information on which to ground changes in the application code or execution environment that lead to improved performance. Parallel to the development of this goal, we want to make the development of extensible performance analysis tools in general easier, by proposing ideas and representations that promote the integration between tools.

1.1. Motivation

Among the many still open problems in the field of automatic performance analysis, we identified and selected some that we consider important, and directed our research aiming to overcome them. Although we always focused on parallel and distributed applications, many of the topics discussed here are also an issue in the analysis of sequential applications.

1. Complex integration of instrumentation/monitoring tools with performance analysis tools

Performance analysis involves reasoning about the structure of a program, collecting performance data during the program's execution (or executions), and analyzing the data collected. Frequently there is already an instrumentation/monitoring tool that can create a representation of the program structure, insert and remove code (probes) to monitor the program, and monitor the program execution, generating the data needed. Nevertheless, it is also often the case that such a tool uses its own formats to represent data or, even worse, that the representation, as well as the interfaces the tool offers to communicate with other tools, depend on a specific environment or language.

Integration with an instrumentation/monitoring tool is a time-consuming activity, and deciding for one tool can mean a life commitment, potentially limiting or hindering further developments in the future, for instance, if the instrumentation/monitoring tool is discontinued, if it is not updated often, or if it is not ported to some platform. Ideally, it should be possible to add or replace an instrumentation/monitoring tool with few or no changes in the rest of the system.

A performance analysis tool can therefore benefit from:

- a standard representation for the program structure, not so detailed as an abstract syntax tree generated by a compiler front-end but still with enough information to allow sophisticated reasoning;
- a standard representation for the performance data generated during the monitored execution of the program; and
- a standard set of requests and responses to be used in the communication between performance analysis tool and instrumentation/monitoring tool.

Moreover, as we do not want to bind tools to any specific language or platform, the format used in the representations, requests, and responses needs to be machine and language neutral.

2. Knowledge about performance problems is difficult to integrate into performance analysis tools

Although there are several well-known formats to describe tracing and profiling information such as SDDF (Pablo Self-Describing Data Format) [1], ALOG [58] or Vampir Tracefile Format [98], there does not exist a generic way to describe performance problems, which is important to build comprehensive performance analysis tools that can be more easily adjusted for new programming languages and computer architectures.

Most existing performance tools are limited to temporal or spatial overheads (e.g. execution and communication times, cache misses) but do not provide higher-level performance information such as performance properties (e.g. scaling behavior, load imbalance). Tools frequently hard-code performance information, which is awkward to be reused by other tools or to be extended for novel programming paradigms and target machines. Additionally, performance information that is not normalized is difficult to interpret and compare. For instance, absolute number of cache misses for a specific code region has very little meaning without being compared against cache-hit counts for the same region.

A generic specification language for experiment-related data (e.g. information about program versions, code regions, target machine, and profiling or tracing information) and performance properties can be used as a standard performance information interface to describe wide classes of performance problems for a large variety of programming languages and target architectures. Such a language can be used to build higher-level performance analysis technology that accesses experiment-related data or performance properties in order to compute, for instance, new performance properties. Finally, a generic specification language can also be used by other tools such as compilers or transformation tools to access performance information in a portable way.

3. Instrumentation and monitoring tools for Java are not portable or are based on obsolete interfaces

Java is a relatively new language, but it has evolved with impressive speed compared to other languages like C or Fortran. In the last ten years, Java has been constantly corrected and improved; new features were introduced while superseded methods, classes, and even complete APIs were deprecated and removed.

The profiler interface of Java was no exception. Introduced with Java version 1.1 in 1997, it was enhanced several times until, for reasons discussed in Chapter 2, it was deprecated in September 29, 2004, when J2SE 5.0 was officially released. The new API introduced for monitoring and profiling is faster, more robust and portable, attending the needs of modern applications. The old API, however, is still used in several instrumentation, monitoring, and performance analysis tools. In fact, we do not know of any tool that makes use of the new profiler API.

Profiler tools based on the old profiler API are subject to portability problems, as it requires the use of C or C++. Moreover, many counters and timers (like the number of times a thread entered a critical section, or the CPU time of an arbitrary thread) are not available and must be programmatically determined. Finally, if one is interested in some event in some specific piece of code, it is necessary to activate this event for the entire application (which introduces a reasonable overhead) and filter the events that refer to the region of interest. For example, a tool interested in entries and exits only for a method *M* in class *C* would be notified whenever any method in any class is invoked or finishes.

The new profiler API can still be accessed using C and C++, but a large subset of its functionality is available also through the Java APIs. Several counters and timers were introduced, as well as the possibility of dynamically instrumenting class files, which is useful if the source code is not available. In addition, with dynamic instrumentation one can control where the code must be monitored and for how much time, avoiding the “all or nothing” approach which was the only alternative prior to J2SE 5.0. An instrumentation tool built using the new profiler API is therefore portable, simpler, and faster.

4. Performance analysis tools cannot compare several executions and are difficult to extend and customize

There is no single technique that can be successfully applied to the performance analysis of all sorts of programming models and languages. Unfortunately, performance analysis tools usually allow only one technique to be applied; for example, they provide either post-mortem or online analysis, or an analysis based on either trace files or profiles. It is often also the case that the algorithm a tool uses to search for performance problems is so deeply “hardcoded” that the tool cannot be easily extended (or it cannot be extended at all) to incorporate advances in the field of performance analysis. Tools may also not allow the user to input knowledge about a specific application to be analyzed, like code regions that are known to be free of performance problems or that are likely to contain a performance problem.

Expert and non-expert users can therefore benefit from a customizable tool for performance analysis into which knowledge can be plugged. Furthermore, as some performance problems may require several executions of an application to be identified (for instance, with different problem sizes, libraries, and execution parameters), a tool for performance analysis must allow the easy generation of several experiments for different input parameter values. This would also allow studying the influence of different input parameters in the application's performance.

1.2. Contributions of This Thesis

We developed solutions for the problems pointed out above which were recognized in the performance analysis community. Indeed, this thesis is based on articles published since 2001 describing our contributions [38, 39, 126, 127, 128, 129, 130, 131].

1. Easier integration between instrumentation/monitoring tools and performance analysis tools

We created a rich XML-based representation of programs written in Fortran, Java, C, and C++, which is suitable for performance analysis. Actually, this representation uses many ideas common to procedural and object-oriented programming, and can be extend to support other languages in the future. We also designed an XML-based protocol for the communication between instrumentation/monitoring tools and performance analysis tools. Our design covers the typical operations present in today's instrumentation tools and the needs of current performance analysis tools.

We do not require that everything that *can* be described with our representation *be* described, or that an instrumentation/monitoring tool fully support all of the possible operations in the communication protocol we propose. For example, when generating the XML representation from a binary file, less information will be available compared to one generated form the source code. The representation in this case may be extremely reduced, but it will still be valid.

We chose XML because it is language and platform independent and because there is already extensive support for traversing XML documents in several languages. With little or no effort, performance tools could change the target language or platform, or even be extended to analyze and compare programs running in heterogeneous environments.

2. Java-based language for specification of performance bottlenecks

We developed a generic performance specification language based on Java for modeling experiment-related data and performance bottlenecks in distributed and parallel programs. This language employs several object-oriented concepts, like polymorphism, abstract classes, and reflection, to describe performance problems. Our language allows, for instance, to describe new problems based on existing ones and to relate problems among each other. We also defined some mechanisms, like filter and statistics, to help in restricting performance analysis to specific experiment-related data, and to compute statistics based on arbitrary sets of performance values.

3. A portable instrumentation/monitoring tool based on state-of-the-art advances to the Java platform

We developed Twilight, a tool for both static and dynamic instrumentation of Java programs. To our knowledge, Twilight is the first instrumentation tool that makes use of the instrumentation and monitoring features recently introduced to the Java platform.

Twilight can parse Java source codes and instrument any code region with a single entry point (multiple exit points are allowed). Class files (bytecodes) can also be parsed and have instrumentation inserted or removed, with the advantage that the changed version can be dynamically reloaded. Twilight can measure several metrics, some of them directly, using the new monitoring API (like the accumulated garbage collection time or the accumulated time a thread has blocked to enter or reenter a critical section), and some indirectly, through instrumentation of the Java API (like number of times a hash table was resized or time spent receiving messages in RMI calls).

Twilight was designed to work in a distributed environment, allowing remote connections from other tools. Such tools may be built in any language and run on any platform, since all communication with Twilight uses the XML-based protocols described above.

4. An extensible and customizable performance analysis tool

We developed Aksum, a highly customizable and flexible system for semi-automatic search of performance problems. In Aksum, the search for performance problems (properties) is user-controllable by restricting the performance analysis to specific code regions, by creating new or modifying existing property specifications and property hierarchies, by providing thresholds that define whether a property is critical, and by indicating conditions under which the search for properties can stop. Based on the performance properties that must be computed, Aksum automatically determines the raw performance data to be collected and, if the user does not specify code regions to look at, decides which code regions must be instrumented. Following the recommendations of Pancake [108], Aksum allows the user to change the "perspective" from which data is viewed and, through a filtering mechanism, examine properties at various levels of detail.

Aksum can generate multiple experiments and compare their performance outcome, thanks to an integrated experiment manager, which allows the user to provide a set of input parameters (like problem and machine sizes, or options for the compiler) and how they can be combined to generate several experiments. Since complex interaction in the application may lead to livelocks or starvation, preventing the application's completion for some sets of input parameters, the experiment manager also allows providing the amount of time an experiment has to finish.

Aksum supports currently Fortran and Java applications through interfaces with two different instrumentation and monitoring systems: SCALEA [147] and Twilight (Chapter 6). The modular architecture of our tool allows, however, the integration of several other instrumentation and monitoring tools in a relatively easy way.

5. Systematic performance analysis based on overhead classification

There is a cause-consequence relationship between the overheads and the performance problems that are commonly found in applications. Once this relationship is known, it can be used to steer the process of performance analysis, since it provides hints about which aspects of the analysis should be better explored and which can be postponed. We studied this relationship and used it to guide the process of searching performance bottlenecks in Aksum.

6. Interpretation of performance data

Many overheads found in an application are irrelevant for the performance, or are not so important when compared to other overheads and to the problems that they cause in the application performance. Nevertheless, tools for performance analysis usually leave the interpretation of the performance data to the user: They provide several absolute numbers and charts but without indicating which of them require more attention and without providing a way of comparing these number and charts with each other.

In Aksum, performance data is always converted to a single value, called severity. Severity values are built in such a way that larger values correspond to the most severe problems, that is, a total order relationship between the problems found can be built using the severity values. Moreover, there are an upper and a lower bound for severity values, which allows one to quickly estimate the magnitude and importance of a problem in the overall performance.

7. Performance analysis based on reinforcement learning

We formalized several concepts used in performance analysis and used this formalization to propose a new way of modeling the performance analysis problem based on reinforcement learning. Reinforcement learning is a branch of artificial intelligence based on trial-and-error and, therefore, well suited for solving inherently empirical problems. At the same time, reinforcement learning has a strong theoretical background that can be used to justify the pragmatic decisions taken during the performance analysis of an application.

8. Improvement in the quality of the Java API and in the Java virtual machine

During the development of Aksum and Twilight, we could find several bugs in the Java API implementation and in the Sun's Java virtual machine. We have always submitted these bugs to Sun, and most of them were fixed. These bugs are listed in Appendix F.

1.3. Outline

The rest of this thesis is organized as follows:

Chapter 2 shows computer architectures and programming models at which our work is targeted. We also define some terms used throughout the thesis.

Chapter 3 discusses some work related to ours, some of which served as source of inspiration for many ideas in this thesis.

Chapter 4 explains our proposal for XML-based interfaces for instrumentation tools and their potential to reduce the effort to adapt performance analysis tools.

In Chapter 5, we propose a powerful representation for performance data and properties written in Java. The representation assumes the models outlined in Chapter 2. This chapter also shows how a normalized value can be computed for several performance properties.

Chapter 6 shows a sophisticated instrumentation and monitoring tool for Java, based on the ideas discussed in Chapter 4. The tool is based on some state-of-the art advances in the Java platform.

Chapter 7 details our tool for semi-automatic performance analysis, Aksum: how it systematically looks for performance problems in an application and how it interprets the performance data collected in order to provide useful information to the user.

Chapter 8 introduces a new approach for improving the efficiency of instrumentation and analysis using reinforcement learning techniques.

Chapter 9 shows experimental results that illustrate the efficacy of our approach using several real-world applications.

Chapter 10 summarizes and concludes the thesis, outlining future work.

2

Model

New computer architectures and programming models are constantly being developed or improved, while others are superseded and die out. In this chapter, we show current trends and delineate the scope of our work by presenting an overview of computer architectures found today and the programming models typical for such architectures. We also define some terms common in the field of performance analysis that will be used throughout our work, many of which common to several architectures and programming models.

2.1. Architectures

Traditionally, and specially when dealing with parallel computing, computer architectures were classified using the taxonomy developed by Flynn [42]:

- Single Instruction, Single Data (SISD) refers to conventional sequential machines; given a stream of instructions and the data to be processed, these machines will execute one instruction at a time, for a single piece of data.
- Single Instruction, Multiple Data (SIMD) refers to machines which can process only one instruction a time, but for several pieces of data simultaneously.
- Multiple Instructions, Multiple Data (MIMD) refers to machines able to execute several instructions simultaneously for different (and possibly independent) pieces of data.

This classification is of little help now, as multiple general-purpose processors are dominant [23]. Moreover, most CPUs today include some degree of parallelism, like a set of SIMD instructions or hyper-threading technology, which makes the term “conventional sequential machine” sound not quite correct.

There has been some divergence when classifying system architectures, especially those for high performance computing [7, 20, 23, 30, 43, 145]. The literature tries to reflect the current trends in the market by employing several criteria, like memory organization, type of interconnection, and programming model likely to be used in a given architecture, but no consensus has been achieved yet, and some terms are abused or loosely and subjectively employed. As new architectures come to the market, current classifications need to be extended, adapted or relaxed; authors are still debating about a taxonomy which is useful and where every architecture can be unmistakably classified. In this work, we will adopt a generic model made up of complete computers connected to each other through a communication network. Some special cases of this model are discussed in the following.

2.1.1. Shared Memory Systems

A system with several processors sharing a global address space is called a shared memory [23], tightly coupled or multiprocessor system [144]. In particular, the system is called symmetric multiprocessor, or SMP, if the cost of a memory access is the same for all processors in the system. In contrast, in a non-uniform memory access (NUMA) system, each processor has a local memory and the cost of accessing the memory varies depending on the access being local or not.

In an SMP system, the memory is usually connected to the processors through a shared bus, as depicted in Figure 1(a). Since processors may access the bus only one at a time, the bus may become a source of contention and cause the processors to stall. For this reason, SMP systems usually have a small number of processors.

The NUMA architecture, shown in Figure 1(b) has been designed to overcome the scalability problem of SMP. In this architecture, each processor (or each small set of processors) has its own local memory, which allows parallel accesses to the memory as long as each processor does not access the local memory of other processes. The hardware may provide a virtually or physically shared address space [23], but this task may also be left to the software (the operating system kernel or a runtime library routine) [132]. In either case, the system is also called distributed shared memory (DSM) system if coherent replication is also provided, that is, if the value obtained when a memory location is read is the last value written to that location [23].

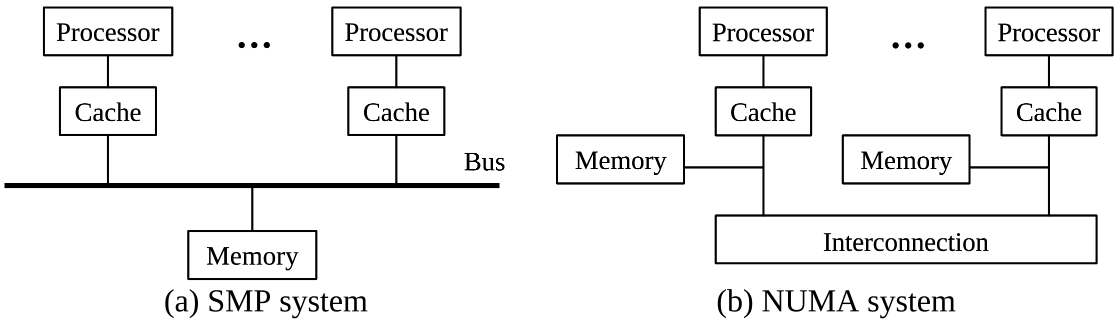


Figure 1. Shared memory systems

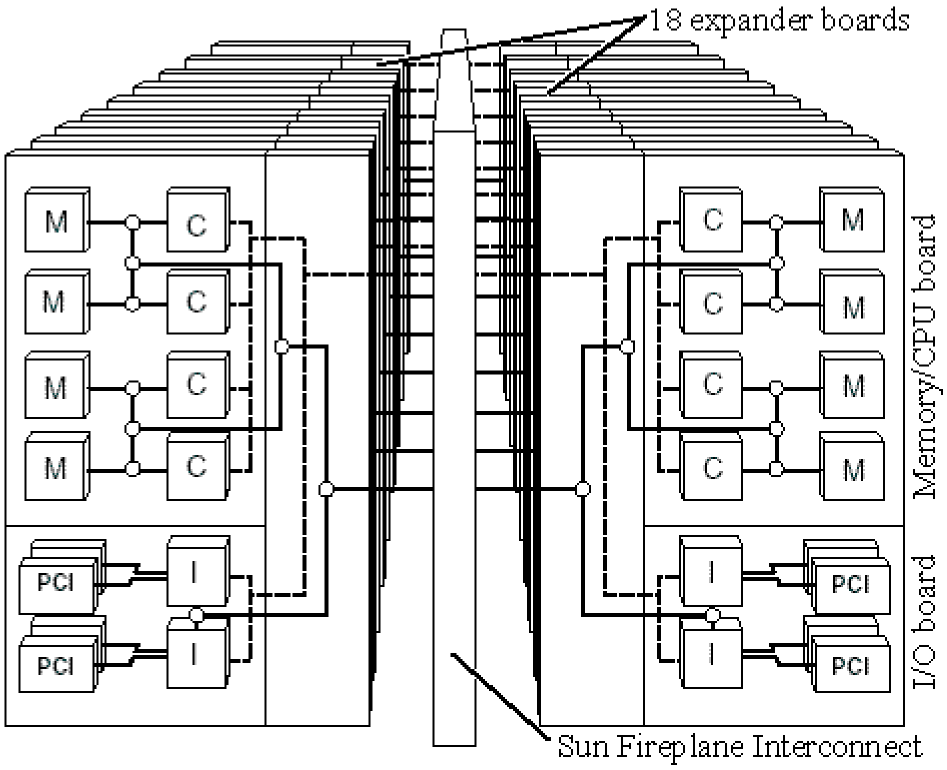


Figure 2. Architecture of Sun Fire E25K

Sun Fire E25K [136], illustrated in Figure 2, is an example of a NUMA shared-memory system. It supports up to 18 board sets, each of which composed of a memory/CPU board, an I/O board, and an expander board. Each memory/CPU board contains 4 CPUs and a memory capacity of 64 Mbytes (16 Mbytes per CPU). All board sets are connected to each other through three 18x18 crossbars Sun Fireplane Interconnect, two of which responsible for keeping the memory coherence, and the other responsible for data transfer between boards. A CPU/memory board has two halves, and each half contains two CPUs. The latency (time for a single data item to be delivered from memory to a CPU) depends on which memory bank is accessed, and is shown in Table 1.

Owner of the memory bank accessed	*Time and clock cycles*
Same CPU	180ns, 27 cycles
The other CPU on the same half of the board	193ns, 29 cycles
A CPU on the other half of the board	207ns, 31 cycles
A CPU on another board (data in the coherence directory cache)	333ns, 50 cycles
A CPU on another board (data not in the coherence directory cache)	440ns, 66 cycles

Table 1. Latency for Sun Fire E25K (best case)

Another example of a NUMA shared-memory system is the HP 9000 Superdome [62], whose architecture is illustrated in Figure 3. The basic components in this architecture are cells (or cell boards), crossbar backplanes, and I/O subsystems. The system supports a maximum of 16 cells (eight in the left cabinet, eight in the right cabinet), where each cell is an SMP with up to eight processors and 64 Mbytes of memory. The four crossbars are fully connected to each other, and each of them is connected to up to fours cell boards. All links have the same bandwidth and latency, and three latency domains exist: cell local (memory and CPU on the same cell), crossbar local (memory and CPU in different cells, but the cells are connected to the same crossbar), and crossbar remote. Table 2 shows the average latencies for HP 9000 Superdome assuming an equally distributed traffic to all memory controllers.

Our third example of a NUMA system is the SGI Altix 3000 series [159]. C-Bricks, M-Bricks, and R-Bricks are some of the basic components in this system. A C-Brick is composed of two SMP nodes, each of which containing 2 processors and up to 16 Mbytes of memory. A M-Brick is like a C-Brick, but without processors, which allows one to add more memory to the system without having to add more processors. R-Bricks are routers connecting M-Bricks and C-Bricks in a fat-tree network topology, as shown in Figure 4. This picture illustrates the current largest possible configuration, with 128 C-Bricks (512 processors) represented as small squares in the center, circles representing R-Bricks, and lines representing the connection (cables) between the bricks. The cables have different sizes, and the latency of memory accesses depends on these sizes and on the number of routers needed to reach the memory in another brick. Table 3 shows the maximum number of hops in the fat-tree topology, as well as the bandwidth for NUMAlink™ 4, the technology used to implement the network in SGI Altix 3700 Bx2, which is the newest model of the series.

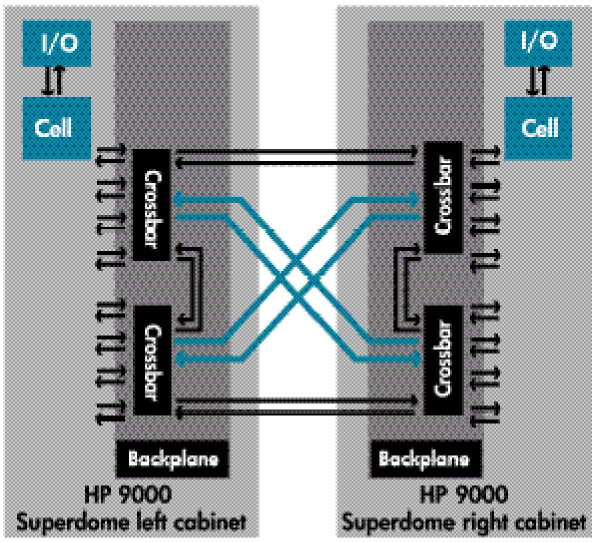


Figure 3. Architecture of HP 9000 Superdome

Number of CPUs	*Time*
8	246ns
16	330ns
32	371ns
64	417ns
128	440ns

Table 2. Average latencies for a memory access in HP 9000 Superdome

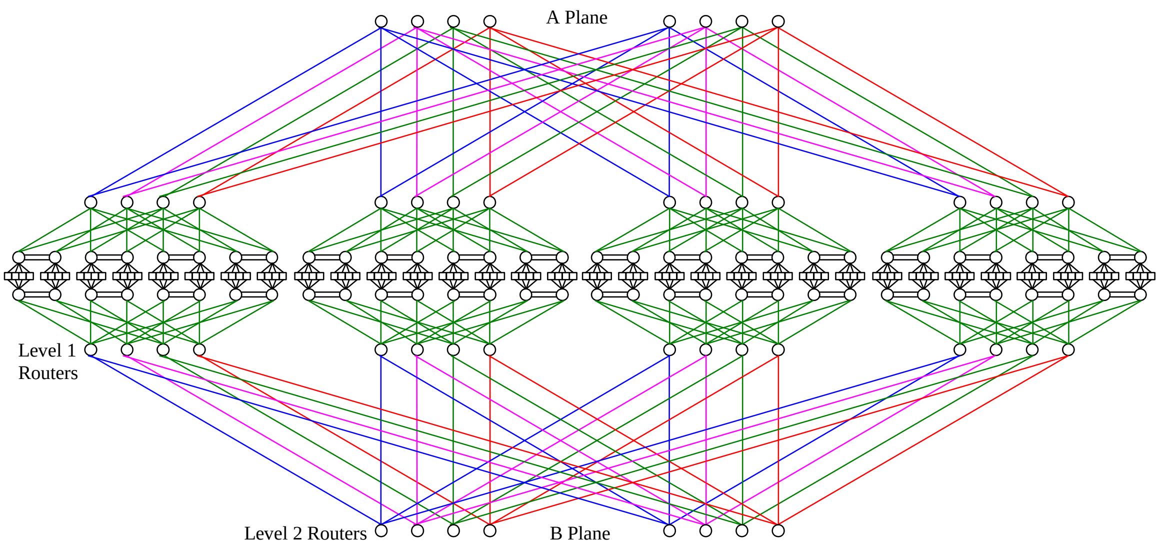


Figure 4. 512-processor dual "fat tree" interconnect topology in the SGI Altix 3000 series

Maximum number of processors	*Bandwidth (Mbytes/s/processor)*	*Maximum number of hops*
16	1600	3
32	1600	4
64	800	4
128	800	5
256	800	5
512	800	7

Table 3. Bandwidth and maximum number of hops for NUMAlink™ 4

The SGI Altix 3700 Bx2 is an example of a Massively Parallel Processing system (MPP). MPPs "employ sophisticated packaging and a fast dedicated proprietary network so that a very large number of processors can be located in a confined space with high-bandwidth and low-latency communication." [23] Although the first two examples given also fit in this definition, they are not considered MPPs (for instance, in the list of the world's top 500 supercomputers [146]) because they were developed and marketed for purposes other than high performance computing [30].

2.1.2. Distributed-memory Systems

A distributed memory system is made up of several complete computers connected through some kind of scalable network, which is used in the communication between the several processors and memories in the system [23]. Communication between processors in different machines is done transmitting data from the memory of a processor (the sender) to the memory of other processor (the receiver) through the network. We will call *node* any computer connected in the distributed memory system, and use the term *site* to refer to some set of nodes. A site usually encompasses the nodes that belong to a sub-network or organization.

Distributed memory systems may be built up from shared memory systems, including MPPs. The Columbia Supercomputer [99], for example, is a distributed memory system composed of thirteen SGI Altix 3700 and eight SGI Altix 3700 Bx2 (described above). Because it was built up from MPPs, the Columbia Supercomputer is also considered an MPP, being currently ranked third on the list of the world's top 500 supercomputers (June 2005).

Clusters represent a special case of distributed-memory systems that use standardized high-performance local area networks (LANs) like Myrinet [97], Infiniband [65], and QsNet [114], and commonly have a front end that "acts as an intermediary between a collection of compute servers and a large number of users at terminals or remote machines" [23]. The term "cluster" is somewhat controversial: In [7], the NUMA systems described in Section 2.1.1 are considered clusters as well, while in [30] it is advocated that the term cluster should be applied only to commodity clusters, which comprise exclusively "commodity computing subsystems and commercial networks such that the computing nodes are developed and employed in standalone configurations for broad (even mass) commercial markets and the networks are dedicated to the private use of the cluster (non-worldly)." This recommendation is adopted in the list of the world's top 500 supercomputers.

Two subclasses of commodity clusters are particularly important: *Constellations* are clusters where "there are more microprocessors in a node than there are nodes in the commodity cluster" [30]; *Beowulf clusters* are clusters based on commodity off-the-shelf hardware and open source operating system [9], although some authors also consider it possible to have a Beowulf cluster that uses a commercial operating system like Windows [139, 140].

The list of the world's top 500 supercomputers contains currently 304 clusters and 79 constellations (the remaining 117 computers are MPPs). MareNostrum [142], for example, which is ranked fifth on the list, is the fastest supercomputer in Europe and the fastest cluster in the world. MareNostrum is composed of 2406 IBM dual-processor BladeCenter JS20 servers connected by a Myrinet network and using Linux as operating system. Ranked 172nd on the same list is the fastest constellation in Europe and fifth fastest constellation in the world, a Sun Fire 25K/6900 Cluster in the RWTH Aachen University (not that it is not called Sun Fire 25K/6900 Constellation). The system is built up from Sun Fire 25K servers (described in Section 2.1.1), containing a total of 672 processors.

In this work, we will employ the term cluster only to commodity clusters, reserving the generic term *distributed system* to non-dedicated networks of autonomous workstations, where computers may have owners, are not centrally managed, and can be connected to each other not only through a local area network, but also through a metropolitan or wide area network (MAN/WAN), like the Internet.

2.2. Programming and Execution Models for Sequential and Parallel Processing

Fundamental for the programming and execution models used in this work is the concept of *process*. A process is the abstraction of a program in execution, consisting of an execution environment and one or more *threads* of control. The execution environment contains resources and an address space that are shared among all the threads, and also some resources which are private for each thread in the process, like registers and the stack. The real implementation of threads in an operating system may not reflect the abstraction above (notably in Linux), but this is not relevant for our programming models.

2.2.1. Message-passing Model

In the message-passing model, there is a collection of single-threaded processes that communicate with each other through explicit I/O operations (network transactions), represented at the user level as *send* and *receive* operations. The send operation "specifies a local data buffer that is to be transmitted and a receiving process (typically on a remote processor)", while the receive operation "specifies a sending process and a local data buffer into which the transmitted data is to be placed" [23]. As shown in Figure 5, the matching *send* and *receive* cause a data transfer between one process and the other.

In the context of this work, the message-passing model is represented by Fortran programs written using MPI-1 (Message-Passing Interface version 1). MPI "specifies the names, calling sequences, and results of subroutines to be called from Fortran programs, the functions to be called from C programs, and the classes and methods that make up the MPI C++ library" [50].

In MPI-2, it is also possible to make use of shared memory and have multiple threads of execution, but we will not cover these cases. With MPI-1, a fixed number of *identical* processes, which operate on different pieces of data, is created at startup; new processes are neither created after the startup nor destroyed before the program ends. Such programs are referred to as SPMD (single program multiple data).

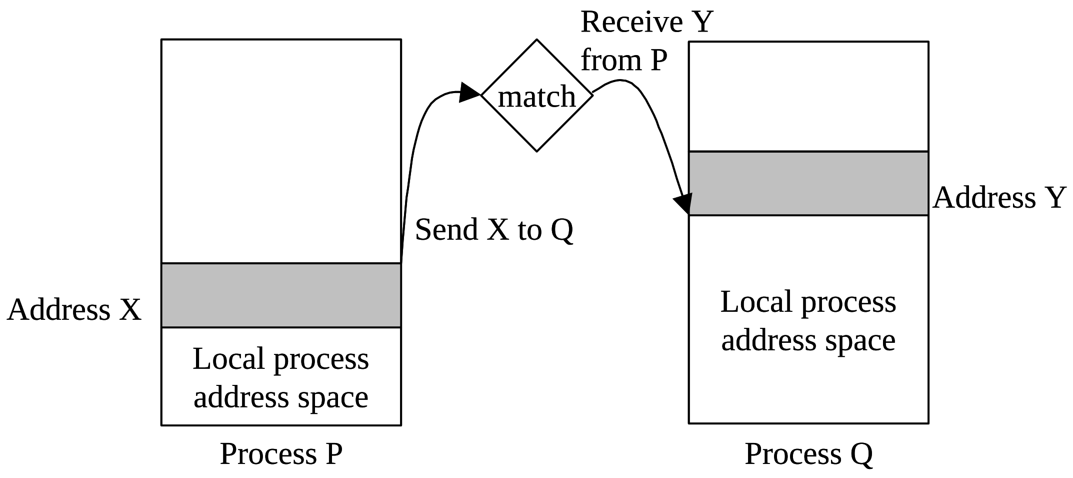


Figure 5. User-level send/receive message-passing abstraction [23]

2.2.2. Fork-join Model

The fork-join model, shown in Figure 6, is one of the programming models we deal with. The process executes a program subdivided into sequential and parallel regions. A process may dynamically create, synchronize, and terminate threads during execution of the program. In a sequential region only one thread within the process, called the "master thread", is active (executes the code in the region), while in a parallel region several threads may be active simultaneously. The master thread creates (forks) new threads at the beginning of a parallel region. At the end of the parallel region, the active threads synchronize and all except the master thread terminate (join). Active threads exchange data by using the shared memory.

It is implementation and language dependent if the threads are really created at the beginning of a parallel region and terminated at the end, or, which may be less expensive, if they are created when the execution starts and just switch from the state inactive to active when the parallel region starts and from active to inactive at the end. The distribution of the computational load among active threads within a parallel region is language and implementation dependent as well.

The fork-join model as above described is adequate for shared memory systems or single SMP nodes in a distributed memory system (specially constellations, described in Section 2.1.2). In the context of this work, the fork-join model is represented by Fortran programs written using OpenMP (MP stands for Multi Processing). OpenMP is "a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C, C++ and Fortran programs" [107].

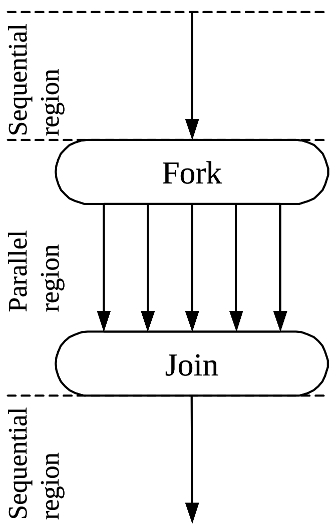


Figure 6. Fork-join programming model

2.2.3. Hybrid Model

The hybrid model, shown in Figure 7, can be seen as a generalization of the message-passing model and the fork-join model. As in the fork-join model, each process is subdivided into sequential and parallel regions and may dynamically create, synchronize, and terminate threads. In addiction, active threads associated with different processes use generic *send* and *receive* operations to exchange data. These operations can be executed in both sequential and parallel regions. Note that this model reduces to the fork-join model if there is only one process, and to the message-passing model if there is always only one active thread per process.

The hybrid model is adequate for distributed memory systems composed of shared memory subsystems. In the context of this work, the hybrid model is represented by Fortran programs written using both MPI-1 and OpenMP.

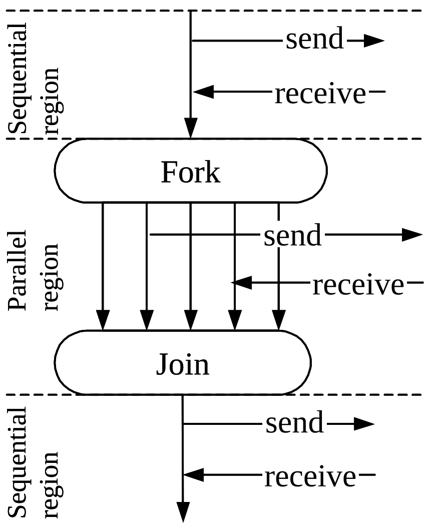


Figure 7. Hybrid programming model

2.2.4. Generic Distributed Multithreaded Model

The generic distributed multithreaded model, shown in Figure 8, is a generalization of the hybrid model. Each thread may create (fork) new threads at any time, and it may or may not await the end of other thread. In fact, a thread may even wait for the end of threads that it did not create, or it may end only when the machine is shut down.

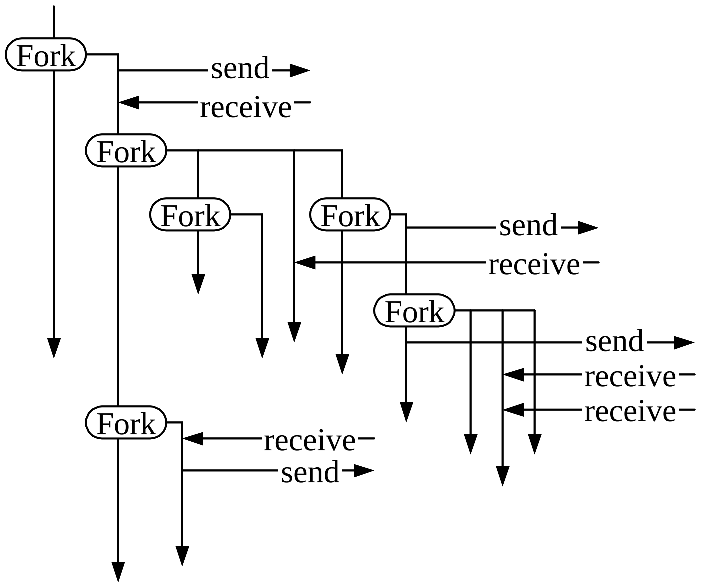


Figure 8. Generic multithreaded model

Threads in a process communicate with threads in another processes using the operations *send* and *receive*, but protocols may be used to create higher-level operations implemented on top of *send* and *receive*, like remote procedure calls (RPC) or remote method invocations (RMI).

Threads are created not only to increase the parallelism, but also to improve the throughput or to simplify the application design; therefore, there need not be a 1-to-1 mapping between threads and processors. The use of send and receive operations, however, will be found in applications targeted at distributed memory systems. In this work, the generic distributed multithreaded model will be represented by programs written in Java, in particular using the following models:

- Client-server model

  A set of processes called *servers*, running on powerful computers and responsible for the main computation, waits for requests from other (remote) processes called *clients*. The client may use only few resources of the machine where it runs (typically only network connections and a graphical user interface), in which case it is called a thin client, or it may be responsible also for some computation, when it is called a fat or thick client.
- N-tier model

  A special case of client-server model, where servers may also act as clients of other servers and each server is responsible for some piece of the logic that solves a problem. A Three-tier model is the most typical case, where the client represents the first tier, the server responsible for the computation is the second tier, and a database server the third.
- Peer-to-peer model

Several processes act simultaneously as server and client and are responsible for the same logic. Each process (peer) may forward a piece or all of its work to other peers for several reasons; for instance, a peer is overloaded, or it is not able to fulfil a request (but knows a peer that might be). The peer-to-peer model can also be combined with the client-server model; for instance, each peer may be client of a database server.

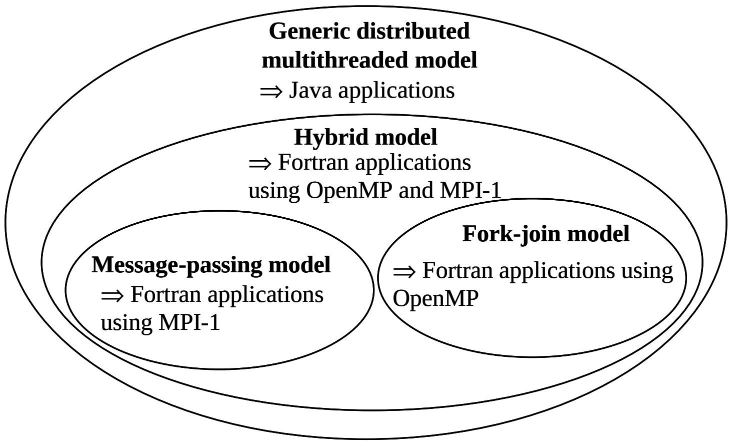

Generic distributed multithreaded model
⇒ Java applications
Hybrid model
⇒ Fortran applications using OpenMP and MPI-1
Message-passing model
⇒ Fortran applications using MPI-1
Fork-join model
⇒ Fortran applications using OpenMP

Figure 9. Relationship between the generic distributed multithreaded model, the hybrid model, the fork-join model, and the message-passing model.

Figure 9 shows the relationship of the models we cover in our work. Applications that fit the hybrid model (including the message-passing and the fork-join model) will be called “parallel applications”, while applications that fit the generic distributed multithreaded model but not the hybrid model will be called “distributed applications” (if they are intended for making use of more than one node) or “concurrent applications” (if they are intended for using only one node).

2.2.5. Data Parallelism Model

Some algorithms can be seen as a sequence of identical (or at least very similar) steps applied to different pieces of a regular data structure, like vectors or matrices. The parallelism inherent in these algorithms, called data parallelism, is realized by assigning pieces of data to different processes, which then execute the computation on the data received.

Programs following the data parallelism model are usually automatically converted to one of the previous models by a compiler, which makes such programs easier to understand and maintain. As the data granularity is often too small, the programmer needs to insert directives in the code in order to help the compiler to determine the best data distribution. HPF (High Performance Fortran [59]) is a well-known example of a data parallel language; a traditional compiler for this language, the Portland Group's HPF compiler [113], can generate, from HPF programs, equivalent SPMD programs that make use of MPI, PVM [141], or sockets and shared memory to exchange data.

We will not deal directly with this model, although later we will show the performance analysis of hybrid OpenMP/MPI-1 programs automatically generated by the Vienna Fortran compiler [8] from programs following the data parallelism model.

2.3. Performance Analysis

There are several places one can look at in order to carry out the performance analysis of an application: its source code can be examined [135, 154], the compiled code, or one or more executions of the application, possibly for different sets of input values. The last case is the focus of our work.

Some basic steps can be identified in the performance analysis: data collection, data transformation, and data visualization or interpretation. These steps are discussed in the following.

2.3.1. Data Collection

Data collection refers to the process of obtaining performance data from an application in execution. The two main techniques for collecting performance data are *tracing* and *profiling*. Tracing typically consists in generating detailed log files that contain time-stamped records representing events significant in the program execution [43]. Tracing allows determining causal relationships between events (which may requires a synchronized clock for applications that make use of more than one node), for instance by identifying well-known event patterns that are indicative of performance problems [36, 158].

Profiling consists of measuring and recording the value of some *metric* whenever specific events occur. Wall clock execution time, CPU execution time, time spent sending messages, and number of cache misses are examples of metrics. They can be measured, for instance, at specific intervals (which is known as *sampling*) or when the execution flow reaches some point in the code. Profiling helps one to find *where* and *when* an application is spending more time or any other resource, like bandwidth.

The process of inserting in the application extra code for profiling and tracing is called *instrumentation*. The code can be inserted statically, that is, before the application is executed, or dynamically, while the application is running. We call *instrumentation tool* any tool that can instrument an application, and *monitoring tool* any tool that can effectively collect the data. If the tool can both instrument the code and collect data, it will be called *instrumentation and monitoring tool*. Such tools are commonly libraries or autonomous agent, but note that they are not responsible for deciding which events should be traced or profiled and, consequently, what must be instrumented.

2.3.2. Transformation, Visualization, and Analysis of Performance Data

The data collected is usually output in a generic format and needs to be transformed before it can be visualized or analyzed; moreover, the amount of data generated may be too large to be examined by a person. Typical transformations group the data according to some criteria, compute statistics like average or maximum, and create data structures out of the collected data that are suitable for some sort of analysis.

The transformed data can be displayed, commonly using elaborate graphical user interfaces, or automatically analyzed, sparing the user the interpretation of the data, which can be very painful for non-expert users, as Figure 10 shows. The analysis can be done while the application is running, in which case it is called *online* or *dynamic* analysis. The term *post-mortem* analysis (or *static* analysis) is used for the analysis done after the program has executed.

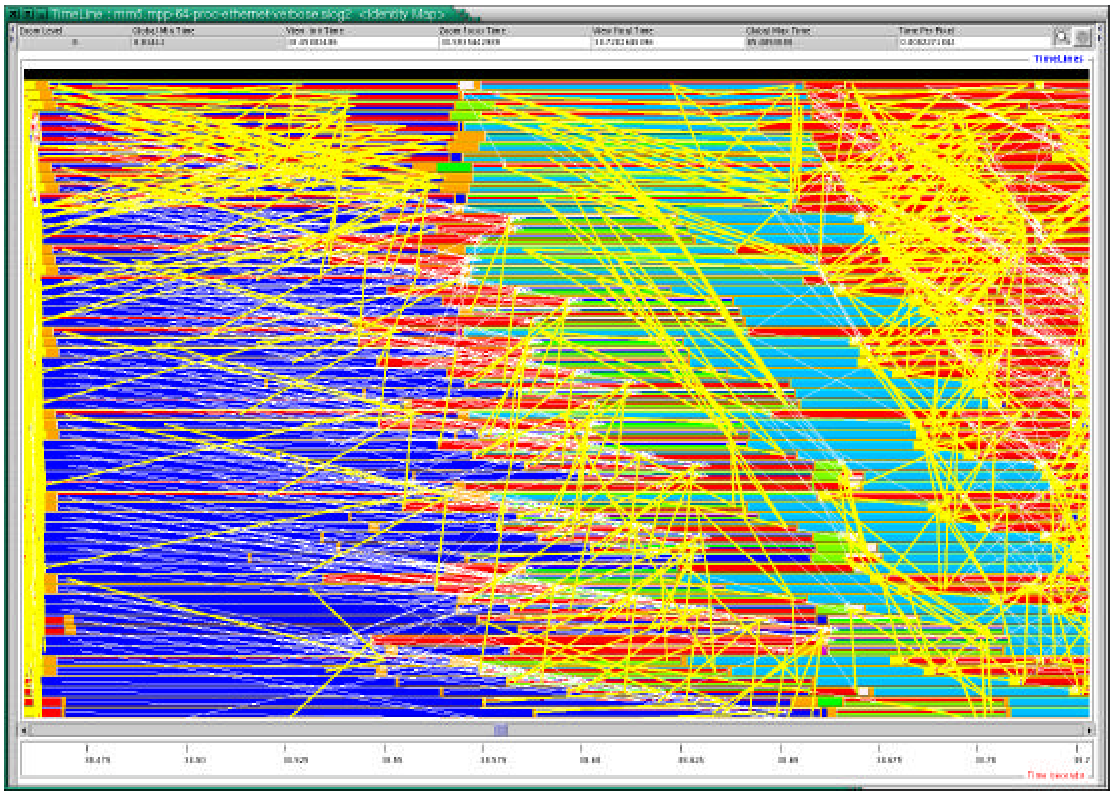


Figure 10. Jumpshot-4 [111], a tool for performance visualization of parallel programs.

A tool that can interpret the data collected, output conclusions about the performance of an application, and pinpoint which code regions are the culprits for the low performance is called a *performance analysis tool*. It is also the task of the performance analysis tool to decide which events should be profiled or traced; these events need to be carefully chosen, as a high level of detail may perturb the performance (probe effect) and change the program behavior, hiding performance problems, creating new ones, and even preventing the application from running. A popular technique today consists in successive event refinements, where more specific and fine-grained events are selected depending on the performance problems that are found. Other techniques are discussed in Chapter 3.

2.3.3. Temporal Performance Overheads

The total temporal overhead in an application execution can be approximately modeled depending on the execution model. We will adopt one definition for parallel and one for distributed applications.

In order to define the total temporal overhead for parallel applications, we need two executions of an application: one, called the sequential execution, employs only one processor, while the other, called the parallel execution, uses more than one processor. Both executions must use the same set of input parameters and the same application code. Assume that the sequential execution spent T_s units of time and that the parallel execution spent T_p units of time using a set of q processors, all of them identical to the processor used in the sequential execution. From [43], we define the *total temporal overhead* T_o as:

$$T_o = qT_p - T_s$$

This definition poses some problems. First, due to time or space constraints, it may be impossible to have a set of input parameters (or, more specifically, a problem size) for which the application can be sequentially and parallel executed. Second, the parallel execution time may not be a lower bound, since an increased number of processors is likely to increase also the amount of available memory, registers and cache, which may reduce the number of accesses to the main memory and the use of virtual memory, consequently also reducing the execution time. Finally, we assumed that the same application code is used for both the sequential and the parallel execution, even though a code that has been adapted for being used with several processors may contain overheads that are not present in a version optimized for the sequential execution. Nevertheless, this model is useful to estimate the amount of unidentified overhead, defined below.

For distributed and concurrent applications, we will model the total temporal overhead of a code region *r* as the difference between the wall clock execution time and the CPU execution time of *r*. Again, this model may not reflect the real total temporal overhead. The reason lies in the nature of the typical distributed and concurrent applications; they may create several threads that do remain most of the time idle, for instance to improve the application's throughput or responsiveness. But, as it happens with the representation for parallel applications, this model may be useful to estimate the unidentified overhead in some situations.

2.3.4. Sources of Overhead

In the following, we discuss some temporal overheads and their sources in light of the architectures, programming and execution models already defined. The overheads are presented hierarchically based on classifications found in the literature [6, 13, 96, 147]. The classification shown does not imply, however, that all overheads are measurable in any situation.

- Data Movement: data moved from one entity to other
  - File I/O: data moved from a file to the memory and vice versa
    - Local (the file and the memory belong to the same node)
      - Read
      - Write
    - Remote (file and memory belong to different nodes)
      - Read
      - Write
  - Communication: data moved, through explicit I/O operations, from the local memory to the remote memory and vice versa
    - Point-to-point: only two nodes are involved in the transfer
      - Receive
      - Send
    - Collective: more than two nodes are involved in the transfer
  - Remote memory: data moved from the local memory to the remote memory and vice versa; the transfer is controlled by hardware in the presence of a single address space
    - Load
    - Store
  - Local memory: memory transactions in a single node
    - Load: data is load from the memory to the processor

    - Store: data is stored in the memory
    - Level 3 to level 2: data moves from the tertiary cache to the secondary cache
    - Level 2 to level 1: data moves from the secondary cache to the main cache
    - TLB miss: the Translation Lookaside Buffer had to be updated
    - Page fault: a page of the virtual memory does not map to any page frame in the physical memory and needs to be loaded
- Synchronization
  - Single address space: locks, semaphores, condition variables, among others
  - Multiple address space: involves processes that do not share the same address space and therefore need to make use of explicit I/O operations (like barriers in MPI programs running on distributed memory systems)
- Additional computation: reflects changes in the parallel version of an application compared to the original sequential version
  - Algorithmic changes: results from changes in the algorithm
  - Implementation changes: results from changes done by a compiler
- Control of parallelism: code to manage parallelism
  - Initialization and finalization of resources
    - Threads
    - Processes
    - Sockets
  - Scheduling: computation of work to be assigned to different processes and threads
- Loss of parallelism: caused by imperfect parallelization of a program
  - Unparallelized code: executed sequentially
  - Partially parallelized code: executed by several, but not all, processes or threads available
  - Replicated code: executed by several processes or threads with the same input data

Ideally, this classification should be complete and orthogonal [22], that is, "any source of overhead should be classifiable within the scheme" and "no source of overhead should appear in two different categories, unless one of the categories is a subset of the other." [13]. On the other hand, orthogonality may be in practice impossible if the available technology does not allow a monitoring tool to definitely classify an overhead source. For instance, loading a value from the memory implies that the value will be also copied into the caches, and a monitoring tool may not be able to detect this overlapping.

Assuming that the overheads above account separately for the total temporal overhead, their sum constitutes the *identified overhead*, while the difference between the total temporal overhead and the identified overhead is called *unidentified overhead* [6].

In general, overheads cannot be completely removed from an application; for example, data must be brought at least once from the memory to the cache, and programs based on the message-passing model will always contain some communication overhead. One can, however, try to reduce the overheads in the process of tuning an application (for a generic or specific hardware or software environment), until the performance requirements are met. For example, as temporal overheads commonly identify time that a processor is idle, one can try to overlap the computation of other threads and the idle time caused by the overhead. Another technique is to reduce an overhead to the detriment of other. For instance, by inlining a function one can reduce the overhead of calling a subroutine and returning, but at cost of increasing the space requirements.

As already mentioned, overheads may not necessarily imply that a processor is idle. Technologies like hyper-threading, for example, allows other threads to execute during the cycles one thread is waiting for a value loaded from the main memory (or some other resource, which is called latency tolerance [23]). In processors that implement these technologies, the cost of a cache miss can be compensated and is more difficult to analyze.

3

Related Work

The traditional and most intensively studied approach for the performance analysis of applications consists in monitoring their executions (possibly using an instrumented version and several sets of input parameters) and inferring some conclusion about the performance using the data collected during the monitoring process. Several issues in this process were attacked, in different ways, by many researchers; in this chapter, we discuss some of these issues and how they were dealt with, as well as some alternative approaches adopted.

3.1. Front-end Interfaces

The task of writing a parser or adapting an existing parser for specific purposes (in our case, for instrumenting an application) is far from trivial. This motivated the development of various formats and interfaces aiming to be a neutral layer between parsers (front ends) and other tools.

The Program Database Toolkit PDT [85] uses compile-time information to create a complete database of high-level program information structured for well-defined and uniform access by tools and applications. PDT is composed of 1) “intermediate language (IL) analyzers,” which are interfaces to existing parsers; 2) a relatively compact and portable program database, containing the information generated by the IL analyzer on source constructs (including source code locations); and 3) a C++ interface, called DUCTAPE, for the program database. Currently, PDT provides IL analyzers for the C/C++ front end of EDG [33], for the Fortran 77/90 front end of Mutek Solutions Ltd., and for the Fortran 77/90/95 parser of Cleanscape Inc. [17].

GENOA [27] is a portable, language-independent querying mechanism with traversal and iteration operators for abstract semantics graphs (ASGs), which are abstract syntax trees annotated with semantic information. Using GENII, a specification that describes the data model of the ASG built by a particular front end, the author of GENOA wrote interfaces for four C++ front ends (Cin, Cfront, REPRISE and EDG).

ASTLOG [21] was developed as a Prolog variant for doing syntactic-level analysis for C/C++ programs (Prolog was not used in order to avoid the overhead of translating the source syntactic structures into the form of a Prolog database.) ASTLOG needs a C/C++ compiler front end that provides a (C++) interface to the syntactic and semantic data structures built during the parse of a given program; the version of ASTLOG presented in [21] used the abstract syntax tree provided by the program analysis group of Microsoft Research.

WHIRL [156] is the binary representation used in the SGI Pro64 compiler (now Open64 compiler suite [105]), designed to support compilation of program code written in C, C++, Fortran 77, Fortran 90, and Java to multiple target processor architectures. WHIRL has in fact 5 levels: very high, high, mid, low, and very low, where the higher the level, the closer the representation to the source code, and the lower the level, the closer the representation to the machine code. The Open64 infrastructure includes a near commercial-quality front end for Fortran 90 from Cray and gcc-based front ends for C and C++; it is used, for example, in the Open64/sl project [106], which builds software tools for source-to-source transformation of programs.

JavaML [5] is an XML representation of Java programs similar to an abstract syntax tree and with enough information to be converted back to the source code from which it was generated. The author of JavaML adapted the front-end of the Jikes compiler [72] to generate JavaML representations from Java programs, and suggested that the representation could be extended for other object-oriented languages.

3.2. Tools for Automatic Performance Analysis

Instrumentation of the source code and post-mortem analysis is a long-established technique for performance analysis. Kappa-pi [35] employs a post-mortem performance analysis to search for performance bottlenecks based on trace files of PVM [141] applications. The tool tries to detect patterns in trace files that match performance-problems patterns in a knowledge base (which cannot be modified by the user) an presents suggestions about how the performance of the application can be improved.

Finesse [96] directs the user to the overheads found in the application, and provides guidance for eliminating such overheads. Based on static instrumentation, Finesse focuses on shared memory programs, but its performance analyzer, which tries to find performance overheads, cannot be configured by the user.

Poirot [56] is the design of a software tool that does not depend on any specific programming environment. To build such a design, Poirot's authors gathered several performance tools with the goal of formalizing performance bottlenecks of a parallel program. The search algorithm in Poirot uses a database containing predefined performance bottlenecks to be searched for.

Gerndt and Krumme [51] proposed hypotheses and successive refinements (discussed below) in the design of a performance analysis environment for applications using SVM-Fortran, a shared-memory parallel programming language based on Fortran 77. The environment they proposed has two set of rules: proof rules, containing information about how to prove a hypothesis, and refinement rules, which determine which hypotheses should be tested if a given hypothesis is proven. Their static approach requires that the application be compiled and executed again after a refinement.

The popularity of techniques based on online instrumentation and analysis, which eliminate the necessity of compile and execute the application several times, has constantly increased since Paradyn [89] was released ten years ago. Based on dynamic instrumentation and monitoring of applications, Paradyn was the first tool able to carry out automatic online performance analysis. Among its several components is the Performance Consultant, which automatically searches for performance problems *while* the application is running through *hypotheses* and *successive refinements* [14, 19]. This technique consists in inserting more detailed instrumentation in those places in the application where problems have already been found (that is, hypotheses were proved as true); for example, if a problem is found in an instrumented subroutine, then instrumentation is inserted to monitor the loops and calls in that subroutine. At the same time, if no problem is found in a given piece of code (hypothesis were proved as false), the instrumentation is removed in order to avoid the overhead that the instrumentation code incurs. Paradyn also looks at the call stacks to find functions that executed often and consequently are likely to be application bottlenecks [118]. Paradyn uses Dyninst [60] to dynamically insert instrumentation in or remove instrumentation from a running application. Even though Paradyn is highly configurable (a configuration language is provided to request specific performance data), the Performance Consultant does not allow the user to modify the existing definition of performance problems or to define new performance problems to be searched for.

MATE [92] can monitor PVM applications, detect performance bottlenecks, and modify them in order to improve the performance. MATE also uses Dyninst to insert dynamically the code responsible for generating events and sending them to a central analyzer, as well as to manipulate the image of the process in memory when the application has to be tuned. The tuning can be automatic, if the user does not provide any information about the application, or cooperative, if the user prepared the application for tuning (for instance, by describing what can be changed to obtain better performance).

As mentioned in Section 2.3.2, instrumentation and monitoring may considerably perturb the execution of an application; moreover, the amount of data generated during the execution of monitored applications has always been a concern for performance analysis, aggravated by the increasing number of processes used to execute parallel applications. This motivated the use of statistical techniques that try to infer conclusions from a not fully instrumented application. PHOTON [149] uses sampling techniques to limit the amount of traced events and consequently the amount of data generated. The sampling may be time-based (an event is traced every *t* units of time), counter-based (every *k*-th event is sampled), or random (an event is traced with probability λ). The tool uses a modified MPI library (based on MPICH 1.2.2 [94]), where the header of each message contains a timestamp and a source code location identifier *if and only if* the event that originated the message is sampled.

Vetter and McCracken [151] used a statistical approach to detect scalability problems in MPI applications, basically by computing the correlation between the machine size and the communication time of code regions. A high correlation in a code region indicates that the communication time becomes larger when the machine size grows, which means that such code region may hinder the scalability of the application. The size of the trace files generated during the monitoring of an application depends on the machine size and the number of sites in the application where communication occurs, but does not depend on the execution time of the application.

The statistical approach of Santiago, Rover, and Rodriguez [123] was based not only on communication times, but on several metrics, like writes per second per disk, address translation, page faults per seconds, and accesses to system buffer cache to read or write. Varying the problem size, the algorithm, and the compiler options (controllable factors), they generated several experiments, computed the correlation coefficient between the metrics and the execution time, and used analysis of variance (ANOVA) to determine the influence of the controllable factors on the metrics. Finally, they applied multidimensional data analysis techniques to find (cluster) the metrics that account for most of the variation in the data. Clustering techniques were also used in [88] and [152].

A system for controlling instrumentation overhead, which makes use of an instrumentation cost model [61], was incorporated into Paradyn. The system has two types of costs: predicted and observed. To compute the predicted cost of the instrumentation inserted, the model uses the cost of each instrumentation primitive (which is knows precisely) and an expected frequency of execution, which is estimated using a static model of procedure call frequency and adjusted at runtime. The observed cost is computed by converting to time the cost in machine cycles of the instrumentation (which is measured using a kind of “meta-instrumentation”); the conversion is approximate, using empirical values. Thresholds for the amount of instrumentation overhead tolerated can be defined by the user.

The use of open and modular architectures for developing performance analysis tools has also increased, which facilitates their adaptability and extensibility for new architectures, languages, and program paradigms. TAU [91, 133] is a program and performance analysis tool framework for high-performance parallel and distributed computing. TAU can generate profiles and trace files for C, C++, Fortran 90, Java and Python codes; the user can either insert manually the instrumentation code (calls to the TAU library) or specify which routines are to be instrumented, so that TAU can instrument them automatically. The automatic instrumentation of C, C++ and Fortran 90 is done using PDT (described in Section 3.1) for source codes, and Dyninst (described above) for binaries, while the automatic instrumentation of Java programs is done at runtime using the JVMPI, a (now deprecated) monitoring and profiling interface [71], and native C functions. In any case, code regions like loops, conditionals, and calls cannot be automatically instrumented, only functions. TAU has also components for storing the data in a database and for visualization of the data generated during the execution by the instrumentation code.

EARL [158] is a language designed to describe event patterns of message-passing programs based on trace files. Implemented as an extension of high-level script languages (currently Tcl, Perl and Python), EARL provides abstractions that hide the trace file details. On the top of EARL, the EXPERT [157] performance tool was implemented. EXPERT is an automatic event-trace analysis tool for MPI and OpenMP applications that searches the traces for execution patterns indicating low performance and quantifies them according to their severity. Together with OPARI [104], a source-to-source translation tool for instrumenting OpenMP applications written in Fortran, C, or C++, EARL and EXPERT make up KOJAK [78], a tool set for automatic performance analysis of parallel programs.

SCALEA [147] is a tool for instrumentation, monitoring and post-mortem performance analysis of parallel programs written either in HPF or in Fortran using OpenMP and MPI. SCALEA has a profiling library whose functions can be automatically inserted by the instrumentation system in the source code or manually added by the user. The fine-grained instrumentation system allows one to instrument not only functions but also several other code regions, like loops, function calls, and I/O operations. SCALEA contains a visualization tool, interfaces to database systems (to store the profiling data generated during the execution) and also supports multiple experiment performance analysis, which allows to compare and to evaluate the performance outcome of several experiments.

Autopilot [117] is a software infrastructure that can be used to build adaptive parallel and distributed software. It provides a set of C++ classes to create sensors (pieces of software inserted in the application to monitor its behavior), actuators (components that change the application's behavior), and decision procedures (rules that govern the changes ordered by the actuators based on the data read from the sensors).

The members of the APART Esprit IV working group have formalized the knowledge that developers acquired by analyzing the performance of their applications over the years in order to concisely express the knowledge that until then had only been hardcoded in performance analysis tools. APART designed ASL (Apart Specification Language [40]) to specify experiment-related data and performance properties. ASL uses an object-oriented model to describe experiment-related data, and functions and constraints to specify performance properties. ASL has been extended several times, which resulted in a rather complex language, being now unclear how efficient it can be implemented, which has yet to be done. The ideas and the knowledge present in ASL, however, have been used in some tools mentioned in this chapter (Kappa-pi, KOJAK).

The use of artificial intelligence techniques to find performance problems is also registered in the literature. Vetter [150] applied decision trees [115] to find problems in message-passing systems. He used trace files from a set of small programs with examples of efficient and inefficient MPI behavior to train the decision tree for a specific hardware and software environment. Later, real programs were classified using the rules recorded in the tree, and problems found were mapped back to the source code.

Many of the tools mentioned above (SCALEA, Paradyn, Kappa-pi, EXPERT, Finesse) use the concept of overhead classification, discussed in Section 2.3.4, to organize the output of the performance analysis. Paradyn also uses overhead classification to conduct a systematic performance analysis, although the classification used is smaller than the one used in our work. Tools based on trace file analysis, such Kappa-pi and EXPERT, carry out the performance analysis also in a systematic way using databases that describe performance problems. EXPERT can also interpret the performance data by weighting the problems found using the application's total CPU time.

With the exception of SCALEA, all these tools concentrate on the analysis of a single experiment, while our work also focuses on multi-experiment analysis. In addition, our work is the only to use reinforcement learning to conduct the analysis process.

3.3. Tools for Java

Commercial and research tools for performance analysis of Java applications have been built since Java was released a decade ago. Although many of the ideas and principles used in Java are not new, its openness, popularity, and ubiquity, as well as the speed at which it has been evolving, contributed to create a prolific research environment in the field of performance analysis.

JVMPI (Java Virtual Machine Profiler Interface) was introduced with Java 1.1 "intended for tools vendors to develop profilers that work in conjunction with Sun's Java virtual machine implementation." [71]. This interface was always considered experimental and, as its definition indicates, no implementation of the Java virtual machine other than Sun's had to support it (although many did). A profiler agent, written in C or C++, could use this interface to communicate with the Java virtual machine, issuing controlling requests and retrieving information about the program in execution and the virtual machine; the agent could also register to be notified when certain events happened, like thread creation, method entries and exits. Since JVMPI had some stability problems and impacted on the performance of the virtual machine [102], it was deprecated as of J2SE 5.0 (Java 1.5) and substituted by JVMTI, the Java Virtual Machine Tool Interface [73]. JVMPI is scheduled to be removed from the next major release of Java (Java SE 6).

JVMTI has the same principles of JVMPI, but a richer and more elaborated interface, without incurring the same performance penalty that JVMPI did. The set of events for which an agent can register is larger, and JVMTI also allows the dynamic change of the bytecodes of a method, which could be done before only by using JVMDI, the (now deprecated) Java Virtual Machine Debug Interface [70].

In Java 1.5, the Java API was also improved, providing a subset of the functionality found in JVMTI. The monitoring and management API provides, for example, information about the amount of memory used, and also timers for obtaining:

- per thread CPU time and user time (with nanoseconds precision but not necessarily nanoseconds accuracy);
- time each thread spent trying to enter a critical section or waiting to be notified by another thread;
- time spent in garbage collection;
- time spent in JIT compilation.

In addition, the instrumentation API introduced in Java 1.5 allows one to change the bytecodes of classes dynamically and to intercept classes before they are loaded, which makes it possible to create sophisticated instrumentation and monitoring agents written purely in Java (and therefore platform independent).

The current generation of tools does not make use of these new features yet. JPMT [53] generates trace files of the execution of Java applications, being able to monitor creation and destructions of threads, object allocation, garbage collection, and synchronization operations. Similar to TAU (described above), the tool uses JVMPI and a C++ library to monitor the application, but it can also rewrite the bytecodes dynamically in order to avoid the overhead of JVMPI, which generates events for all method entries and exits. Process scheduling information can also be traced, but only in Linux, using the Linux Trace Toolkit. The user must provide a configuration file informing the events, methods, and threads to be traced.

JIS [16, 52] is an instrumentation tool that generates trace files containing important thread events, which can later be displayed using Paraver [80]. JIS originally captured calls to specific functions used in the Java virtual machine in order to probe them; later, it has been updated to use JVMPI. Since JIS records scheduling information, which only the operating system can provide, it cannot run on any platform.

Intel® VTune™ [31] is a profiling tool for Intel machines (IA-32 and Itanium) also based on JVMPI. VTune constructs a call graph of the application using the notification mechanism of JVMPI (an event is generated at each method call) and, by using sampling techniques, determines the path in the call graph that consumes the most time in the application.

Paradyn-J [101] was a prototype of a tool to analyze Java applications and their interactions with the Java virtual machine. The tool worked by intercepting the Just-In-Time (JIT) compiler of the virtual machine, so that pre-compiled versions of methods were loaded instead of the dynamically compiled version generated by the virtual machine (that is, only the methods for which the authors wrote a pre-compiled version could be monitored.) Paradyn-J was based on Paradyn (described above), but was discontinued because it depended on a specific implementation of the Java virtual machine and would need to be changed for every new virtual machine released.

Veneer [161] is a “virtual Java virtual machine” that runs on the top of any Java virtual machine and acts as a controlled environment where Java applications can run. In particular, Veneer allows the dynamic instrumentation of methods, a feature used in JUDI [161] to measure execution times of code regions. The overhead of Veneer is still high, but both tools are still in early development stages.

ProbeMeister [110] is an instrumentation tool capable of deploying probes into remotely running Java applications; it uses JDI, the Java Debug Interface [68] to connect to remote virtual machines and dynamically insert or remove instrumentation code. As JDI is not part of the Java Application Programming Interface, one needs to have it available on each Java virtual machine where the application runs.

3.4. Other Approaches for Performance Analysis

The difficulty in managing the data generated by monitoring tools for hundreds or thousands of processes and threads, the high cost of modifying the code of an already implemented program, and the need of predicting the outcome of an application execution in an environment that is for some reason unavailable motivated the development of performance estimators and application simulators.

P^{3}T [37] is an interactive performance estimator for regular scientific Fortran programs. P^{3}T uses a sequential profile run of an instrumented version of the application to analyze so as to obtain values for program unknowns (like average loop counts). These values, together with a program representation generated by a Fortran front end, are used to estimate the influence of several parameters in the application outcome for other machine sizes: load imbalance, number of transfer (send/receive) operations, amount of data transferred, network contention, transfer time, computation time, and number of cache misses.

Clement and Quinn [18] built an analytical model targeted at Dataparallel C [54] applications. Using an instrumented run of the application to be modeled and specifications of the architecture where the application will run, a symbolic equation for the application execution time, having as coefficients several system parameters, is generated as a linear function of machine and problem sizes. The system parameters (cache, page faults, latency for interprocessor communications and bandwidth characteristics for different communication patterns) are estimated using multiple linear regression techniques such as linear least-square error. Like in P^{3}T, the authors acknowledged that communication patterns are more recognizable and predictable in data parallel languages. Nevertheless, such languages could not deliver the performance achieved with, for instance, MPI; they were difficult to optimize, which is one of the main reasons that explain why HPF failed to gain popularity among programmers [25, 57, 81].

PerformanceProphet [112] can be used to model and simulate parallel applications that make use of explicit message-passing and shared-memory constructs (like those found in OpenMP and MPI, which are de facto standards today). The two main components of PerformanceProphet are Teuta, an UML-based modeling tool, and Performance Estimator, responsible for the simulation of the application using discrete events. Each code region represented in the model has a cost function associated, which provides its estimated execution time. The cost function is defined by the user, for instance, by running an instrumented version of the application with different sets of input data and then using some statistical technique to create the function. Teuta can output the UML model as C++ classes, which is the format the Performance Estimator accepts as input.

Scal-Tool [134] uses measurements of hardware event counters obtained from application executions on DSM machines for different number of processors. The measurements are used to model, with relative low cost, the effect of insufficient caching space, synchronization, and load imbalance on the scalability of the application. The model can be used not only to support the programmer in the early stages of the application development, but also to estimate the impact of using faster or slower secondary caches, interconnection networks, and synchronization support, or the impact of increasing the size of secondary cache sizes.

Vetter and Worley [153] propose the use of performance assertions, a source code annotation system that allows the user to specify a performance expectation for a given code region. For example, the user may specify that the number of instructions per cycle in a loop must be greater than 40% of the peak value that can be achieved on the machine where the program runs. The system can be used for raising performance exceptions (if an assertion fails) and validate performance models. As usual, assertions can be disabled when the user knows that all performance assertions are true.

4

Instrumentation Interfaces

Many performance tools rely on performance information (overheads, trace files, etc.) commonly obtained by statically instrumenting the source code or dynamically instrumenting an executable. In the best case, developers of performance analysis tools need to integrate their tools with instrumentation engines for different languages and platforms; in the worst case, they need to build themselves separate instrumentation engines, a tedious and time consuming effort.

We propose a **S**tandardized **I**ntermediate **R**epresentation as an interface between instrumentation engines and higher-level performance tools. The SIR is intended to be an abstract representation for procedural and object-oriented programs. Basically, a SIR contains information about statements and directive types (e.g. OpenMP) with very little details on their structures, being simpler than intermediate languages like WHIRL (used in the Open64 compiler suite [105]). This simplicity, which oriented the design of SIR, was based on the fact that higher-level performance tools commonly only have two requirements: they need to know the type of a statement in order to make a decision about specific instrumentation requests or performance analysis, and they need eventually to map the problems found back to the source code. SIR would not suitable, for instance, for a tool trying to determine the best register allocation for a program.

We also propose a set of **M**onitoring and **I**nstrumentation **R**equests and responses for the communication with instrumentation engines. This set, which we call MIR, comprehends the needs of performance tools and the *ideas* behind the current instrumentation techniques without specifying the techniques themselves. This idea has also been used in OMIS [86], although MIR has a strong link with the code being instrumented and monitored, which OMIS does not keep.

Based on the idea of SIR and MIR, higher-level performance tools can request the generation of the program representation for arbitrary programs. The performance tool can then traverse this representation and request the instrumentation of specific code regions. Important is that the generation of a SIR based on a specific input program as well as the actual instrumentation is done by an external tool. On the basis of SIR and MIR, higher-level performance tools are provided with a high-level and portable instrumentation/profiling/monitoring interface for a broad variety of programming languages without dealing with low-level details such as instrumentation, tracing, etc.

In the following sections we describe and exemplify the format of SIR for Fortran 95, Java, C, and C++ programs; we also describe the format of monitoring and instrumentation requests and responses. As extensive support already exists to traverse XML documents, we chose to define the SIR and MIR using XML.

4.1. SIR Description

A SIR is an XML document representing a Fortran 95, Java, C or C++ program (referred to simply as *input program* in the rest of this chapter). A valid SIR may contain several types of elements, the most important of which are *sir*, *unit*, and *codeRegion*. All the elements are described in detail in this section, which also gives the markup declarations (element type declarations and attribute-list declarations) that must compose the DTD describing the syntax of SIRs (the full DTDs can be found in Appendix B).

A tool that generates SIRs does not need to represent all elements described in this specification in order to be SIR compliant; nevertheless, the tool must document that, if the generated SIR does not contain a certain element or attribute, this is not because the element is absent in the input program, but because the tool chose to ignore it.

4.1.1. The Element sir

The root of any SIR is given by a *sir* element. A *sir* element must specify:

- the "main" language the input program is written in (for instance, if a Java program uses native C functions, the language must be Java, not C);
- at least one group (for instance, a class; see Section 4.1.2) or a program unit (for instance, a function; see Section 4.1.3).

Moreover, if it is known that the program communicates with other programs (processes or threads) by sending and receiving messages, the *messagePassing* attribute may be specified with the value *true* (default: *false*); similarly, if communication is done (also) through shared memory, the *sharedMemory* attribute may be specified with the value *true* (default: *false*).

The markup declarations representing these requirements are given below:

```
<!ELEMENT sir (variable*, (group|unit)+)>
<!ATTLIST sir
   language (fortran|java|c|cpp) #REQUIRED
   messagePassing (true|false) #IMPLIED
   sharedMemory (true|false) #IMPLIED>
```

4.1.2. The Elements group and inheritance

The *group* element is used to represent an organizational, non-executable unit:

- modules in Fortran;
- packages, classes, interfaces, array types, and *enums* in Java;
- namespaces and classes, as well as *structs* and unions that define methods, in C++.

Any *group* element must specify:

- the type of the group it represents (with the attribute *type*);
- a unique identifier (with the attribute *id*).

The type used for Java arrays and enums and for C++ structs and unions is *class*, while the type used for C++ namespaces is *package*. A *group* element may also specify the name of the group it represents (using the attribute *name*) and the internal name the compiler assigned to the represented group (with the attribute *internal*). In

C++, aliases of namespaces are ignored, as well as any alias for a class, struct or union name created with *typedef*. This rule holds also for struct and union names in C. A *group* element contains also zero or more *group* elements and zero or more *unit* elements. The declaration of variables (or fields) is represented with the element *variable*, described in Section 4.1.5.

When representing a class or interface, a *group* element may specify superclasses and superinterfaces using the element *inheritance*; this element accepts either the identifier or the name of a superclass or superinterface (with the attributes *id* and *name* respectively). The name must be used when the identifier is not available, since determining such an identifier may not be trivial.

Finally, a *group* may contain a *location* element, to provide where the group has been declared (see Section 4.1.6). If the URI of a location in a *group* element is left unspecified (or the entire location element), and if the immediate element containing this *group* element is either a *group* element representing a class or interface, or a *unit* element representing a method, one must assume that the URI of both elements (container and nested) is the same.

The requirements for a *group* element in a DTD are given below:

```
<!ELEMENT group (inheritance*, location?, variable*, (group|unit)*)>
<!ATTLIST group
   id ID #REQUIRED
   type (module|package|class|interface) #REQUIRED
   name CDATA #IMPLIED
   internal CDATA #IMPLIED>
<!ELEMENT inheritance EMPTY>
<!ATTLIST inheritance
   id IDREF #IMPLIED
   name CDATA #IMPLIED>
```

The language of the input program imposes certain additional restrictions; one should assume that these restrictions are also respected in the *group* elements of any SIR (although this is not enforced):

- a *group* element representing a Fortran module may not contain any *group* element;
- a *group* element for C++ may be nested only in a *group* element representing a namespace;
- in a *group* element representing a Java class, the *inheritance* element must always be specified (except if the represented class is *java.lang.Object*);
- the *name* element is never specified in a *group* element representing a Java anonymous class (the *internal* element, however, may be);
- the name of a Java class or interface must *not* be fully specified (that is, it must *not* contain package names), as the full name can be always derived from the SIR structure. In particular, nested classes must not contain the name of the class they are nested within.

4.1.3. The Elements unit and alias

The *unit* element is used to represent:

- functions, subroutines and the main program in Fortran;
- methods in Java and C++;
- functions in C and C++.

Any *unit* element must specify:

- the type of the unit it represents;
- a unique identifier.

A *unit* element specifies, through the attribute *name*, the name of the unit it represents. The *unit* element may also specify the language of the unit represented (attribute *language*), which is useful when representing C methods linked to Java or Fortran programs. It may also specify an internal, compiler-assigned name for the unit it represents (attribute *internal*).

Furthermore, a *unit* element must specify the attribute *instrumentable* with the value *false* (the default is *true*) if the tool that generates the SIR knows that the unit cannot be later instrumented (e.g. it is a library function, but the instrumentation tool can only instrument the source code). Finally, the attribute *virtual* must appear with the value *false* (the default is *true*) if, and only if, one of the following conditions is true:

- the *unit* element represents a Java method declared as private;
- the *unit* element represents a C++ method not declared as virtual.

Nested in a *unit* element there are zero or more *unit* elements, zero or more *group* elements, and zero or more *codeRegion* elements. Similar to *group* elements, a *unit* element may also contain a *location* element, to provide the location where the unit has been declared (see Section 4.1.6). If the URI of a location in a *unit* element is left unspecified (or the entire location element), and the immediate element containing this *unit* element is either a *group* element representing a class or interface, or a *unit* element representing a method with the same language attribute, one must assume that the URI of both elements (container and nested) is the same.

The declaration and use of variables are represented with the elements *variable* and *variableRef*, described in Section 4.1.5. When representing a method, function or subroutine, a *unit* element must specify the method (or function, or subroutine) signature by specifying the attribute *arguments* and providing the identifiers of variables, as described also in Section 4.1.5.

When representing a Fortran function or subroutine, a *unit* element may specify a name under which the function or subroutine may also be called using the *alias* element.

Note that the fact a function or method is *inline* is ignored.

The syntactic requirements for a *unit* element in a DTD are given below:

```
<!ELEMENT unit    (alias?, location?, variable*, variableRef*,
                   (group|unit|codeRegion)*)>
<!ATTLIST unit
   id ID #REQUIRED
   type (function|subroutine|program|method) #REQUIRED
   name CDATA #IMPLIED
   arguments IDREFS #IMPLIED
   virtual (true|false) #IMPLIED
   internal CDATA #IMPLIED
   language (fortran|java|c|cpp) #IMPLIED
   instrumentable (true|false) #IMPLIED
>
<!ELEMENT alias (#PCDATA)>
```

The language of the input program imposes certain additional restrictions; one should assume that these restrictions are also respected in the *unit* elements of any SIR (although this is not enforced):

- only a *unit* element representing a Fortran function or subroutine may be nested within a *unit* element representing a Fortran subroutine, function, or main program;
- the nesting level for *unit* elements in a *sir* element representing a Fortran program is at most 2;
- for Java and C++ programs, the name used in the *unit* element representing a constructor must be the same name used in the *group* element representing the class where the constructor has been declared;
- for Java programs, the name used in the *unit* element representing a class or interface initializer must be *&lt;clinit&gt;*. The “correct” name would be <*clinit*>, but the characters < and > may not be used in an element’s attribute;
- for Java programs, *&lt;init&gt;* must be the name used in the *unit* element representing an instance initializer;
- a *group* element may be nested within a *unit* element only if the first represents a Java class and the second a Java method (but even if a class is declared inside a method, it may be represented simply nested within the class the method is member of);
- only *unit* elements representing Java classes, Java methods and C functions may be nested within a *group* element representing a Java class;
- a *unit* element may not be nested within another *unit* element if the *sir* element represents a C or C++ program.

4.1.4. The Elements codeRegion, callee, expression, loopControl, lower, upper, stride, and scheduling

A *codeRegion* element is used to represent a sequence of specific executable program statements and directives in a *unit*.

Any *codeRegion* element must specify:

- the type of the program statement it represents in the input program (element *type*);
- a unique identifier (element *id*).

A *codeRegion* element contains zero or more nested *codeRegion* elements and zero or more nested *group* elements (in Java, it is allowed that a class is declared inside a method). It may also contain:

- a *location* element to provide the location of the represented program statement (see Section 4.1.6);
- a *callee* element, giving the identifier or the name of a method invoked or a function or subroutine called (details under the item *call*, below);
- an *expression* element, giving information about an expression (or expressions) evaluated before the represent code region executes (more details below);
- a *loopControl* element, giving information about the start, stop, and increment expression (or expressions) evaluated by certain kinds of loop constructs (details under the items *loop* and *forall*, below);

- elements *variable* and *variableRef* to represent the declaration and use of variables (described in Section 4.1.5).

Because of the complexity and diversity of the different languages and programming models, we do not intend to define a fixed set of allowed types that a *codeRegion* element must follow. In fact, different instrumentors may have their own favorites on what types of code regions are distinguished. However, in the following, we do provide a predefined set of types based on our experiences, which should be regarded only as a recommendation rather than a full specification. In the rest of this section, a *codeRegion* element with a certain type *x* will be called an x*CodeRegion* element for brevity.

- assignment

Corresponds to an explicit scalar assignment in the input program, that is, using the operator = and, in the case of Java, C and C++, also the operators ++, --, +=, * =, and so on (see the type *vector* below for vector assignments).

Multiple assignments in a single statement (like *failed = (file = openFile()) == NULL*) should be represented by using nested *expression* elements, but may be also represented by expanding the assignments to several *assignmentCodeRegion* elements (respecting the evaluation order). See an example in Section 4.2.1.

- block

Some language constructs are composed of several blocks. For instance, the *if* construct has *then*, *elseif* (in Fortran), and *else* blocks, and the constructs *switch* and *SELECT* are composed of several blocks to be executed depending on the value of an expression. Instead of creating a new kind of block for each of these constructs, the SIR just defines the generic type *block*, which can be used in any situation. A block can also be used to represent an arbitrary sequence of statements in the input program which cannot be represented by any of the types described in this section.

- if

Corresponds to the *if* construct in Java, C, C++, and Fortran. An *ifCodeRegion* element has one or more nested *codeRegion* elements, the type of which must be *block*. The *codeRegion* elements inside the first *blockCodeRegion* element corresponds to the *if* part of the *if* construct, and the other *blockCodeRegion* elements, if present, to the *else* or *else if* part. Each *blockCodeRegion* element (except the one corresponding to the "else" part of the *if* construct) may contain also an *expression* element representing the constructs that are evaluated in the respective condition (see an example in Section 4.2.3). The expression evaluated in the *if* part of an *if* construct may be also represented as a *codeRegion* element immediately before the *codeRegion* element representing the *if* construct; the use of a nested *expression* element is preferred, though.

Note: *else if* is allowed only in Fortran; therefore, when representing a Java, C or C++ program, at most two blocks are allowed nested within an *ifCodeRegion* element.

- switch

Corresponds to the *switch* construct in Java, C or C++, and to the *SELECT* construct in Fortran. The *codeRegion* elements nested within a *switchCodeRegion* element must have the type *block*; each of them corresponds to a "case" (including the "default case") of the *switch* or *SELECT* construct. The presence or absence of an implicit jump after each "case" must be inferred from the language attribute of the *unit* or *sir* element containing the *switchCodeRegion* element. Moreover, a *switchCodeRegion* element may have one *expression* element representing the condition evaluated by the *switch* or *SELECT* construct (as in *expression* elements for *ifCodeRegion* elements). Although the use of an *expression* element is the preferred way of representing such a condition, it may also be represented by a *codeRegion* immediately before the *codeRegion* representing the *switch* or *SELECT* construct.

- loop

  Corresponds to any kind of loop in the input program: *for*, *while, do...while* in Java, C and C++; *DO, DO WHILE* (but not *FORALL*) in Fortran. A *loopCodeRegion* element may have either an *expression* element representing the stop condition evaluated by the represented loop construct (as in *expression* elements for *ifCodeRegion* elements) or a *loopControl* element representing the start, stop and increment expressions in these three kinds of loop constructs: *for* (Java, C and C++), *DO* (Fortran). An example is shown in Section 4.2.4.

- jump

  Corresponds to an unconditional jump in the input program (*break*, *continue*, and *return* in Java, C and C++, *throw* in Java and C++, *goto* in C and C++, and *GO TO, CYCLE, EXIT*, and *RETURN* in Fortran). Note that a call to the function *longjmp* is not considered a jump. If the return construct being represented returns a value computed from an expression declared in front of the return statement, this expression should be represented in an *expression* element nested within the *jumpCodeRegion*. Alternatively, it may also be represented as one or more *codeRegion* elements immediately before the *jumpCodeRegion* element representing the *return* construct. The same is valid for the representation of throw constructs, that is, the expression computing the object to be thrown may be represented either as a nested element within the *jumpCodeRegion* or as one or more *codeRegion* elements immediately before it.

- call

  In Fortran, corresponds to a function or subroutine call or to a statement for dynamic storage allocation or deallocation (ALLOCATE, DEALLOCATE, and NULLIFY). In C, it corresponds to a function call, and in C++ to a function call, dynamic storage allocation and deallocation (*new* and *delete*) or a method invocation. In Java, it corresponds to a method invocation or dynamic storage allocation (*new*).

  The creation of class instances, which usually includes dynamic allocation *and* a method (constructor) invocation, must be represented as a single *callCodeRegion,* as if only the constructor were invoked.

  Nested within a *callCodeRegion* element there must be one *callee* element giving either the identifier of the function, subroutine or method invoked (or “supposed” to be invoked in the case of virtual methods) or, if the identifier is not available, the name of the invoked unit. In C++, when allocating or deallocating memory for a type that cannot be represented as a *unit* in the SIR (like *int*), the callee will be *new* or *delete*, respectively. An *expression* element may also appear nested within a *callCodeRegion* element, indicating assignments and other function calls or method invocations that are performed before the represented call is executed, the results of which will be used as arguments of the call or invocation. Alternatively (but not preferably) these assignments and calls (or invocations) may be represented as *codeRegion* elements appearing immediately before the *codeRegion* representing a call or invocation.

In indirect calls (for instance, with function pointers), only the signature of the method invoked must be specified for the callee. See an example in Section 4.2.6.

- io (Fortran specific)

Corresponds to an IO statement in Fortran (like PRINT or OPEN). As with *callCodeRegions*, *expression* elements may appear nested within an *ioCodeRegion* to represent other function calls performed before the IO statement is executed. An example is shown in Section 4.2.8.

- try, catch (Java and C++ specific), finally (Java specific)

Correspond to the construct *try...catch...finally* in Java or *try...catch* in C++.

- where (Fortran specific)

Corresponds to the *WHERE* construct in Fortran. A *whereCodeRegion* element has one *expression* element, which represents the condition evaluated by the *WHERE* construct (in the same way the *expression* element for *ifCodeRegion* elements), and one or two nested *blockCodeRegion* elements. The *codeRegion* elements inside the first *blockCodeRegion* correspond to the *where* part in the *WHERE* construct, while the second *blockCodeRegion*, if present, corresponds to the *elsewhere* part.

- forall (Fortran specific)

Corresponds to the *FORALL* construct in Fortran. Different of *loopCodeRegions*, a *forallCodeRegion* element has one *loopControl* element for each index, representing the start, stop and increment expressions of the index. It may also contain an *expression* element representing the condition ("scalar mask") evaluated for each iteration. An example is shown in Section 4.2.5.

- vector (Fortran specific)

Corresponds to an explicit vector assignment in the input program. A *vectorCodeRegion* may contain also an *expression* element representing functions called before the assignment take place (for instance, *C = log(A)*).

OpenMP directive	*parallel...codeRegion in SIR*
PARALLEL	ParallelRegion
DO	ParallelLoop
SECTIONS	ParalleSections
SINGLE	parallelSingle
WORKSHARE	parallelWorkshare
MASTER	parallelMaster
CRITICAL	parallelCriticalSection
ATOMIC	parallelAtomic
BARRIER	parallelBarrier
FLUSH	parallelFlush
ORDERED	parallelOrdered

Table 4. Mapping from OpenMP directives to *parallel...codeRegions*

Furthermore, motivated by OpenMP [107], we also defined a set of *parallel...codeRegions*, which can, in fact, be applied to any similar shared-memory paradigm. Table 4 shows the mapping of OpenMP directives to the corresponding *parallel...codeRegions*.

- parallelRegion
  Corresponds to a code region executed by several threads in parallel.
- parallelLoop
  Corresponds to a work-sharing construct that distributes the iterations of a loop among several threads. The loop is represented by a nested *loopCodeRegion* element. A *scheduling* element may also be nested to inform the scheduling type (static, dynamic, guided, or runtime) and, if applicable, the chunk to be used. Finally, if threads that finish the work they have been assigned do not need to wait until other threads also finish their work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelSections
  Corresponds to a work-sharing construct that distributes the execution of several code regions among several threads. Nested within such a *codeRegion* element there may be only *blockCodeRegions*, each of which representing a code region that is assigned to a thread. If threads that finish the work they have been assigned do not need to wait until other threads also finish their work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelSingle
  Used to group a sequence of code regions that must be executed by only one thread; this sequence is represented by one or more *codeRegion* elements nested within the *parallelSingleCodeRegion*. If the other threads do not need to wait that the thread that executes the code regions finishes its work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelWorkshare
  Corresponds to a work-sharing construct that distributes the execution of several code regions among several threads. Nested within a *parallelWorkshareCodeRegion* there may be any number of *codeRegion* elements (of any type); the way the execution of the code regions is distributed among threads depends on the library that implements the construct.

  If threads that finish the work they have been assigned do not need to wait that other threads also finish their work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelMaster
  Used to group a sequence of code regions that must be executed by only one thread, called the master thread; this sequence is represented by one or more *codeRegion* elements nested within the *paralle-masterCodeRegion*.
- parallelCriticalSection
  Used to represent a critical section. Nested within this element there may be any number of *codeRegion* elements. A unique name may be specified (in the attribute *criticalSectionName*) to identify a set of critical sections that must be executed by only one thread at a time. Among the *parallel...CodeRegions*, this is currently the only one that may be used nested within a *sir* element representing a Java program. In

Java, however, it is not in general possible, at compile time, to determine a name to give to the *parallelCriticalSectionCodeRegion*, but the expression evaluated to compute the lock to be acquired should be represented either as an *expression* element nested within the *parallelCriticalSectionCodeRegion* (preferredway) or as one or more *codeRegion* elements immediately before the *parallelCriticalSectionCodeRegion* element representing the *synchronized* construct.

- parallelAtomic
  Used to inform that an assignment is performed atomically. Nested within a *parallelAtomicCodeRegion* element there may be only one *codeRegion* element, namely an *assignmentCodeRegion*, which must represent the atomic assignment. Atomicity achieved through library invocations (for instance, using the package *java.util.concurrent.atom*) must be represented ordinarily with a *callCodeRegion*.

- parallelBarrier
  Corresponds to a language construct that synchronizes all threads within the dynamic scope of a parallel region. Barriers used through library invocations must be represented ordinarily as a *callCodeRegion*.

- parallelFlush
  Corresponds to an explicit construct that provides consistency between a thread (the one that executes the construct) and the main memory.

- parallelOrdered
  Corresponds to a construct that ensures that a sequence of code regions “is executed in the order in which iterations would be executed in a sequential execution” of a loop [107]. Nested within a *parallelOrderedCodeRegion* there may be any number of *codeRegion* elements (of any type).

As we said, the types of the code region are language and programming model specific and cannot be fully specified by the list recommended above. However, the set of types could be the basis for a custom definition.

The requirements for a *codeRegion* element in a DTD are given below. Note that the DTD does not (and cannot) enforce semantic rules involving the *type* attribute of *codeRegion* elements, like the fact that a *callee* element may appear only immediately inside a *callCodeRegion* element.

```
<!ELEMENT codeRegion    (callee?, location?, variable*, variableRef*,
                         scheduling?, (expression|loopControl)*,
                         (codeRegion|group)*)>
<!ATTLIST codeRegion
   id ID #REQUIRED
   type CDATA #REQUIRED
   criticalSectionName CDATA #IMPLIED
   noWait (true|false) #IMPLIED
>
<!-- The recommended code region types include
(block|assign|loop|if|switch|where|jump|call|io|try|catch|finally|
parallelRegion|parallelLoop|parallelSections|parallelSingle|
parallelWorkshare|parallelMaster|parallelCriticalSection|
parallelAtomic|parallelBarrier|parallelFlush|parallelOrdered|
vector|forall)
-->
<!ELEMENT callee EMPTY>
   <!ATTLIST callee
   id IDREF #IMPLIED
```

```
  name CDATA #IMPLIED
>
<!ELEMENT expression ((codeRegion|group)+))>
<!ELEMENT loopControl (lower?,upper?,stride?)>
<!ELEMENT lower (codeRegion+)>
<!ELEMENT upper (codeRegion+)>
<!ELEMENT stride (codeRegion+)>
<!ELEMENT scheduling EMPTY>
<!ATTLIST scheduling
  type (static|dynamic|guided|runtime) #REQUIRED
  chunk CDATA #IMPLIED>
```

The order of *codeRegion* elements in the *sir*, as well as the way they are nested, reflect the syntactical order and nesting of the represented program statements in the input program the *sir* represents. For instance, if the program statement (or sequence of program statement) *A* appears in the input program before the program statement (or sequence of program statements) *B*, then the *codeRegion* element representing *A* must appear in the SIR before the *codeRegion* element representing *B.*

4.1.5. The Elements variable and variableRef

The *variable* element represents the definition of a variable (scalar or array). Each variable must have an attribute of unique *id*, and can have optional attributes like a *name*, a *type*, and *dimensions*. If *dimensions* is defined as -1 or if it is omitted, then the variable is simply a scalar. For arrays, the lower bound and upper bound of each dimension can be specified with one nested element *dimension*, while the *index* attribute indicates which dimension is being described. The type used for a *variable* element is language dependent (that is, this specification does not dictate the name under which the type of a variable must be represented), but it should be used consistently throughout the input program representation.

As usual, the *location* element informs where a variable is declared in the input program.

The DTD segment for *variable* element is given below:

```
<!ELEMENT variable (location?, dimension*)>
<!ATTLIST variable
  id ID #REQUIRED
  name CDATA #IMPLIED
  type CDATA #IMPLIED
  dimensions CDATA #IMPLIED
>
<!ELEMENT dimension EMPTY>
<!ATTLIST dimension
  index CDATA #REQUIRED
  upperBound CDATA #REQUIRED
  lowerBound CDATA #REQUIRED
>
```

As method, function and subroutine arguments are in fact variables, they are also represented as such; in addition, the attribute *arguments* of a unit will contain a list of identifiers referring to the variables that are arguments in the unit.

References to variables in each unit and code region are represented by *variableRef* elements. Each *variableRef* element must specify *targetId*, which is used to identify the variable that it references. The optional attribute *accessType* can also be provided to indicate if the variable is read, written, or both. The DTD segment for *variableRef* element is given as follows:

```
<!ELEMENT variableRef EMPTY>
<!ATTLIST variableRef
  targetId IDREF #REQUIRED
  accessType ( read | write | readwrite ) #IMPLIED
>
```

4.1.6. The Element location

A *location* element represents the location of a unit, a program statement, a variable declaration, a directive or a sequence of program statements and directives in a file. The *location* element contains attributes for representing the start line, the start column, the end line and the end column the represented code occupies in a “file” (not necessarily all of them need to appear in the element). The location of a file is given by the attribute *uri*, and does not need, in fact, to refer to a file, but to any resource. If the resource where the represented code is defined is not the same as the resource a nested unit, program statement or directive is defined, the *location* element in the nested unit or program statement must also be specified.

The requirements for a *location* element in a DTD are given below:

```
<!ELEMENT location EMPTY>
<!ATTLIST location
  startLine CDATA #IMPLIED
  startColumn CDATA #IMPLIED
  endLine CDATA #IMPLIED
  endColumn CDATA #IMPLIED
  uri CDATA #IMPLIED>
```

An example that uses the *location* element is shown in Section 4.2.8. The syntax of a *uri* attribute can be found in [116].

4.1.7. Open Issues

- Templates in C++ and Java are not represented.
- Overloaded operators in C++ are not represented, although they may represent rather complex functions.
- *firstprivate*, *lastprivate*, reduction in *OpenMP* are not represented; it is not clear if they should be.
- Extra compiler information is not represented. Sometimes, it is possible to determine through compiler analysis the real method that is going to be invoked (or a set of possible methods). The same is valid for indirect function calls. For instance:

```
if (condition) myfunction = max else myfunction = min;
x = myfunction(10, 20);
```

or

```
Shape s;
if (condition) s = new Circle(...) else s = new Square(...);
s.draw();
```

Even if the compiler has this information, it cannot be represented in the SIR.

4.2. Examples

4.2.1. Multiple Assignments

This example shows two ways of representing several assignments appearing in a single statement, as well as how variables are represented.

- C code:

```
int failed;
FILE* f;
failed = (f = fopen("file.txt", "r+")) != NULL;
```

- SIR mapping using the element *expression*:

```
<unit type="function" name="fopen" arguments= "v1 v2"
      instrumentable="false" id="u1">
  <variable type="char*" id="v1"/>
  <variable type="char*" id="v2"/>
</unit>
...
<variable type="integer" name="failed" id="v3"/>
<variable type="FILE*" name="f" id="v4"/>
<codeRegion type="assignment" id="a1"> <!-- failed = ... -->
  <variableRef targetId="v3" accessType="write"/>
  <variableRef targetId="v4" accessType="read"/>
  <expression>
    <codeRegion type="assignment" id="a2"> <!-- f = ... -->
      <variableRef targetId="v4" accessType="write"/>
      <expression>
        <codeRegion type="call" id="c1"> <!-- fopen() -->
          <callee id="u1"/>
        </codeRegion>
      </expression>
    </codeRegion>
  </expression>
</codeRegion>
```

- SIR mapping without the element *expression*:

```
<codeRegion type="call" id="c1"> <!-- fopen() -->
  <callee id="u1"/>
</codeRegion>
<codeRegion type="assignment" id="a1"> <!-- f = ... -->
  <variableRef targetId="v4" accessType="write"/>
</codeRegion>
<codeRegion type="assignment" id="a2"> <!-- failed = ... -->
  <variableRef targetId="v3" accessType="write"/>
  <variableRef targetId="v4" accessType="read"/>
</codeRegion>
```

4.2.2. Inheritance and Constructors

This illustrates the mapping of inheritance and constructors of Java classes, as well as method (in this case, constructor) invocations.

- Java code:

```
package example;
class MyClass extends java.awt.Button
            implements Runnable, java.awt.event.KeyListener {
  MyClass(String s) { super(s); }
  ...
}
```

- SIR mapping:

```
<group type="package" name="java" id="p1">
  <group type="package" name="lang" id="p2">
    <group type="interface" name="Runnable" id="i1"/>
  </group>
  <group type="package" name="awt" id="p3" instrumentable="false">
    <group type="class" name="Button" id="c1">
      <unit type="method" name="Button" arguments="v1" id="m1">
        <variable type="java.lang.String" id="v1"/>
      </unit>
    </group>
    <group type="package" name="event" id="p4">
      <group type="interface" name="KeyListener" id="i1"/>
    </group>
  </group>
</group>
<group type="package" name="example" id="p5">
  <group type="class" name="MyClass" id="c2">
    <unit type="method" name="MyClass" id="m2">
      <codeRegion type="call" id="cr1">
        <callee id="m1">
      </codeRegion>
    </unit>
    ...
  </group>
</group>
```

4.2.3. If Constructs and Functions Calls

This example illustrates the mapping of an *if* construct in C to an *ifCodeRegion*, including the use of the elements *blockCodeRegion*, *callee*, and *expression*.

- C code:

```
if (f(n) > g(m)) {
  a = 10;
} else {
  flag = false;
}
```

- SIR mapping:

```
<!-- assume that the id of f is "f" and the id of g is "g" -->
<codeRegion type="if" id="i1"> <!-- if (...) -->
  <codeRegion type="block" id="i2">
    <expression>
      <codeRegion type="call" id="i3"> <!-- f(n) -->
        <callee id="f"/>
      </codeRegion>
      <codeRegion type="call" id="i4"> <!-- g(m) -->
        <callee id="g"/>
      </codeRegion>
    </expression>
    <codeRegion type="assignment" id="i5"/> <!-- a = 10 -->
  </codeRegion>
  <codeRegion type="block" id="i6"> <!-- else -->
    <codeRegion type="assignment" id="i7"/> <!-- flag = false -->
  </codeRegion>
</codeRegion>
```

4.2.4. Loop Constructs

This example illustrates the mapping of a *for* loop in C (or C++ or Java) to a *loopCodeRegion*, including the use of the element *loopControl*.

- C/C++/Java code:

```
for(i = fg(5), j = 4; i <= gh(100) && j <= mn(8); i += hi(2), j++) {
...
}
```

- SIR mapping:

```
<!-- assume that the functions have identical ids and names -->
<codeRegion type="loop" id="i1">
  <loopControl> <!-- i -->
    <lower> <!-- fg(5) -->
      <codeRegion type="call" id="i2">
        <callee id="fg"/>
      </codeRegion>
    </lower>
    <upper> <!-- gh(100) -->
      <codeRegion type="call" id="i3">
        <callee id="gh">
      </codeRegion>
      <codeRegion type="call" id="i4">
        <callee id="mn"/>
      </codeRegion>
    </upper>
    <stride> <!-- hi(2) -->
      <codeRegion type="call" id="i5">
        <callee id="hi"/>
      </codeRegion>
    </stride>
  </loopControl>
  ...
</codeRegion>
```

4.2.5. FORALL

This example illustrates the mapping of a FORALL loop in Fortran to a *forallCodeRegion,* including the use of the element *loopControl*.

- Fortran code:

```
FORALL (i = fg(5):gh(100):hi(2), j = 4:mn(8), op(i) < op(j))
...
END FORALL
```

- SIR mapping:

```
<!-- assume that the functions have identical ids and names -->
<codeRegion type="loop" id="i1">
  <loopControl> <!-- i -->
    <lower> <!-- fg(5) -->
      <codeRegion type="call" id="i2">
        <callee id="fg"/>
      </codeRegion>
    </lower>
    <upper> <!-- gh(100) -->
    <codeRegion type="call" id="i3">
      <callee id="gh">
    </codeRegion>
    </upper>
    <stride> <!-- hi(2) -->
```

```
      <codeRegion type="call" id="i4">
        <callee id="hi"/>
      </codeRegion>
    </stride>
  </loopControl>
  <loopControl> <!-- j -->
    <upper> <!-- mn(8) -->
      <codeRegion type="call" id="i5">
        <callee id="mn"/>
      </codeRegion>
    </upper>
  </loopControl>
  <expression> <!-- op(i) < op(j) -->
    <codeRegion type="call" id="i6">
      <callee id="op"/>
    </codeRegion>
    <codeRegion type="call" id="i7">
      <callee id="op"/>
    </codeRegion>
  </expression>
  ...
</codeRegion>
```

4.2.6. Pointer Functions

This example shows the mapping of pointer functions in C.

- C code:

```
void sort(void *array, int size,
          int (*cmpfunc)(const void *, const void *)) {
  ...
  cmpfunc(a, b);
  ...
}
```

- SIR mapping:

```
<unit type="function" name="sort" arguments="v1 v2 v3" id="f1">
  <variable name="array" type="void*" id="v1"/>
  <variable name="size" type="int" id="v2"/>
  <variable name="cmpfunc"
           type="(int)(const void *, const void *)" id="v3"/>
  ...
  <codeRegion type="call" id="c2">
    <callee id="u1"/>
  </codeRegion>
  ...
</unit>
```

4.2.7. Overloaded Functions

This example shows the mapping of overloaded functions in Fortran.

- Fortran code:

```
INTERFACE PHI
  FUNCTION IPHI(X)
    INTEGER IPHI, X
  END FUNCTION IPHI
  FUNCTION RPHI(X)
    REAL RPHI, X
  END FUNCTION RPHI
END INTERFACE PHI
```

- SIR mapping:

```
! function contents not important
<unit type="function" name="IPHI" arguments="v1" id="f1">
  <variable name="X" type="INTEGER" id="v1"/>
  <alias>PHI</alias>
  ...
</unit>
<unit type="function" name="RPHI" arguments="v2" id="f2">
  <variable name="X" type="REAL" id="v2"/>
  <alias>PHI</alias>
  ...
</unit>
```

4.2.8. IO Statements and Location

This example shows a piece of Fortran code mapped to a SIR including the *location* element, and also how an IO statement is mapped to an element in the SIR.

- Fortran code:
  file F1.f90

```
column 123456789012345678901234
line 1:    SUBROUTINE f(x)
line 2:    REAL :: x
line 3:    INCLUDE "F2.f90"
1ine 4:    END SUBROUTINE f
```

  file F2.f90

```
column 12345678901234567890
line 1:    PRINT *, foo(1)
```

- SIR mapping:

```
<!-- assume that the id of PRINT is "print"
     and the id of foo is "foo"-->
<unit type="subroutine" name="f" id="i1">
  <location startLine="1" startColumn="5" endLine="4" endColumn="20"
            uri="file:///home/joe/programs/F1.f90"/>
  <codeRegion type="io" id="i2">
    <location startLine="1" startColumn="5"
              endLine="1" endColumn="19"
              uri="file:///home/joe/programs/F2.f90"/>
    <expression>
      <codeRegion type="call" id="i3">
        <location startLine="1" startColumn="14"
                  endLine="1" endColumn="19"/>
        <callee id="foo"/>
      </codeRegion>
    </expression>
    <callee id="print"/>
  </codeRegion>
</unit>
```

4.3. MIR Description

This section describes the format of several kinds of requests used to control the instrumentation and monitoring of an application, as well as the format of the responses expected from these requests. There are four types of requests that can be used:

- SIR: a request for the SIR of a set of programs in an application;
- Snapshot: a request for the current status of an application in execution;
- Instrumentation: a request for instrumenting the application;
- Control: a request for altering the instrumentation of the application or to get the value measured by the instrumentation code.

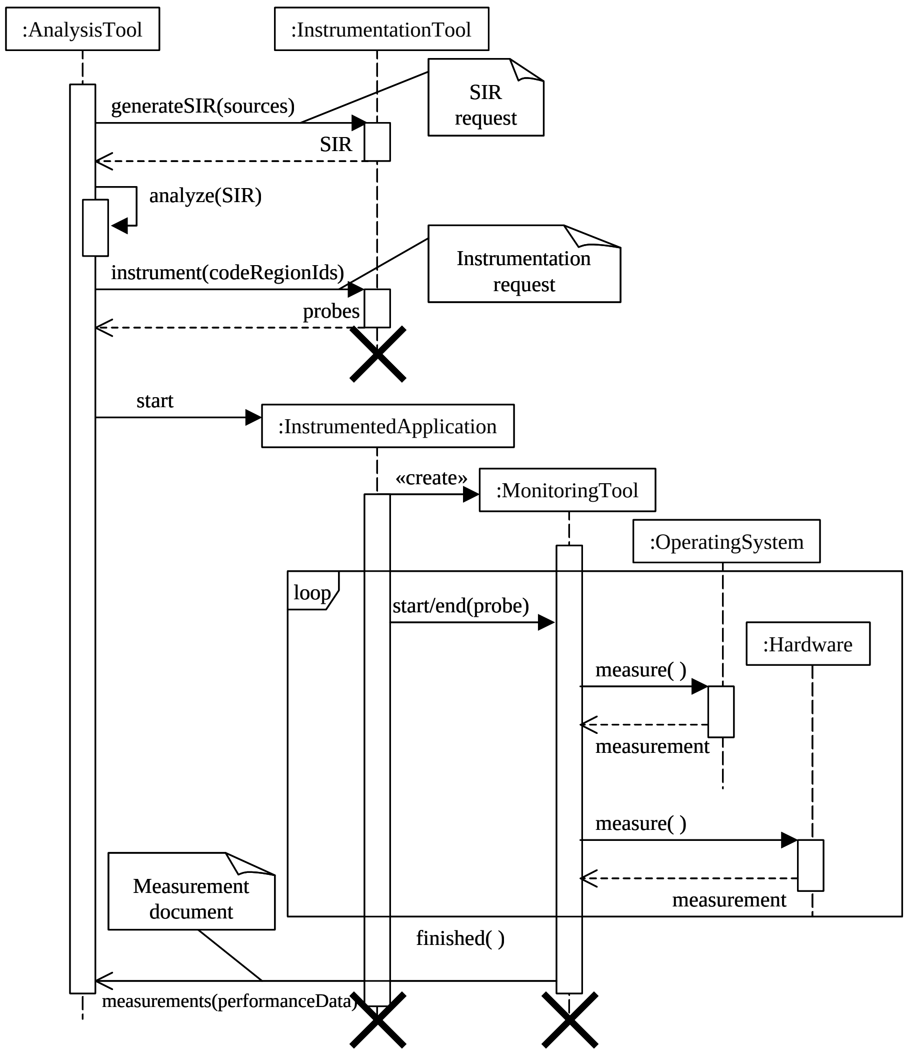


Figure 11. Interactions between tools using MIR (post-mortem analysis)

The UML Interaction Diagram in Figure 11 shows one possible interaction between tools that use MIR to communicate with each other in a post-mortem analysis scenario with static instrumentation. The Analysis Tool sends a SIR request to the Instrumentation Tool asking for the SIR of the sources to be instrumented. The SIR is generated and sent back to the Analysis Tool. By analyzing the SIR received, the Analysis Tool decides what to instrument and sends an instrumentation request to the Instrumentation Tool containing the code regions to be instrumented. The Instrumentation Tool instruments the code and returns identifiers for the probes inserted. After the instrumented sources have been compiled, the Analysis Tool starts the Instrumented Application, possibly with an embedded Monitoring Tool (e.g. a library linked in the application) that collects performance data, like values of hardware counters or operating system timers. When the Instrumented Application finishes, the performance data collected is sent back to the Analysis Tool where they can be analyzed.

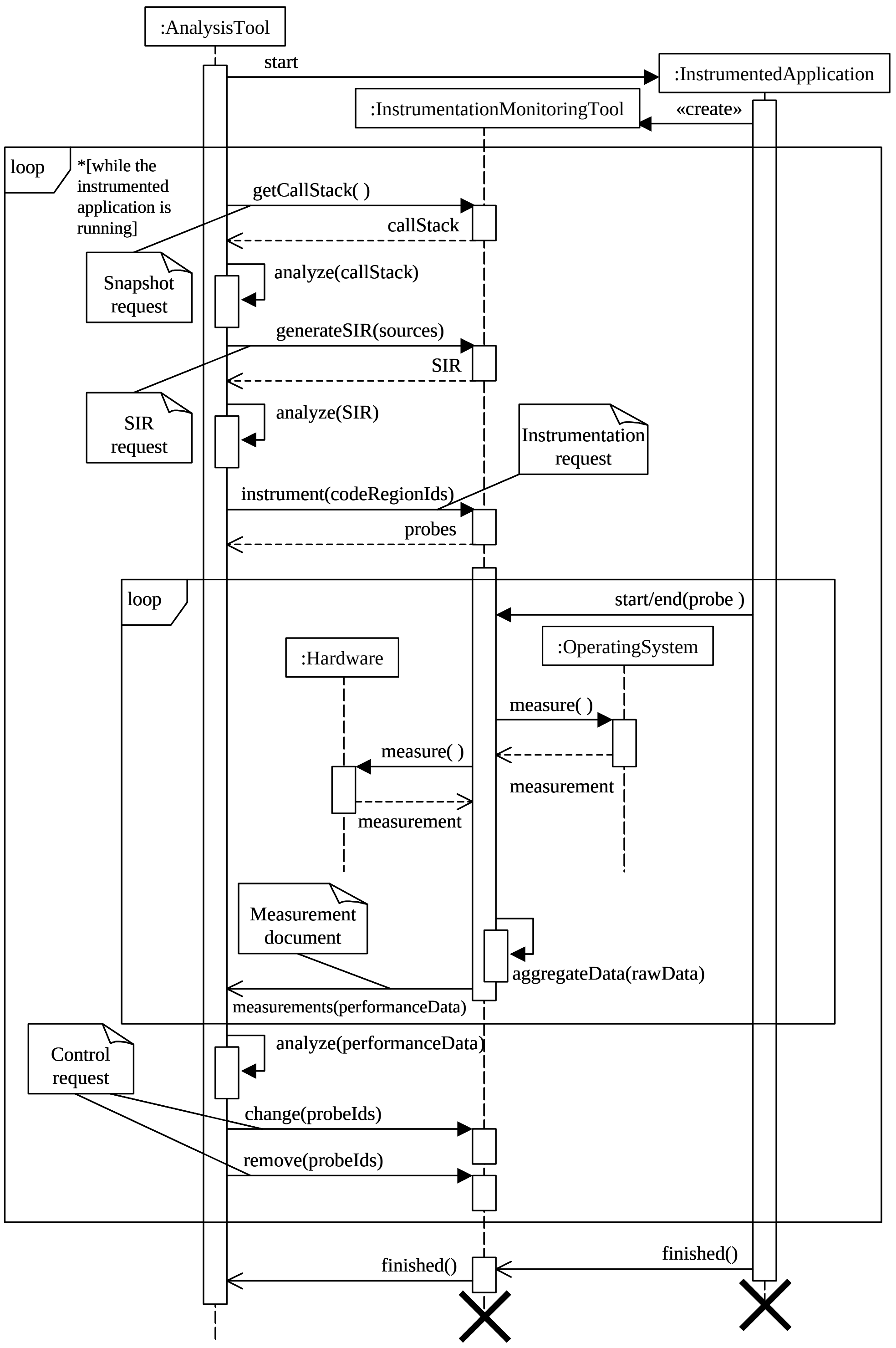


Figure 12. Interactions between tools using MIR (dynamic analysis)

Figure 12 shows another possible scenario, now with dynamic analysis and instrumentation. While the instrumented application is running, the Analysis Tool sends to the Instrumentation and Monitoring Tool 1) Snapshot requests, in order to find the subroutines that are executed, 2) SIR requests, so as to get details about the application structure, 3) Instrumentation requests, to insert probes that collect performance data about code regions, and 4) Control requests, in order to change the data that is collected or remove probes that are too intrusive or that never measure anything.

The syntactic and semantic rules of the four request types and their responses are described in the following.

4.3.1. The SIR Request

A SIR request is used in order to obtain the SIR of one or more programs that make up an application. The SIR generated can be analyzed and its code region identifiers used to instrument the application; a SIR request also specifies where the programs must be written back after the SIR has been instrumented.

SIR requests are simple; besides the root element, *sirreq*, there may be only one other kind of element, *resource*, which names the "files" (more generally speaking, resources) used to generate the SIR (attribute *in*) and where they should be written back after the instrumentation (attribute *out*).

The following DTD describes the syntax of a SIR request

```
<!ELEMENT sirreq (resource+)>
<!ELEMENT resource EMPTY>
<!ATTLIST resource
  in CDATA #REQUIRED
  out CDATA #IMPLIED>
```

The syntax of the *in* and *out* attributes is defined by RFC 2396: Uniform Resource Identifiers (URI) [116]. For instance, the following request could be used to get a program from the Web (along with a file needed to its compilation) and write it on the local disc:

```
<sirreq>
  <resource
    in="http://www.fictive.com/mmul.c"
    out="file:///home/clovis/mmul.c">
  <resource in="ftp://anonymous@fictive.com/prototypes.h">
</sirreq>
```

4.3.2. The Snapshot Request

A snapshot request is used to get information about some entities of an application in execution: sites, nodes, processes, and threads. The request itself is simple and small (it has only the root element, *snapshotreq*), while the response may contain not only the entities enumerated above, but also call stacks of the execution.

The following DTD describes the syntax of a snapshot request:

```
<!ELEMENT snapshotreq EMPTY>
<!ATTLIST snapshotreq
    site (true|false) #IMPLIED
    node (true|false) #IMPLIED
    process (true|false) #IMPLIED
    thread (true|false) #IMPLIED
  named (true|false) #IMPLIED
  stack CDATA #IMPLIED
  freeze (true|false) #IMPLIED>
```

The attributes *site*, *node*, *process*, and *thread* specify which entities must be present in the snapshot (the default value is implementation dependent). The attribute *named* specifies if the snapshot must also contain the names (if available) of the entities in the snapshot, as the default is the generation of snapshots only with identifiers for these entities. The attribute *stack* specifies if the call stack of the execution is wished, and how deep it must be. The default value for *stack* is zero, that is, no call stack. Finally, the attribute *freeze* specifies the state of the entities after the snapshot request: if true, they will be suspended and will not make any progress until another snapshot request, with the attribute *freeze* set to false, is received.

The following DTD, which describes the syntax of a snapshot, is more complex, though (the root element is snapshot):

```
<!ELEMENT snapshot (site*, node*, process*, thread*)>
<!ELEMENT site (node*|(process*, thread*))>
<!ATTLIST site
  id CDATA #REQUIRED
  name CDATA #IMPLIED>
<!ELEMENT node (process*, thread*)>
<!ATTLIST node
  id CDATA #REQUIRED
  name CDATA #IMPLIED>
<!ELEMENT process (thread*|stack*)>
<!ATTLIST process
  id CDATA #REQUIRED
  name CDATA #IMPLIED>
<!ELEMENT thread (stack*)>
<!ATTLIST thread
  id CDATA #REQUIRED
  name CDATA #IMPLIED
  master (true|false) #IMPLIED>
<!ELEMENT stack (#PCDATA)>
```

Each entity in a snapshot has a unique identifier that can be used in an instrumentation request, as shown later. The names of the entities appear if available and if the snapshot request specified the attribute *named* with value true. The attribute *master* is used to specify whether a certain thread in the snapshot is the master thread in the process, in which case it appears with the value *true* (the default value is *false*. It makes sense only for applications with multithreaded processes (e.g. using OpenMP [107], hybrid OpenMP/MPI, multithreaded MPI). Finally, each stack element describes, in an application dependent format, a stack frame. The maximum number of stack elements in a thread or process element is limited by the value of the attribute *stack* in the snapshot request.

The following example shows a snapshot request for a Java program and the snapshot received as answer:

```
<snapshotrequest named="true" stack="3">

<snapshot>
  <thread id="15" name="AWT-EventQueue-0">
    <stack>java.lang.Object.wait</stack>
    <stack>java.awt.EventQueue.getNextEvent</stack>
    <stack>java.awt.EventDispatchThread.pumpOneEventForHierarchy
    </stack>
  </thread>
  <thread id="16" name="DestroyJavaVM"/>
  <thread id="1" name="main"/>
```

```
     <stack>java.lang.Thread.join</stack>
     <stack>App.main</stack>
  </thread>
</snapshot>
```

4.3.3. The Instrumentation Request

Instrumentation requests are issued before or during the program execution so as to instrument an application. They may refer to code regions (using identifiers obtained from a SIR document) and entities like processes and threads (using identifiers obtained from a snapshot document).

The code generated through an instrumentation request is called a probe. An instrumentation request may actually generate several probes, one for each code region specified in the request. A probe has, at every instant, a value associated to it, which corresponds either to the last value measured by the probe or to the aggregation of this value and previously measured values. This value will be called here probe value.

Each probe receives also a unique identifier called probe identifier, which can be used to retrieve the probe value, as well as to alter the probe or even remove it.

The following DTD describes the syntax of an instrumentation request. The definition of the elements *site*, *node*, *process*, *thread* was omitted, as it is the same as in the snapshot request (Section 4.3.2).

```
<!ELEMENT instrreq (
    codeRegion*,
    metric*,
    event*,
    measuring?,
    site*, node*, process*, thread*)>
<!ATTLIST instrreq
    defaults (true|false) #IMPLIED
    activated (true|false) #IMPLIED
    flush (true|false) #IMPLIED>

<!ELEMENT codeRegion EMPTY>
<!ATTLIST codeRegion
    from CDATA #REQUIRED
    to CDATA #IMPLIED>

<!ELEMENT metric EMPTY>
<!ATTLIST metric
    name CDATA #REQUIRED>

<!ELEMENT event EMPTY>

<!ELEMENT measuring (aggregate*)>
<!ATTLIST measuring
    delivery    CDATA #IMPLIED
    destination CDATA #IMPLIED
    interval    CDATA #IMPLIED
    duration    CDATA #IMPLIED>

<!ELEMENT aggregate EMPTY>
<!ATTLIST aggregate
    function (AVERAGE|MAXIMUM|MINIMUM|SUM|VARIANCE) #IMPLIED
    group CDATA #IMPLIED>
<!-- group contains SITE NODE PROCESS THREAD METRIC -->
```

The root of an instrumentation request is the element *instrreq*. Inside it the following elements are allowed:

- The *codeRegion* element, with the attributes *from* and *to*. These attributes contain the identifier of a *unit* or *codeRegion* element in a SIR, and delimit the beginning and end of a region to be monitored in the input program. Some metric will be measured at the beginning and at the end of the region, and the probe value will be the difference between these two values. If the *to* or the *from* element is omitted, then the metric will be measured only at the beginning or at the end of the code region (no difference will be computed). The concept of "valid" region, however, is application dependent—one instrumentation tool may allow to define a region that begins in a function and ends in other, while another tool not. Several *codeRegion* elements may be present in a single instrumentation request.
- The *metric* element, defining which metric should be measured for the region of interest. Several metric elements may be present in a single instrumentation request. Possible values for metrics depend on specific implementations (see Section 4.3.7).
- The *event* element, indicating that event traces must be generated for the region of interest. The format of the event traces remains to be defined and is not covered in this specification.
- The *measuring* element, which defines:
  - how often, in milliseconds, measurements must be done (attribute *interval*). The default value is zero, which means that the measurements are done only when the code region defined with the element *codeRegion* is executed.
  - how often, in milliseconds, values measured are automatically delivered (attribute *delivery*). The default value is zero, which means that the values are not automatically delivered; they must be retrieved through a control request (see Section 4.3.4). The value -1 has a special meaning: the values are delivered every time they are measured.
  - where values measured must be delivered (attribute *destination*). Values are always delivered as measurement documents (see Section 4.3.5), but the default value for the destination attribute is application dependent. The format, though also application dependent, must follow the URI syntax [116].
  - how long, in milliseconds, a measurement takes (attribute *duration*). The default is zero; a non-zero value T means that the measurement must be done at instant K, then at instant K + T, and that the difference between the two values measured must become the probe value.

  A measuring element may contain aggregate elements, to indicate that measurements must be grouped according to certain statistic functions (AVERAGE, MINIMUM, MAXIMUM, SUM, and VARIANCE). If no aggregate element is specified, or if it is specified with no function, then the probe value will always be the last value measured; otherwise, it will be the value returned by one of these functions (or values, if more than one function is specified). Aggregate elements may also specify levels of grouping when computing statistics with the attribute *group*. For example, instead of having the maximum among all process, one can say that the maximum should be grouped by processes by specifying *group="PROCESS"*.
- The thread, process, node, and site elements, specifying the identifiers of threads, processes, nodes, and sites for which measurements must be taken. The format of an identifier for any of these elements must be obtained from a snapshot document

(see Section 4.3.2). Two symbols, however, have a special meaning for an identifier: the asterisk, which means “all,” and the question mark, which means “the current entity” (or “the entity doing the measurement”). For instance, *<process id="*">* means that the measurement must be taken for all processes related to the application, while *<process id="?">* means that the metric must be measured only in one process, namely the one that is doing the measurement itself. Question marks are useful, for instance, when taking measurements for a code region. In fact, if an instrumentation request specifies a code region but no entity, the question mark is assumed as “identifier” for the elements thread, process, node, and site.

Note that there is a difference between a request that specifies

```
<process id="P1"/>
```

and one that specifies

```
<process id="P1">
  <thread="*">
</process>
```

The first request asks for one single value, measured for the process *P1*, while the second request asks for several values, one for each thread of process *P1*.

An instrumentation request may also have three attributes: *flush*, *activate,* and *defaults*. When the *flush* attribute has the value *true*, the current instrumentation request is flushed, together with all the previous instrumentation requests where the attribute *flush* was *false* or absent. One particular consequence is that only now SIRs may be parsed back to source code (now also with instrumentation code). The attribute activate, if specified with the value *true*, indicates that the measurements must start as soon as the instrumentation is flushed. When the value is false (or the attribute is absent), the probe starts inactive, and a control request (see Section 4.3.4) will be needed to activate it. The attribute *defaults*, if specified with the value *true*, indicates that the request should not create probes, but only set default values for the other elements and attributes, which can potentially make the next requests shorter. When the value is *false* (or the attribute is absent), one or more probes will be created.

The response to an instrumentation request is a *probe* document, the syntax of which is defined as follows:

```
<!ELEMENT probes (probe+)>

<!ELEMENT probe EMPTY>
<!ATTLIST probe
    id CDATA #REQUIRED>
```

Each nested probe element represents a probe inserted; if more than one code region was specified in the instrumentation request, then there will be more than one probe element, one for each probe created. The attribute *id* of a probe identifies the probe inserted; it may be used later in a control request and also to identify a value in a measure document (see Section 4.3.5).

If the instrumentation request does not generate a probe (because it is just setting default values) the response will be simply *<ok>*.

The following example shows how to measure the number of bytes sent and received in the network for threads 1045 and 1032 blocked to enter in a critical section. This value is measured every time any thread executes the code region with identifier *c1*, and the maximum of the values measured is sent every second to the file */tmp/foo.txt*.

```
<instrreq>
  <codeRegion from="c1"/>
  <metric name="NET_SEND"/>
  <metric name="NET_RECV"/>
  <measuring
    delivery="1000"
    destination="file:///tmp/foo.txt"
    <aggregate function="MAXIMUM"/>
  </measuring>
  <thread="1045"/>
  <thread="1032"/>
</instrreq>
```

If, instead of *<aggregate function="MAXIMUM"/>*, we had used *<aggregate function="MAXIMUM" group=”METRIC”/>*, we would have created a level of grouping for the statistics, and two values would be sent every second: the maximum of number of bytes sent (metric NET_SEND) and the maximum of number of bytes received (metric NET_RECV). If we had used *<aggregate function="MAXIMUM" group=”METRIC THREAD”/>* we would have created two levels of grouping, and four values would be sent every second: the maximum of number of bytes sent for thread 1045, the maximums of bytes sent for thread 1032, the maximum of number of bytes received for thread 1045, and the maximums of bytes received for thread 1032.

If we had used *<thread id=”?”>* instead of the thread identifiers, the metrics NET_SEND and NET_RECV would be measured for each thread, but the measurements for any thread *T* would be taken only when *T* executed the code region *c1*. By using *<thread id=”*”>*, however, the measurements would be taken for all threads every time any of them executed the code region *c1*.

4.3.4. The Control Request

A control request is used to access the probe created through an instrumentation request. With control requests and the probe identifiers returned by the instrumentation requests (see Section 4.3.3) it is possible to change the instrumentation or retrieve the values it measures.

The root of a control request is the *ctrlreq* element. The DTD giving the syntax of control requests is given below:

```
<!ELEMENT ctrlreq (
  probe+,
  metric*,
  measuring?,
  site*, node*, process*, thread*)>
<!ATTLIST ctrlreq
  flush (true|false) #IMPLIED
  action (VALUE|ACTIVATE|DEACTIVATE|RESET|REMOVE) #REQUIRED>
```

- The element *probe* specifies, with the attribute *id*, the identifier returned by a previous instrumentation request. Several probe elements may be specified in the same request.

- The attribute *action* defines the effect of this control request on the probe(s). Possible actions are:
  - VALUE: The last value measured (or the last aggregation) is returned as a measurement document (see Section 4.3.5); the instrumentation does not change.
  - ACTIVATE: The measurements start to be taken for the specified probe(s). If they already were active, nothing happens.
  - DEACTIVATE: The measurements stop to be taken for the specified probe(s). The perturbation generated by the probe(s) should be minimal.
  - RESET: Resets the aggregations associated with the specified probe(s). All the measurements taken for that probe(s) until the moment of this request will be forgotten, as if the probe had just been inserted.
  - REMOVE: Invalidates the specified probe, possibly removing the instrumentation.

The attribute *flush*, as well as the elements *metric*, *measuring*, *site*, *node*, *process*, and *thread* are equivalent to their counterparts in an instrumentation request. If left unspecified, the previous settings associated with the specified probe(s) remain unaltered.

The following example removes the probes *p1* and *p2*:

```
<ctrlreq>
  <probe id="p1"/>
  <probe id="p2"/>
  <action type="REMOVE"/>
</ctrlreq>
```

This example returns the last value measured for probe *p2*:

```
<ctrlreq>
  <probe id="p2"/>
  <action type="VALUE">
</ctrlreq>
```

Finally this example changes the probe *p3* to measure every 4 seconds the time that spent sending messages in node *mynode.ac.at*:

```
<ctrlreq>
  <probe id="p3"/>
  <action type="RESET"/>
  <metric name="NET_SEND"/>
    <!-- NET_SEND means time spent sending messages -->
  <measuring interval="4000"/>
  <node id="mynode.ic.at"/>
</ctrlreq>
```

4.3.5. The Measurement Document

The response to a control request whose action has the type VALUE, as well as the document sent automatically if the probe is associated to a delivery interval different from zero, is a measurement document, the syntax of which is defined as follows:

```
<!ELEMENT measurement (measurement)*>
<!ATTLIST measurement
  probeId CDATA #IMPLIED
  siteId CDATA #IMPLIED
  nodeId CDATA #IMPLIED
  processId CDATA #IMPLIED
  threadId CDATA #IMPLIED
  value CDATA #IMPLIED>
```

A measurement document is generated from a set of tuples (*probeId*, *siteId*, *nodeId*, *processId*, *threadId*, *metric*, *value*), where *null* is also a possible value for *siteId*, *nodeId*, *processId*, and *threadId* (in a pure sequential program, for instance, all of them can be *null*). Each tuple corresponds either to the value measured for some metric or to an aggregation of values (average, maximum, minimum, sum, variance); for the aggregations average, sum, and variance, *siteId*, *nodeId*, *processId* and *threadId* will always be *null*.

In order to generate the document in a compact form, the following algorithm is applied (where we call *last defining request* the instrumentation request that created a probe or the control request that last modified it):

1. Generation: For each tuple, a measurement element is generated using the values in the tuple as the values of the respective attributes in the element. Note that there is no attribute for metric.

2. Sorting:

- If two measurement elements *m1* and *m2* have the same value for the attribute *probeId* but different values for the attribute *siteId*, then *m1* must appear in the document before *m2* if the site in *m1* was neither an asterisk nor a question mark and it has been specified before the site in *m2* in the last defining request of the corresponding probe. A similar rule is applied if two measurement elements have:
  - the same value for the attributes *probeId* and *siteId* but different values for the attribute *nodeId*;
  - the same value for the attributes *probeId*, *siteId*, *nodeId* but different values for the attribute *processId*;
  - the same value for the attributes *probeId*, *siteId*, *nodeId*, and *processId* but different values for the attribute *threadId*.
- If two measurement elements *m1*, generated from tuple *t1*, and *m2*, generated from tuple *t2*, have the same values for the attributes *probeId*, *siteId*, *nodeId*, *processId*, and *threadId*, then *m1* must appear in the document before *m2* if the metric in *t1* was specified before the metric in *t2* in the last defining request of the corresponding probe. This rule guarantees that the metric a measurement element refers to can always be inferred from the last defining request.

3. Compression:

- Remove what can be inferred from the last defining request:
  - If the document is the response to a control request whose action has the type VALUE, then the attribute *probeId* is removed from all measurement elements if the document contains values for only one probe.
  - An attribute *siteId*, *nodeId*, *processId* and *threadId* is removed if its value is *null* or if the last defining request for the probe did not specify an aggregation and used neither an asterisk nor a question mark as identifier of the corresponding site, node, process or thread.
- If there is still more than one measurement element, a new “root” measurement element, without any attribute, is generated to nest the other measurement elements.

- If there is a measurement element *M* and an attribute *a* such that the value for attribute *a* is the same in each of the elements nested in *M*, then the attribute *a* is removed from all nested elements and added to *M*.

The following example shows an instrumentation request that measures the number of threads for nodes *c1* (containing the processes *p1* and *p2*) and *c2* (containing the process *p3*) in the site *isc*, as well as a possible measurement document generated for the values measured (comments between <-- and --> are not generated):

```
<instrreq>
  <metric name="THREAD_COUNT"/>
  <site id="isc">
    <node id="c1">
      <process id="*"/>
    </node>
    <node id="c2">
      <process id="*"/>
    </node>
  </site>
</instrreq>

<measurement> <!-- site isc -->
  <measurement> <!-- node c1 -->
    <measurement processId="p2" value="3"/>
    <measurement processId="p1" value="4"/>
  </measurement>
  <measurement> <!-- node c2 -->
    <measurement processId="p3" value="5"/>
  </measurement>
</measurement>
```

If the instrumentation request had specified the aggregate element with the attribute function = MAXIMUM, the measurement document would be simply:

```
<measurement siteId="isc" nodeId="c2" processId="p3" value="5"/>
```

4.3.6. Errors

Responses to requests may also return errors instead of a normal answer according to the following syntax:

```
<!ELEMENT errors (error)+>
<!ELEMENT error (#PCDATA)>
```

where each error element contains an application-dependent error message. For example:

```
<errors>
  <error>File not found: mm.c</error>
</errors>
```

4.3.7. Metrics

Each metric has a unique name. The attribute name in the element metric specifies the unique name of the metric. Metric can be temporal (e.g. wall clock time), spatial (e.g. memory allocated), counter (e.g. number of function calls), and hardware counter (e.g. Level 2 cache misses). The name and the number of metrics supported are dependent on specific implementations of the instrumentation and monitoring tool. Each implementation should provide a metric catalog that documents its supported metrics. As a performance tool may work with different instrumentation and monitoring tools, a metric may need to be associated with a name space. Section 6.4 provides and explains an exhaustive list of all performance metrics that we used in our work.

4.4. Summary

This chapter has shown how to represent programs in several languages (Fortran, C, C++, and Java) using a neutral format defined in XML, and proposed a standard set of requests and responses for communicating with instrumentation engines. This approach not only reduces the dependence of performance tools on a specific instrumentation tool, but also increases their portability, making it possible to support new languages and instrumentation tools at low cost.

A compromise was sometimes necessary in order to unify under a single SIR element several constructs that fundamentally represent the same idea. For example, a C++ programmer may find it strange that a namespace is called a "package", and an object-oriented purist might complain that a *call* element is used to represent a method invocation. Another problem is that not always a lowest common denominator can be found; some concepts are specific for only one language or paradigm and do not have a parallel in other languages.

We must also note that not everything that *can* be represented with SIR *must* be represented, nor does an instrumentation engine need to fully support all of the possible requests in this document to be "MIR compatible". For example, when generating the SIR from a binary file, less information will be available compared to the SIR generated form the source code. The SIR in this case will be extremely reduced, but it will still be valid.

5

Modeling of Performance Data and Problems

This chapter describes JavaPSL, a generic performance specification language for modeling experiment-related data and performance properties of sequential, distributed and parallel programs. Performance properties characterize a specific negative performance behavior of a program and are defined over experiment-related data. JavaPSL is intended to be used as a standard performance information interface that can model a large variety of performance information, and enable portable access to performance information. By using JavaPSL, one can build sophisticated performance tools, for example to provide automatic bottleneck analysis.

JavaPSL uses powerful Java mechanisms, in particular, polymorphism, abstract classes, and reflection, to describe performance properties. Moreover, JavaPSL provides meta-properties (defined as Java abstract classes) so as to describe new properties based on existing ones and to relate properties to each other. JavaPSL was inspired by ASL (see [40] and Section 3.2), but while there is no compiler for ASL, JavaPSL can be compiled with any Java compiler, which makes it easy for a performance analysis tool to use knowledge represented in JavaPSL.

Performance properties are related to the code regions were they are found through experiment-related data. JavaPSL filters and statistics classes can be used to restrict performance analysis to specific experiment-related data, and to compute statistics based on arbitrary sets of performance values.

Figure 13 shows a design of a generic performance tool that tries to automatically find all performance bottlenecks of a program by using JavaPSL. The program files are input to the performance tool. An experiment decision system requests performance data from one or more profiling/tracing/prediction tools, which is then stored by JavaPSL classes describing experiment-related data. Based on this data and pre-defined property specifications, a bottleneck analysis system computes a set of performance properties, which are stored together with experiment-related data as JavaPSL classes.

A cyclic search process for performance bottlenecks is invoked, during which the bottleneck analysis system computes, examines and stores performance properties, and initiates additional performance experiments through the experiment decision system. If all bottlenecks—specified in the performance property specification repository—are found or a time limit is reached the search is stopped and performance bottlenecks can be visualized or further examined. Moreover, not only performance tool builders but also the user can be given the possibility to add new properties and change or delete existing ones in the performance property specification repository.

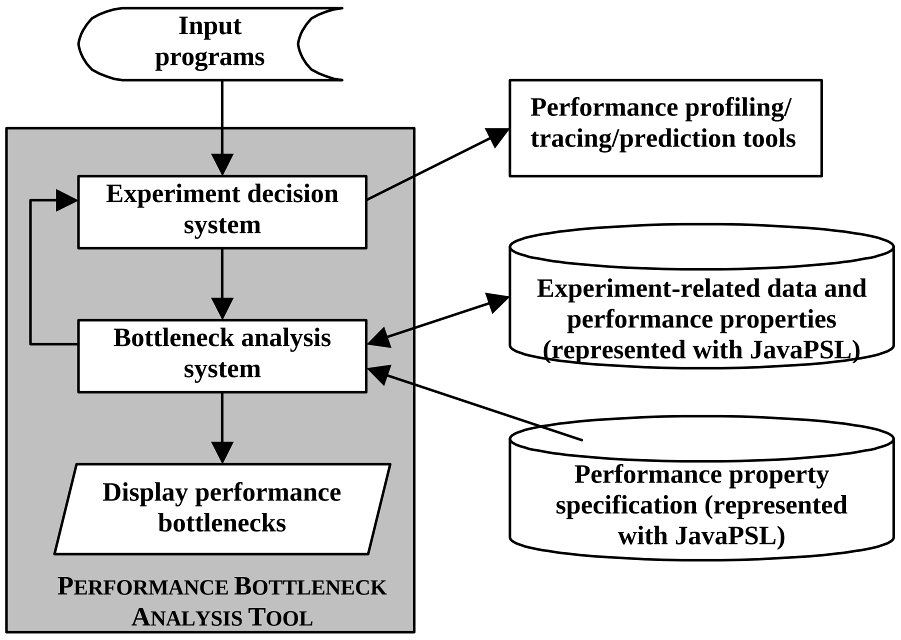

Input programs
Performance profiling/ tracing/prediction tools
Experiment decision system
Experiment-related data and performance properties (represented with JavaPSL)
Bottleneck analysis system
Performance property specification (represented with JavaPSL)
Display performance bottlenecks
PERFORMANCE BOTTLENECK ANALYSIS TOOL

Figure 13. Design of a performance tool that automatically tries to find all performance bottlenecks by using JavaPSL

This chapter presents several examples that show how to model performance properties of an application, including non-scalability, load imbalance, inefficiency, and various overheads such as synchronization, communication, loss of parallelism, control of parallelism, late sender, and cache misses. We focus on JavaPSL as a performance specification language and on its flexibility to describe large classes of experiment-related data and performance properties.

5.1. Experiment-related Data

An *experiment* refers to a sequential, parallel or distributed execution of a program on a given target architecture. Every experiment is described by *experiment-related data*, which includes information about the application code, the machine on which the code has been executed, and measurements obtained from the execution.

JavaPSL uses the syntax and semantic rules of the Java programming language in order to specify experiment-related data. Figure 14 visualizes the JavaPSL classes for experiment-related data by using UML (Unified Modeling Language [119]).

An *application* (program) may have a number of code *versions* (implementations), each of them consisting of a set of *source files*. Every source file is identified by a *URI* (Uniform Resource Identifier [116]) and has one or several static *code regions* (ranging from the entire program to single statements) whose location is specified by *startLine*, *startColumn*, *endLine* and *endColumn* (positions where the region begins and ends in the source file). A code region *p* that statically contains code region *c* is the *parent* of *c* if there is no other code region that *p* statically contains and that also contains *c*. In this case, *c* is one of the *children* of *p*.

Versions are also associated with one or several *experiments*. Each experiment is executed with a set of *input parameters*. The *semantics* of an input parameter says if the parameter is machine-size related, problem-size related, or neither.

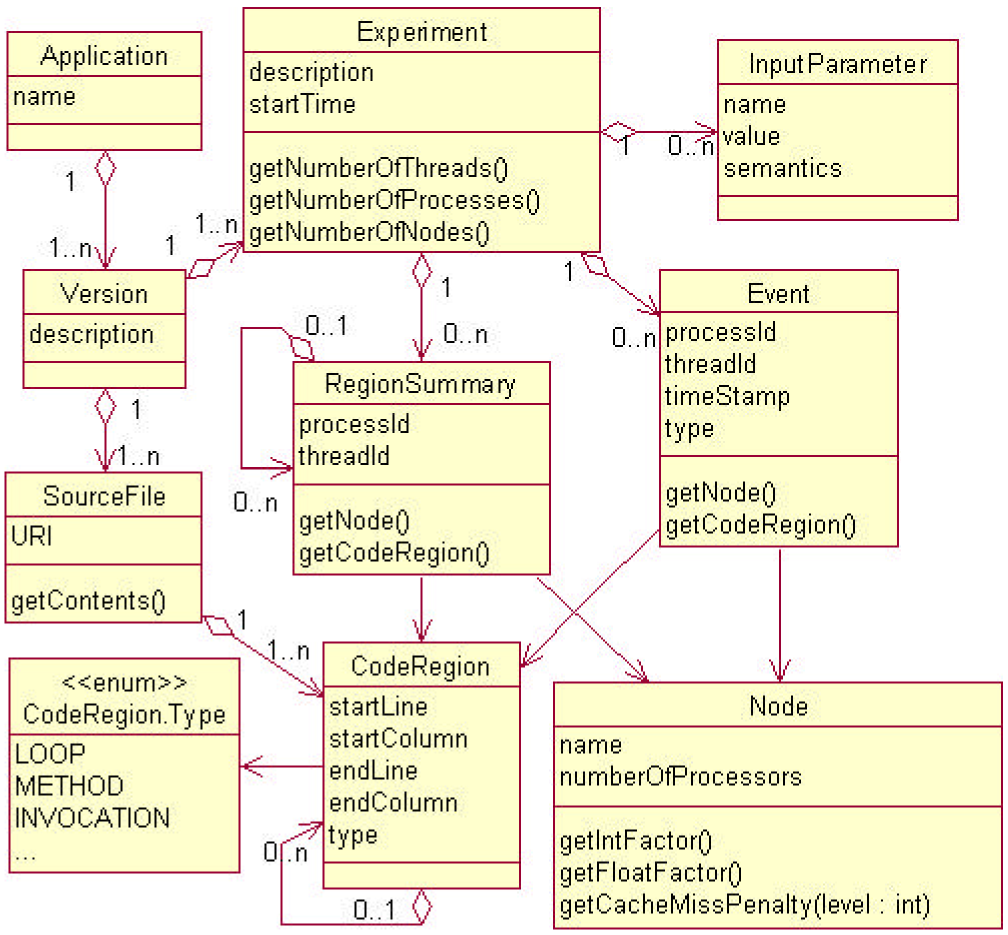


Figure 14. Experiment-related data

A *region summary* (profile information) represents an execution of a code region by a certain process and thread in some *node* (machine). Similar to code regions, region summaries have also a parent-child relationship: Let $r_{p,T}$ be the region summary representing an execution of code region p in thread T, and $r_{c,T}$ the region summary representing an execution of code region $c \neq p$ also in thread T. If the execution of p started when the execution of c started or before, and finished when the execution of c finished or later, then r_p is the parent of r_c (and r_c is one of the children of r_p) if there is no code region g ($g \neq p$ and $g \neq c$) for which an execution in thread T:

- started when the execution of p started or later, and
- finished when the execution of p finished or before, and
- started when the execution of c started or before, and
- finished when the execution of c finished or later.

Metrics measured during the execution must be represented in a subclass of *RegionSummary* specific for a given environment (e.g. a parallel Fortran application using message-passing or a distributed Java using RMI) as shown in Section 7.5.5. For example, subclasses of *RegionSummary* may provide information about execution, synchronization, communication, and waiting time, or about hardware metrics like cache misses or number of floating-point instructions executed.

A set of *events* can also be used to obtain performance information. Each event has at least a type, the location where it originated (code region, node, process, and thread), and the time it occurred (time stamp). As with region summaries, one can add event information specific for a given environment by extending the class *Event*.

Each *node* has specific characteristics which, as shown later, are important for evaluating the performance of an application: its number of processes, its *int factor*, which gives the relative power of the machine when executing integer operations, its *float factor* (similar in concept to the *int factor*), and the penalty for a cache miss. The power of a machine for integer or floating pointing operations can be computed using benchmarks like SPECint95 [138], as long as the powers of all nodes are comparable and one can say how a node is faster than other. Additional characteristics may be added by extending the class *Node*.

With the present advanced monitoring and profiling technologies (e.g. dynamic profiling [89], hardware profiling [1], and source code profiling [91, 147]), there is basically no barrier to represent experiment-related data for arbitrary parallel and distributed programs.

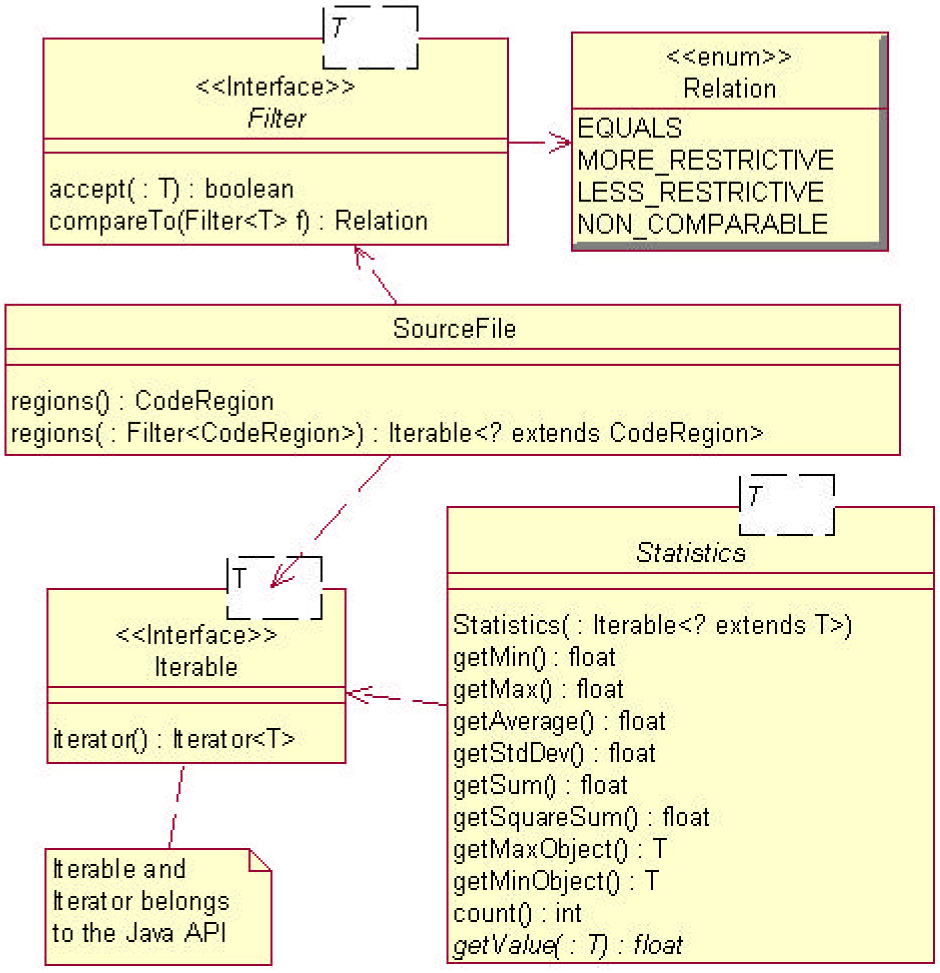


Figure 15. Statistics, Filters and Iterable objects

5.2. Filters and Statistics for Experiment-Related Data

Figure 14 shows several aggregations, denoted by an arrowhead line with a diamond ◊ at its base. Following the *Iterator pattern* [47], aggregate objects (those pointed to by the arrow) are represented in JavaPSL with *Iterable* objects, which allow them to be accessed without exposing their internal structure. For example, the method *summaries* of an Experiment instance can be used to obtain an aggregate object that iterates over all region summaries of this experiment. Optionally, a *filter* can be passed as a parameter to this method. In this case, the aggregate object returned will provide access only to the summaries satisfying the filter condition(s) and restricting the performance analysis to a subset of the data. Finally, statistics about an iterable object may be computed using the class Statistics. Figure 15 shows the relationship between *Filters*, *Statistics* and *Iterable* objects using *SourceFile* objects.

Filters must implement the interface *Filter<T>*, the most important method of which, *accept*, verifies whether an object of type T is accepted or rejected by the filter. A partial-order relationship is also defined between filters: if filter f_1 is more restrictive than filter f_2 (denoted $f_1 > f_2$), then the set of objects returned using the filter f_1 *is contained in* the set returned using filter f_2, and a tool may use this fact to speed up the iteration, for instance by caching the set of objects returned and using the cached value with more restrictive filters.

The abstract class *Statistics* provides statistical methods for specific performance information (e.g. communication time of a region summary), which must be provided by overriding the method *getValue* in a subclass of *Statistics*.

In what follows we present a brief example to demonstrate the usage of filters and statistics under JavaPSL (for clarity, code related to partial-order relationship was dropped):

```
1    filter = new Filter<CodeRegion>() {
2      public boolean accept(CodeRegion c) {
3        return c.getType() == CodeRegion.Type.LOOP;
4      }
5    };
6    aggregate = sourceFile.regions(filter);
7    statistics = new Statistics<CodeRegion>(aggregate) {
8      public getValue(CodeRegion c) {
9        return c.getEndLine() – c.getStartLine() + 1;
10     }
11   };
12   average = statistics.getAverage();
13   stdDev = statistics.getStdDev();
```

Lines 1 to 5 create a filter that accepts only code regions that represent loops, line 6 creates an iterable object representing the code regions in the source file *sourceFile* that satisfy the filter condition, and lines 7 to 11 create an object to compute statistics over the size of loops in that source file. Finally, lines 12 and 13 determine the average and standard deviation for the size of loops in the source file.

A special filter called *CodeRegionFilter*, which selects those region summaries that refers to a specific code region, is already predefined in JavaPSL as follows (checks for non-null arguments were omitted for brevity):

```
public class CodeRegionFilter implements Filter<RegionSummary> {
  private final CodeRegion region;
  public CodeRegionFilter(CodeRegion c) { region = c; }

  public boolean accept(RegionSummary rs) {
    return rs.getCodeRegion().equals(region);
  }

  public Relation compareTo(Filter<RegionSummary> f) {
    return f != null && f.region == region ?
```

```
        Relation.EQUALS : Relation.NON_COMPARABLE;
  }
}
```

5.3. Performance Property Specification

A performance property (e.g. load imbalance, synchronization overhead) characterizes a specific negative performance behavior of a program and is defined by three components:

- *holds*: boolean value that determines whether a property holds or not.
- *confidence*: normalized value between 0 and 1 that indicates the degree of confidence in the correctness of the value of *holds*. A confidence value 1 means that the value of holds is very likely to be correct. The closer the confidence value is to 0, the more uncertain the correctness of *holds* is. Defining confidence values for performance properties is based on empirical observations. For instance, properties based on measurements and predictions may, respectively, yield higher and lower confidence values.
- *severity*: normalized value between 0 and 1 that indicates the importance of the property. Severity value 0 means that the property has little importance whereas a severity value 1 may imply a detrimental effect on the overall performance. Severity values can therefore be used to concentrate performance tuning on the most important performance properties first.

JavaPSL uses syntax and semantic rules of the Java programming language in order to specify performance properties and experiment-related data. In the following sections we introduce the key concepts to specify performance properties and show all the properties we defined in our work.

5.3.1. The Interface *Property* and the Abstract Class *SimpleProperty*

All JavaPSL performance properties implement the common interface *Property*, which includes specific methods to express the hold, confidence and severity value of all properties. The value returned by *getSeverity* is undefined if the method *holds* returns *false*.

```
public interface Property {
  boolean holds();
  float getConfidence();
  float getSeverity();
}
```

Properties need also to define constructors, otherwise they cannot be instantiated. Issues about instantiation are covered in Section 5.5.

Many performance properties obey a generic pattern: they compute a severity value which, if greater than 0, indicates that the property holds; the confidence value is per default set to 1. In order to incorporate this generic pattern for performance properties we created an abstract class with the name *SimpleProperty*.

```
public abstract class SimpleProperty implements Property {
    protected float severity;
    public boolean holds()       { return severity > 0; }
    public float getConfidence() { return 1; }
    public float getSeverity()   { return severity; }
}
```

In order to define a novel simple property, commonly only a constructor must be provided to compute a normalized severity value (between 0 and 1). In the following we describe and discuss several important simple properties. These definitions make use of the factor *weight*(c), which denotes the ratio between the execution time of code region c and the overall execution time of an experiment and assures that the severity of the property for code regions with smaller execution times is also small, even if these code regions considered alone are problematic.

Because JavaPSL does not prescribe the information present in subclasses of region summaries (that is, how one can retrieve specific measurements for the execution of a code region), we will make use of the following utility methods:

- *getExecutionTime(codeRegion, experiment)*: computes the maximum execution time of the given code region in the given experiment.
- *power(codeRegion, experiment)* computes an estimation of the total computational power of the machines used to execute the given code region in the given experiment. The estimation may use for instance the *int factor* of the machine (see Section 5.1), or the *float factor*, or a combination of both, depending on the characteristics of *codeRegion*.
- *power(regionSummary)* computes an estimation of the computational power of the machine *regionSummary.getNode()* considering the characteristics of *regionSummary*. For example, if *regionSummary* only has floating point operations, the estimation will use only the *float factor* of the node.

5.3.2. Inefficiency

Given a parallel experiment, the efficiency [79] of a code region is defined as $\frac{T_s}{q \cdot T_p}$, where Ts is the sequential execution time, Tp the parallel execution time, and q the number of processing units that execute the code region. Usually, the efficiency lies between $\frac{1}{q}$ (worst case) and 1 (maximum efficiency).

This definition is adequate if all processors participating in the parallel experiment are equal, but it will lead to wrong conclusions if they have different computational powers. Therefore, we define the efficiency of the execution of a code region c with a set of processors u compared to the execution of the same code region with a more powerful set of processors w as:

$$\mathit{efficiency}(c,u,w) = \frac{T(c,u)}{T(c,w)} \times \frac{P(u)}{P(w)}$$

where T is the execution time of a code region for a set of processors, and P is the power of a set of processors (computed through benchmarks). Note that the above definition of efficiency reduces to the “traditional” one if all processors are equal.

In this new definition, the efficiency usually lies between $\frac{P(u)}{P(w)}$ (worst case) and 1 (maximum efficiency). Values beyond these extremes are in practice also possible, however: a value below $\frac{P(u)}{P(w)}$ indicates that the execution time with a more powerful set of processors is worse then the execution time with the less powerful set, while a value greater than 1 shows a gain above the expected (for example, doubling the computational power causes the application to run three times faster).

As performance properties reflect some negative performance behavior, we define the severity of the property *Inefficiency* for code region c and sets of processors u and w as:

- 0 if $efficiency(c,u,w) > 1$;
- $weight(c)$ if $efficiency(c,u,w) < \frac{P(u)}{P(w)}$;
- $\frac{1}{1-\frac{P(u)}{P(w)}}(1-efficiency(c,u,w))\times weight(c)$ if $\frac{P(u)}{P(w)} \le efficiency(c,u,w) \le 1$.

The factor $\frac{1}{1-\frac{P(u)}{P(w)}}$ guarantees that the severity reaches its maximum value when the execution time of c with set w equals the execution time of c with set u.

The property *Inefficiency* is therefore defined using JavaPSL as:

```
1  public class Inefficiency extends SimpleProperty {
2    public Inefficiency(CodeRegion cr, Experiment expW,
3                        Experiment expU, CodeRegion basis) {
4      float execTimeU = getExecutionTime(cr, expU));
5      float execTimeW = getExecutionTime(cr, expW);
6      float basisExecTime = getExecutionTime(basis, expU);
7      float ratio = power(expU) / power(expW);
8      float eff = (execTimeU / execTimeW) * ratio;
9      float weight = execTimeU / basisExecTime;
10     if (eff > 1)          severity = 0;
11     else if (eff < ratio) severity = weight;
12     else                  severity = (1/(1-ratio))*(1-eff)*weight;
13   }
14 }
```

The constructor of *Inefficiency* receives as arguments a code region *cr*, the inefficiency of which is to be computed, an experiment *expW*, executed with set of processors W, an experiment *expU*, executed with set of processors U, and a code region *basis*, the maximum execution time of which corresponds to the execution time of experiment *expU*. Using the utility method *getExecutionTime*, the property computes the execution time of *cr* in *expU* (*execTimeU*, line 4) and *expW* (*execTimeW*, line 5) as well as the execution time of *basis* in *expU* (*basisExecTime*, line 6), and using the utility method *power*, the property computes the *ratio* between the computational powers (line 7). Finally, the property computes the efficiency of the code region *c*r (line 8), the weight of this code region in the experiment (line 9), and the severity of the property (lines 10 to 12).

5.3.3. Load Imbalance

Given a code region c executed in n processors, $p_1, p_2, \ldots, p_n$, we define the workload $workload(c, p_i)$ of c when executed in processor p_i as

$$workload(c, p_i) = E(c, p_i) \times power(p_i)$$

where $E(c, p_i)$ is the execution time of c when executed by processor p_i, and $power(pi)$ is the power of processor p_i.

The total workload of code region c is defined as:

$$twl(c) = \sum_{i=1}^{n} workload(c, p_i) .$$

Next we define the ideal workload of code region c for processor p_k $(1 \le k \le n)$ as a fraction of the total workload proportional to the power of pk, that is:

$$iwl(c, p_k) = twl(c) \times \frac{power(p_k)}{\sum_{i=1}^{n} power(p_i)}$$

and, if $iwl(c, p_k) < workload(c)$, we define the overload in code region c when executed by processor p_k as

$$overload(c, p_k) = 1 - \frac{iwl(c, p_k)}{workload(c, p_k)}$$

(Note that overload lies always between 0 and 1.)

Finally, the severity of *LoadImbalance* in code region c is defined as

$$\max_{i=1}^{n}(overload(c, p_i)) \times weight(c) .$$

The property *LoadImbalance* can therefore be defined using JavaPSL as:

```
public class LoadImbalance extends SimpleProperty {
  public LoadImbalance(CodeRegion cr, Experiment exp,
                       CodeRegion basis) {
    float totalPower = 0, totalWorkload = 0;
    CodeRegionFilter cfil = new CodeRegionFilter(cr);

    // computes totalPower = $\sum_{i=1}^{n} power(p_i)$ and
    // totalWorkload = $\sum_{i=1}^{n} workload(cr, p_i)$
    for(RegionSummary rs : exp.summaries(cfil)) {
      float power = power(rs);
      float workload = getExecutionTime(rs) * power;
      totalPower += power;
      totalWorkload += workload;
    }
    // computes maxOverload = $\max_{i=1}^{n}(overload(cr, p_i))$
    float maxOverload = 0;
    for(RegionSummary rs : exp.summaries(cfil)) {
      float power = power(rs);
      float workload = getExecutionTime(rs) * power;
      float idealWorkload = totalWorkload * (power/totalPower);
      if (workload > idealWorkload) {
        float overload = 1 - idealWorkload / workload;
        maxOverload = Math.max(maxOverload, overload);
      }
    }
26
27     // computes the severity
28     float weight = Util.getExecutionTime(cr, exp) /
29                    Util.getExecutionTime(basis, exp);
30
31     severity = maxOverload * weight;
32   }
33 }
```

The constructor of *LoadImbalance* receives as arguments a code region *cr*, the load imbalance of which is to be computed, an experiment *exp* which executed *cr*, and a code region *basis*, the maximum execution time of which corresponds to the execution time of experiment *exp*. The first loop in the constructor (lines 9 to 14) computes the total workload of code region *cr* and the total computational power used to execute *cr* in the experiment given; the second loop (lines 17 to 25) computes the maximum overload in *cr*. Finally, the constructor computes the weight of *cr* in the experiment (lines 28 and 29) and the severity of the property (line 31).

5.3.4. Temporal Overheads

For every individual temporal overhead we can specify a unique performance property, the severity of which is computed as the ratio of the corresponding measurement (e.g. synchronization or message passing time) and a reference value (for instance, the execution time of the region or the entire program).

The definition of performance properties for temporal overheads is straightforward if there is a tool able to do the measurements needed. Assume that there is a subclass *FullRegionSummary* of *RegionSummary* which can provide measurements for the metric communication time through the method *getCommunicationTime*. The property *CommunicationOverhead* can be defined as following:

```
public class CommunicationOverhead extends SimpleProperty {
   public CommunicationOverhead(FullRegionSummary rs,
                                CodeRegion basis) {
      severity = rs.getCommunicationTime() / getExecutionTime(basis);
   }
}
```

Many other overhead properties are defined in Appendix G.

5.4. Meta-properties

A meta-property is an abstract property whose definition depends on a set of already defined properties, possibly known only during the execution time. Although this requires the use of Java reflection capabilities, JavaPSL definition of class *MetaProperty* hides the utilization of the Java reflection library from the user.

The public methods of *MetaProperty* are:

- `add(Class<Property> propertyClass, Object… arguments)`

An instance of the property *propertyClass* is created and added to the meta-property. The elements of *arguments* are used as parameters for the constructor of *propertyClass*.

- `add(Class<Property>[] propertyClasses, Object… arguments)`

An instance of each property in *propertyClasses* is created and added to the meta-property. The elements of *arguments* are used as parameters for the constructor of each property in *propertyClasses*.

- boolean allHold()
  Determines if all of the properties that have been added to the meta-property hold.
- boolean anyHolds()
  Determines if at least one of the properties that have been added to the meta-property holds.
- float getAvgSeverity()
- float getStdDevSeverity()
- float getMaxSeverity()
- float getMinSeverity()

Statistical methods that return the average, standard deviation, maximum, and minimum of the severity values among all of the holding properties added to the meta-property.

- float getAvgConfidence()
- float getStdDevConfidence ()
- float getMaxConfidence ()
- float getMinConfidence ()

Statistical methods that return the average, standard deviation, maximum and minimum of the confidence values among all of the holding properties added to the meta-property.

The properties *OverheadForAnyExecution* and *NonScalability*, defined in the following, are examples of meta-property usage.

5.4.1. OverheadForAnyExecution

This abstract property verifies if a single property holds for a code region *cr* in at least one experiment, setting the severity to the maximum severity and the confidence to the minimum confidence among all properties that hold.

```
public abstract class OverheadForAnyExecution
       extends Metaproperty implements Property {
    protected OverheadForAnyExecution(Class<Property> property,
         CodeRegion cr, Experiment[] experiments, CodeRegion basis) {
        for(Experiment exp : experiments) {
            add(property, cr, exp, basis);
        }
    }
    public boolean holds()       { return anyHolds(); }
    public float getConfidence() { return getMinConfidence(); }
    public float getSeverity()   { return getMaxSeverity(); }
}
```

Based on the meta-property *OverheadForAnyExecution*, we can now easily create concrete properties to verify if there is at least one execution of a region for which an overhead property holds:

```
public class CommunicationOverheadForAnyExecution
       extends OverheadForAnyExecution {
```

```
  public CommunicationOverheadForAnyExecution(
       Experiment parallelExp, RegionSummary rankBasis,
       CodeRegion r)    {
     super(CommunicationOverhead.class, parallelExp, rankBasis, r);
  }
}
public class SynchronizationOverheadForAnyExecution
       extends OverheadForAnyExecution {
  public SynchronizationOverheadForAnyExecution(
       Experiment parallelExp, RegionSummary rankBasis,
       CodeRegion r) {
     super(SynchronizationOverhead.class, parallelExp, rankBasis, r);
  }
}
```

5.4.2. NonScalability

The scalability of a parallel application reflects the execution behavior for changing machine and problem sizes. Based on a set of experiments, we say that a code region scales if its efficiency is nearly the same for every experiment in the set.

By quantifying “nearly the same” as the difference between the maximum and the average inefficiency, we can define the property *NonScalability* based on the property *Inefficiency* as follows:

```
1  public class NonScalability extends Metaproperty {
2    private float severity;
3
4    public NonScalability(CodeRegion cr, Experiment[] setExpW,
5              Experiment expU, CodeRegion basis) {
6         for(Experiment expW: setExpW) {
7           add(Inefficiency.class, cr, expW, expU, basis);
8         }
9         severity = getMaxSeverity() - getAvgSeverity();
10   }
11   public boolean holds()       { return severity > 0; }
12   public float getSeverity()   { return severity; }
13   public float getConfidence() { return 1; }
14 }
```

The constructor of *NonScalability* receives as arguments: a code region *cr*, the non-scalability of which is to be computed; a set of experiments setExpW; an experiment *expU*, executed with set of processors U and such that U is less than the computational power of any set of processor used to execute an experiment in setExpW; and a code region *basis*, the maximum execution time of which corresponds to the execution time of experiment *expU*. Now, the constructor just need to compute the severity of several instances of *Inefficiency* keeping all arguments fixed, except for expW (lines 6 to 8). Finally, the severity is computed as the difference between the maximum and the average severity values for all instances of *Inefficiency* created.

5.5. Property Instantiation

Properties need to be instantiated in order to be used by a performance analysis tool. The tool does not know, however, what a property intends to compute, and it might need to create instances with all possible combinations of performance data whose type is compatible with the parameters of the property constructor. This has two undesirable effects:

- Properties need to mix validation code with the logic for computing the severity.
- A large number of meaningless properties will be created, which consumes time and memory.

In JavaPSL, each property may have a static method (that is, a method that does not need an instance to operate on) defining the rules for property instantiation. Let *P* be a property, the constructor of which has *n* parameters p_0, p_1, …, p_{n-1}. Given an index $i < n$, a list *A* of arguments (a_0, a_1, …, a_{i-1}) already fixed, and a list *C* of candidate arguments, the method must return a list *C’* of arguments such that, for each element *c* in *C’*, an instance of *P* created with arguments $p_0 = a_0$, $p_1 = a_1$, …, $p_{i-1} = a_{i-1}$, $p_i = c$ is significant.

The method signature is:

```
public static Object[] argumentAnalyzer(
   Object[] args, int index, Object[] candidates)
```

where *args* corresponds to the list *A*, *index* to the index *i*, *candidates* to the list *C*, and the return value to the list *C’*.

The following example shows the method *argumentAnalyzer* for the property *Inefficiency*, defined in Section 5.3.2. Recall that the constructor receives four arguments: a code region, an experiment, a second experiment with a less powerful set of processes, and the code region representing the main program. The example shows only the principles of the method *argumentAnalyzer*, with ellipsis … denoting code that has been omitted for clarity.

```
public static Object[] argumentAnalyzer(
         Object[] args, int index, Object[] candidates) {
   switch (index) {
      case 0:
         // for the first argument:
         // all code regions are acceptable
         return candidates;

      case 1: {
         // for the second argument:
         // accepts only experiments that executed (CodeRegion)args[0]
         // with more than one thread
         ArrayList<Experiment> list = new ArrayList<Experiment>();
         ...
         return al.toArray();
      }

      case 2: {
         // for the third argument:
         // selects only the experiments that:
         // 1. executed (CodeRegion)args[0]
         // 2. were started with the same input parameters as
         //     (Experiment)args[1]
         // 3. executed using fewer nodes than (Experiment)args[1]
         ArrayList<Experiment> list = new ArrayList<Experiment>()
         ...
         return list.toArray();
      }

      case 3:
         // for the forth argument:
```

```
        // finds the code region in <candidates> corresponding to
        // the main program in (Experiment)args[1] and return it
        CodeRegion[] main = new CodeRegion[0];
        main[0] = ...
        return main;
   }
}
```

5.6. Summary

In this chapter, we introduced JavaPSL, a generic performance specification language for modeling experiment-related data and performance properties of distributed and parallel programs. Performance properties characterize a specific negative performance behavior of a program and are defined over experiment-related data.

JavaPSL uses powerful Java mechanisms like polymorphism, abstract classes, and reflection, to describe performance properties. In addition, JavaPSL provides meta-properties–defined as Java abstract classes–in order to describe new properties based on existing ones and to relate properties among each other.

Performance properties can be related to the code regions that cause them through experiment-related data. JavaPSL filter and statistics classes can be used to restrict performance analysis to specific experiment-related data, and to compute statistics based on arbitrary sets of performance values.

A variety of pre-defined performance properties are supported to analyze one or several experiments, which examine, for instance, the load imbalance or the scalability behavior of a program.

We propose JavaPSL to be a standard performance information interface to model a large variety of performance information, to build sophisticated performance tools (e.g. to provide automatic bottleneck analysis), and to enable portable access to performance information. This interface is intended to be used by performance tools, compilers, program transformation systems, etc.

6

Twilight: An Instrumentation-monitoring Agent for Java

Executed on the same virtual machine where the program to be analyzed runs, Twilight is a Java thread that remains most of the time inactive, waiting for instrumentation and monitoring requests, carrying on the requests that arrive, and sending back the response to these requests. Requests and responses are always XML documents following the syntax for instrumentation and monitoring requests defined in Chapter 4. Written purely in Java, Twilight runs, as a *Java agent*, in any virtual machine that supports Java 1.5. A Java agent is characterized by:

- a JAR file, containing the agent's code;
- a boolean attribute that indicates if the agent may redefine (change) the class files after the Java virtual machine has loaded them (in Twilight, this attribute has the value *true*);
- a class path, which is appended to the virtual machine's boot class path; and
- a method *premain*.

The method *premain* of an agent is executed before the method *main* of the application. This means the agent can embed itself in the Java virtual machine before the application starts running. The method *premain* of Twilight basically just opens a socket that listen to instrumentation and monitoring requests, and creates the thread that will process these requests when they arrive.

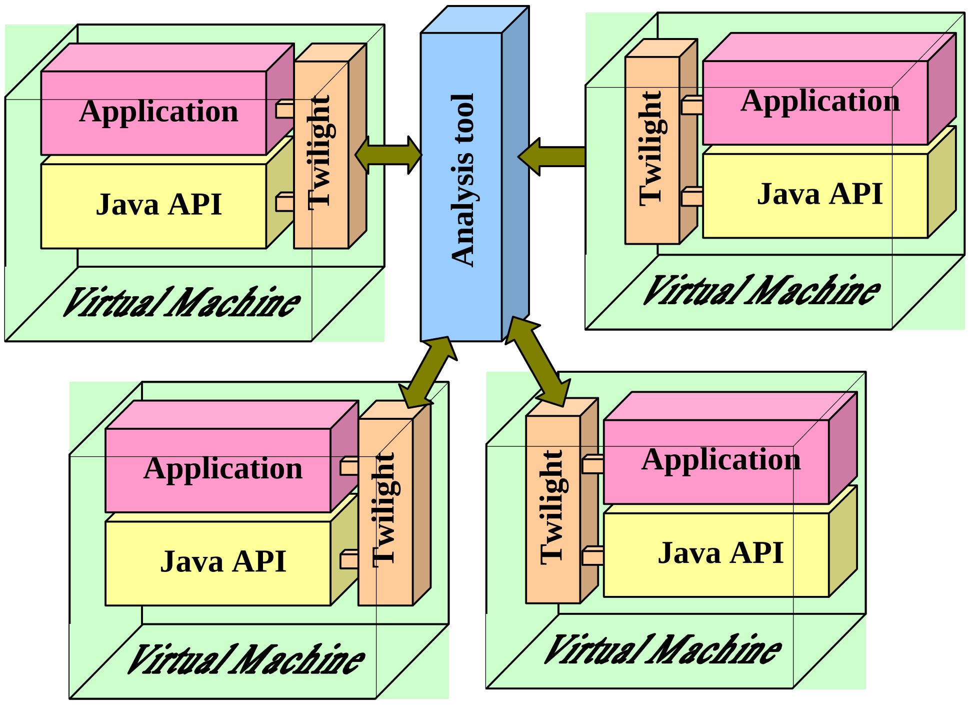


Figure 16. Twilight

Figure 16 shows several virtual machines, each of which containing an "embedded Twilight agent", and how an analysis tool can use the Twilight agents to instrument and monitor a Java application. Note that there is always only one "application" being executed on the virtual machine. Although this application may conceptually contain or be composed of several other sub-applications, Twilight (like the virtual machine itself) cannot see how an application is internally organized. For example, a web server written in Java may host several unrelated services; in this case, Twilight will monitor the web server (and, indirectly, also the services provided), but will not see any distinction between the "conceptual applications."

Since many computers are behind a firewall, an issue that arises when communicating through sockets is security. Java supports transparent access through firewalls at TCP and UCP level by using proxies based on the SOCKS protocol [69] A proxy server, however, must be running to redirect connections between the analysis tool and Twilight agents.

XML is often criticized due to its excessive overhead, in particular for its large document sizes compared to binary files. For this reason, Twilight *may* also work assuming compressed communication, in which case all documents exchanged between the Twilight and the Analysis tool are compressed using the DEFLATE compression algorithm [26] provided in the Java API. This is the same algorithm used in popular compression tools like *gzip* and *WinZip*.

In order to use Twilight as an instrumentation and monitoring agent, one just needs to have a single JAR file installed and start the virtual machine providing the installation path as a Java agent, for instance:

```
java -javaagent:~/Twilight.jar HelloWorld
```

Twilight also provides classes and methods that can be used independently of the agent technology.

As the virtual machine can provide the stack frame of any thread through an invocation of an API method, it is straightforward for Twilight to attend snapshot requests. How Twilight deals with other kinds of requests is examined in the rest of this chapter.

6.1. SIR and SIR Requests

Twilight can generate the SIR (Sections 4.1 and 4.3.1) from Java source codes or class files. The source code must be valid according to the Java Language Specification [74] and its name must end with *.java*. Class files must be valid according to the Java Virtual Machine Specification [75, 84] and its name must end with *.class*. Twilight does not accept the attribute *out* in a SIR request for a class file, as class files are never written back after the instrumentation, but automatically reloaded in the virtual machine (see Section 6.2).

When generated from Java source codes, the SIR will represent the hierarchy of packages, classes, and methods. In addition, for each method, the SIR will contain loops, conditionals, (*if* and *switch* constructs), exception handling (*try…catch…finally* blocks), critical sections (*synchronized* blocks and methods), method invocations (including creation of objects and arrays), and assignments. We used JavaCC [67] to write the source code parser used in Twilight. The bytecode parser we wrote is based on the specification of the class file as defined in the Java Virtual Machine Specification. No tool was used to build the bytecode parser.

When generated from class files, the SIR will represent the hierarchy of packages, classes, and methods. Method invocations, synchronized blocks, creation of objects and arrays, and the majority of loops are also represented. Twilight detects *natural loops*, that is, loops that have only one entry point, using the algorithm described in [95] for detection of natural loops (the main ideas of this algorithm are delineated in Section E.8). Since the Java programming language does not have a *goto* instruction, it is impossible to create a Java program containing loops with more than one entry point; therefore, one can expect that all loops in a class file are also natural loops. We must note, however, that nothing prevents an optimizing compiler or a code obfuscator from generating a non-natural loop from a natural loop, even though we do not know of any such compiler (and we do not see any point in analyzing obfuscated code).

Some information present in the source code is also lost during the compilation, so that equivalent source code constructs may be compiled to the same sequence of instructions, like the loops (a) and (b) shown in Figure 17. This situation arises in Twilight every time two natural loops with the same header are found: it is impossible to know if they were originated from a single loop in the source code or from a loop nested in other. We analyzed 166 class files in the Java API (all the files in the package *java.lang* and subpackages) and found 241 natural loops, 8 of which had the ambiguity problem just discussed. By examining the respective source codes, we determined that 7 cases refer to a single loop, while only one case refers to a loop nested in other. Based on this result, we set Twilight to interpret ambiguous natural loops always as a single loop.

```
do {                                do {
  i++;                                do {
  if (i <= 10) continue;                i++;
  j++;                                } while (i <= 10);
} while (j < 10);                     j++;
                                    } while (j < 10);

              (a)                                 (b)
```

Figure 17. Semantically equivalent constructs that are compiled to the same sequence of bytecodes

Both the detection of natural loops and the detection of synchronized blocks require the computation of successors and predecessors of each instruction in the code, which can be complicated in Java by exception handlers. Consider for example the following code:

```
try {
  a();
  b();
}
catch (InterruptedException e) {
  c();
  d();
}
finally {
  e();
}
```

Besides `b()`, also `c()` and `e()` are possible successors of `a()`; similarly, `a()`, `b()`, `c()` and `d()` are, all of them, possible predecessors of `e()`. On the one hand, successors and predecessors that are product of exception handlers may create cycles which, in some cases, are detected (falsely) as natural loops; on the other hand, we cannot simply ignore the code in exception handlers because it may really contain loops. In Appendix E, we show details about how exception handlers are compiled, how successors and predecessors can be detected, and how we detect only real natural loops and synchronized blocks.

Location information in a SIR generated from class files will always refer to the original source code from which the class files were compiled, which allows one to analyze the class file and relate the analysis back to the source code. Note that location information will be present in the SIR only if the class file was also generated with such information, that is, if the compiler created the class file with debug information.

If a class from which the SIR was generated refers to another class that was not mentioned in the corresponding SIR request, then Twilight adds the referred class automatically to the SIR, but only what is enough to resolve the references. For example, body of methods will not be added, nor methods that the first class does not refer to. This procedure is recursively applied also to the referred classes that are added. When generating the SIR from bytecodes, Twilight can also work in "deep" mode, which means that all of the methods in the referred classes will be added to the SIR, as well as the code of these methods. Some classes, called "forbidden", are always ignored when a deep SIR is generated, however; these are the classes belonging to the Java API and the classes that belong to the Twilight package. The set of forbidden classes can be augmented to include, for example, libraries that are known to have good performance.

Appendix C shows an example of SIR generated from a compiled Java program.

6.2. Instrumentation Requests

The instrumentation code inserted in the application through an instrumentation request is called a probe. Twilight handles the probe insertion differently for source code and for class file instrumentation. When instrumenting the source code, Twilight will write back the new source code with probes. When instrumenting class files, Twilight will insert the probes directly in the bytecodes and ask the virtual machine to reload the new, instrumented version of the class. Therefore, source code instrumentation is suitable for static instrumentation, while bytecode instrumentation is adequate for dynamic instrumentation, as the instrumented version becomes effective immediately.

The attributes *from* and *to* in the element *codeRegion* of an instrumentation requests must refer to code regions in the SIR directly nested in the same element, and the SIR must have been previously generated through a SIR request. Note that the SIR is always generated so as to guarantee that any code region has a single entry point.

6.2.1. Instrumenting Source Codes

Source code instrumentation in Java is tricky; naïve probe insertion may lead to invalid source codes according to the Java Language Specification. Moreover, when instrumenting a code region, we must be aware that an exception may be thrown at any time during the execution of this code region, causing it to finish abnormally.

Consider, for instance, the class below:

```
1  class Example {
2    final int c = 4;
3    void method() {
4      int d = f(c);
5      int c = f(d);
6      int b = f(c);
7      System.out.println(d + " " + c + " " + b); // prints 5 6 7
8    }
9    int f(int arg) { return arg + 1; }
10 }
```

Now, suppose that we want to measure the execution time of the code region starting at line 4 and ending at line 6. Twilight's approach is the following:

1. A construct *try { … } finally { … }* is inserted around the code region; the start probe, which starts the measurement, is inserted right before the construct, and the end probe, which ends the measurement, is inserted in the *finally* block. This guarantees that, even in there is a jump to outside the code region (for instance, because of an exception thrown), the end probe will be executed. Note that ⟨start probe⟩ must be before, not inside, the *try* block; otherwise, if an exception were thrown when the code in ⟨start probe⟩ is executed, ⟨end probe⟩ would also be executed.

```
class Example {
  int c = 4;
  void method() {
    <start probe>
    try {
      int d = f(c);
      int c = f(d);
      int b = f(c);
    }
    finally {
      <end probe>
    }
    System.out.println(d + " " + c + " " + b);
  }
  int f(int arg) { return arg + 1;  }
}
```

2. As the *try* block creates a new scope, the declaration of all variables in the instrumented code region must be moved to outside the try block, otherwise these variables will not be visible when the block finishes (like the variables *b* and *d* in the example). Note that only the declaration is moved, not the initialization.

```
class Example {
  int c = 4;
  void method() {
    <start probe>
    int d, c, b;
    try {
      d = f(c);
      c = f(d);
      b = f(c);
    }
    finally {
      <end probe>
    }
    System.out.println(d + " " + c + " " + b);
  }
  int f(int arg) { return arg + 1;  }
}
```

3. Finally, in order to prevent that any of the variables moved is confused with an instance variable (like the local variable *c* in the example, which hides the instance variable *c*), the moved variables are renamed, as well as references to them.

```
class Example {
  int c = 4;
  void method() {
    <start probe>
    int d$twilight, c$twilight, b$twilight;
    try {
      d$twilight = f(c);
      c$twilight = f(d$twilight);
      b$twilight = f(c$twilight);
    }
    finally {
      <end probe>
    }
    System.out.println(d$twilight+" "+c$twilight+ " " + b$twilight);
  }
  int f(int arg) { return arg + 1; }
}
```

Except for the instrumentation code, the rewritten class and the original one are equivalent. Note that the reference to the instance variable *c*, like the first *f*(*c*), was not renamed.

Twilight also allows to instrument the expressions evaluated inside the constructs *if*, *while*, *for*, *do … while*, *switch* and *synchronized*, as well as the expression of a *return* statement. For these cases, the code is rewritten in an equivalent way that allows the instrumentation of the expression. Consider, for instance, the code fragment below:

```
while (f()) { ... }
```

If the invocation of method *f( )* is instrumented, the code is first rewritten in an equivalent away as shown:

```
boolean exp$1;
exp$1 = f();
while (exp$1) {
  ...
  exp$1 = f();
}
```

Now, the invocation of method *f( )* can be normally instrumented.

Twilight cannot break complex expressions, however. For instance, if the expression in the previous example were *g(f( ))*, Twilight could rewrite the code so as to insert probes that instrument the entire invocation *g(f( ))*, but it would not be able to insert probes to measure only the invocation of method *f( )*.

Synchronized methods are also rewritten if instrumented. Consider the method below:

```
synchronized void method() { ... }
```

When instrumented with Twilight, the method becomes:

```
void method() {
  <start probe>
  try {
    synchronized (this) {
      ...
    }
  }
  finally {
      <end probe>
  }
}
```

Except for the instrumentation code, the rewritten method is equivalent to the original one. Static methods are rewritten similarly, but using the lock associated to the class where the method is declared instead of *this*.

Care is also needed in order to instrument the body of constructors. Consider the following example:

```
class MyWindow extends java.awt.Dialog {
  MyWindow(java.awt.Frame parentFrame) {
    super(parentFrame, "My Window", true);
  }
}
```

According to the Java Language Specification, the invocation of the superclass constructor—*super(parentFrame, "My Window", true)* in the example—*must be*, if present, the first statement in the constructor and, consequently, we cannot insert any start probe before it. Nevertheless, according to the Java Virtual Machine Specification, a class file is allowed to have code before the invocation of the superclass constructor as long as the code does not refer to the instance being initialized. Twilight can then solve the problem of instrumenting the body of a constructor in two steps:

1. The body of the constructor is temporarily assumed to start after the invocation of the superclass constructor, that is, the start probe is inserted *after* the invocation of the superclass constructor.
2. When the compiled class is loaded in the virtual machine, Twilight intercepts the loading, parses the compiled class, moves the start probe to the point *before* the invocation of the superclass constructor, and then return the class to the virtual machine, so that the loading process can go on.

6.2.2. Instrumenting Class Files

With class file instrumentation, it is possible to dynamically insert probes during the execution of an application. If a method is being executed when a probe is inserted, it continues to run without the probe. The version containing the inserted probe will be used in all new invocations (unless the probe is later removed; see Section 6.3).

The instrumentation of class files is analogous to the instrumentation of source files, with the difference that all instrumentation is done at the bytecode level. For example, the *try { … } finally { … }* block explained in Section 6.2.1 is also needed when instrumenting class files, but it is inserted already “compiled”, as well as the start and end probes.

In the following, we show in details how instrumentation in class files is done. An instrumented class file must obey certain constraints, otherwise the Java virtual machine will refuse to execute it. These constraints, as well as the format of class files, are described in Appendix E.

Assume that we want to instrument the code region *R* starting at offset *x* and ending at offset *y+n-1* in the code *C* of a method in a class, where *n* is the size of the last instruction of *R*. Assume also that *R* is a code region with a single entry point but with multiple exit points, that is, outside *R* there are jumps only to the beginning of *R* or to outside *R*, and inside *R* there are jumps to the beginning of *R*, to some other instruction in *R*, and to outside *R*, as shown:

```
     <goto x>
     ...
x:   <first instruction of R>─────┐
     ...                          ├──► Code region to instrument
p:   <some non-branch instruction>│
     ...                          │
     <goto x>                     │
     ...                          │
     <goto p>                     │
     ...                          │
     <goto q>                     │
     ...                          │
q:   <goto r>                     │
     ...                          │
y:   <last instruction of R>      │
y+n: ...                     ─────┘
r:   <some instruction>
     ...
     <last instruction of C>
```

Step 1: Twilight adds the start probe just before *R*. The end probe is inserted immediately after *R*, and also before any instruction inside *R* that causes the control flow to jump to outside *R*. After this step, the code becomes:

```
     <goto x>
     ...
x:   <start probe>
x_1: <first instruction of R>
     ...
p_1: <some non-branch instruction>
     ...
     <goto x_1>
     ...
     <goto p_1>
     ...
     <goto e>
     ...
e:   <end probe>
q_1: <goto r_1>
     ...
y_1: <last instruction or R>
y_1+n:<end probe>
     ...
r_1: <some instruction>
```

```
    ...
```

where x_1, y_1, p_1, q_1, and r_1 correspond the updated offsets for the instructions originally at offsets *x*, *y*, *p*, *q*, and *r*. Note that each instruction after the start probe had its offset changed and therefore any reference to an instruction after the start probe was updated. This was done as following: Given an instruction K, with offset *k*, and an instruction *N* referring to *K* using offset *k*, if the offset of K changes to k_1 after the probe insertion, then:

- If *N* is outside *R* and *K* is the first instruction of *R*, *N* needs not be updated; it already refers to the start probe added just before *K* (first <*goto x*> in the example).
- If *N* and *K* are inside *R* and an end probe was added just before *K*, *N* is updated to refer to the end probe added (<*goto q*> in the example).
- Otherwise, *N* is update to continue referring to *K*, now using k_1.

These rules guarantees that, in the case of a jump to the beginning of *R* from outside *R*, the start probe is executed, but from inside *R* not. They also guarantee that, if the control flow leaves *R* for any reason other than an exception, the end probe is executed.

Step 2: Twilight also changes the code so that, if an exception is thrown and not caught, the end probe is executed before the control flow leaves the instrumented code region. The idea is that Twilight catches the exception and rethrows it after having executed the end probe. Twilight inserts code just after *R* to rethrow the exception, and code after *C* to execute the end probe, as shown:

```
          ...
x:        <start probe>
x1:       <first instruction of R>
          ...
y1:       <last instruction of R>
y1+n:     <goto y1+n+4>
y1+n+3:   <athrow> ;throws the object on the top of the operand stack
y1+n+4:   <end probe>
          ...
          <last instruction of C (originally)>
w:        <end probe>
          <goto y1+n+3> ; this is now the last instruction of C
```

Now, Twilight looks for the first exception handler that is active for the entire code region *C*, that is, it looks in the table of exception handlers of *C* for the smallest *i* for which the exception handler $H_i = (s_i, f_i, h_i, t_i)$ of *C* is such that $s_i \leq x_1$ and $f_i \geq y_1$. A new exception handler, $H = (x_1, y_1, w,$ <*any exception*>), is added before H_i, where *w* is the offset of the exception handler code. If there is no such H_i, then the handler is added at the end of the table. Finally, any exception handler $H_j = (s_j, f_j, h_j, t_j)$ of *C* such that $s_j \leq x_1$ and $f_j = y_1$ is changed to $H_j' = (s_j, y_1+n+3, h_j, t_j)$. Twilight assumes there are not exception handlers that partially overlap, that is, given exception handlers $H_1 = (s_1, f_1, h_1, t_1)$ and $H_2 = (s_2, f_2, h_2, t_2)$ such that $s_1 \leq s_2$, then $f_1 \geq f_2$. This is always the case for class files compiled from Java programs.

What are the effects of these changes? If an exception is thrown when *R* is executed, the added exception handler guarantees that the flow branches to *w*. The end probe is executed and the flow branches to y_1+n+3, where the exception is rethrown. The exception will either be caught by some existent exception handler or, if there is no matching handler, cause the abrupt termination of the method, exactly as it would happen if there were no instrumentation.

6.2.3. Dynamic Instrumentation Without XML

Twilight also provides an interface for dynamic class file instrumentation that is not based on SIR and MIR; in this case, one must parse the class file oneself. Nevertheless, for cases where the entire method is to be instrumented, that is, the code region is the method body, no parsing is needed. The input data for this interface is:

- The class to be instrumented
- The methods in the class to be instrumented
- The offsets of the instructions defining the begin and the end of the code regions to be instrumented (if not given, the method body is used as the code region)
- The methods to be invoked when the code region starts, when it ends normally, and when it ends abruptly, through an exception.

6.3. Control Requests

Requests for modifying a probe or retrieving measured values work with both source code and bytecode instrumentation. A request for modification is used to change the metrics measured or to alter the interval at which the values measured are delivered, while a request for value retrieval is used to get the values measured for the probe, which is useful when the probe is not configured to deliver automatically the values measured. The modification becomes effective immediately.

Requests for removing a probe are supported only for probes inserted directly in the class file. As it happens with probe insertion, if a method is being executed when one of its probes is removed, then it continues to run with the probes. The version without probes will be used in new invocations.

Requests for activate and deactivate a probe are currently not supported.

6.4. Performance Data and Metrics

Performance data measured with Twilight can be sent upon request or at regular intervals to a given destination. In addition, Twilight may send the data when the virtual machine shuts down, for example to a file, an FTP server, or an analysis tool (like Aksum, see Chapter 7) listening to performance data at some port. Performance data are sent always as Measurement Documents, described in Section 4.3.5.

Twilight can currently measure thirty-three metrics, which are shown in Table 5. Moreover, Twilight can be linked with PAPI [109], which in addition allows the measurement of hardware counters (this will make Twilight platform dependent, however.)

Metric Name	*Description*
WC_TIME	Wall clock time.
CPU_TIME	CPU time.
LOADED_CL_TOTAL	Number of classes loaded since the application started.
LOADED_CL_CURR	Number of classes currently loaded.
UNLOADED_CL	Number of classes that have been unloaded.
COMP_TIME	Time spent with just-in-time compilation.

Metric Name	*Description*
GC_COUNT	The total number of garbage collections that have occurred.
GC_TIME	The accumulated garbage collection time.
HEAP_MEM_USAGE	Amount of used heap memory, in bytes.
NON_HEAP_MEM_USAGE	Amount of used non-heap memory, in bytes.
NON_FINALIZED_OBJECTS	Number of objects for which finalization is pending.
THREAD_COUNT	The current number of live threads.
DAEMON_THREAD_COUNT	The current number of live daemon threads.
THREAD_WAITED_COUNT	Total number of times a thread has waited for notification, that is, number of times a thread waited in the methods *Object.wait*, *Thread.sleep, LockSupport.parkUntil*, *LockSupport.parkNanos, LockSupport.park, and Thread.join*.
THREAD_WAITED_TIME	The accumulated time a thread has waited for notification.
THREAD_BLOCKED_COUNT	Total number of times a thread blocked to enter or reenter a critical section.
THREAD_BLOCKED_TIME	The accumulated time a thread has blocked to enter or reenter a critical section
STRING_CREATION	Number of invocations of *StringBuffer.toString()* and *StringBuilder.toString().*
ARRAY_COLLECTION_RESIZE	Number of times a *Vector* or *ArrayList* was resized.
HASH_COLLECTION_RESIZE	Number of times a *Hashtable*, *HashMap*, *WeakHashMap* or *IdentityHashMap* was resized.
NET_SEND	Time spent sending messages through sockets.
NET_BYTES_SEND	Number of bytes sent through sockets.
NET RECV	Time spent receiving messages without blocking.
NET_BYTES_RECV	Number of bytes received through the network.
NET_BLOCKED_RECV	Time spent waiting for the first byte when reading from the network.
NET_INIT	Time spent initializing connections.
NET_CLOSE	Time spent finalizing connections.
NET_ACCEPT	Time spent waiting for a socket to connect.
RMI_SEND	Time spent sending messages in RMI calls.
RMI_BYTES_SEND	Number of bytes sent in RMI calls.
RMI RECV	Time spent receiving messages in RMI calls.
RMI_BYTES_RECV	Number of bytes received in RMI calls.
RMI_BLOCKED_RECV	Time spent waiting for the first byte when reading from the network in RMI calls.

Table 5. Metrics supported in Twilight (time always in milliseconds). Metrics in italics require Twilight to instrument also the Java API

In order do measure some of the metrics in Table 5 (those in italics), some classes in the Java API must be also instrumented. The instrumentation is inserted dynamically, according to the following requirements:

- NET_INIT needs that the methods *bind* and *connect* of class *java.net.Socket* are instrumented.
- NET_CLOSE needs that the method *close* of class *java.net.Socket* is instrumented.
- NET_ACCEPT needs that the method *accept* of class *java.net.ServerSocket* is instrumented.
- NET_RECV, NET_BYTES_RECV and NET_BLOCKED_RECV need that the input stream returned by the method *getInputStream* of class *java.net.Socket* is instrumented.
- NET_SEND and NET_BYTES_SEND need that the output stream returned by the method *getOutputStream* of class *java.net.Socket* is instrumented.
- RMI_XXX metrics need the instrumentation of the method
  Object invoke(java.rmi.Remote, java.lang.reflect.Method, Object[], long)
  in any class that implements the interface *java.rmi.server.RemoteRef* or *java.rmi.server.ServerRef*. This method carries out remote method invocations. The instrumentation inserted measure the values for the corresponding NET_XXX metrics.
- STRING_CREATION needs the instrumentation of the method *toString* in the classes *java.lang.StringBuffer* and *java.lang.StringBuilder*.
- ARRAY_COLLECTION_RESIZE needs the instrumentation of the methods *ensureCapacity* in class *java.util.ArrayList* and *ensureCapacityHelper* in *java.util.Vector*.
- HASH_COLLECTION_RESIZE needs the instrumentation of the method *rehash* in class *java.util.Hashtable*, as well as the instrumentation of the method *resize* in the classes *java.util.HashMap*, *java.util.IdentityHashMap*, and *java.util.WeakHashMap*.

Note: The instrumentation for NET_XXX and RMI_XXX metrics follows the Java 2 API Specification and, unless the specification changes, will work also in future versions of Java. On the other hand, as there is no official documentation about which methods are responsible for resizing a *Vector*, *ArrayList*, *HashTable*, *HashMap*, *IdentityHashMap*, and *WeakHashMap*, the instrumentation for ARRAY_COLLECTION_RESIZE and HASH_COLLECTION_RESIZE depends on specific implementations of these classes. We used as reference the classes in the version 5.0 of the Java Runtime Environment (JRE) distributed by Sun Microsystems [66]. The resizing methods, although not officially documented, are unlikely to change, as they have been the same in the last versions of the Sun JRE and are also used in other JREs, like IBM's [63]. Also note that datagram sockets are currently not supported.

The API is instrumented for measuring some metric only if there is at least one probe in the code that requires the metric to be measured. If, during the execution, the probe is removed or have its set of metrics changed, and there is no probe that needs the metric anymore, the instrumentation is removed (but it may be reinserted later, if necessary).

6.5. Summary

In this chapter, we described Twilight, a library based on state-of-the-art Java technology which can insert probes in the source code of a Java application and dynamically insert and remove probes in the bytecodes of a running Java application.

Twilight can measure several metrics, some obtained directly from the Java virtual machine and some obtained through instrumentation of the Java API.

Both source and bytecode instrumentation can be very tricky in Java, because each code region may end through an exception, that is, any code region has potentially more than one exit point, and because the rules that govern the format of sources and class files may never be broken, even partially, or the application cannot be compiled or executed. We presented these rules and showed how Twilight deals with (and in some cases circumvents) them.

Twilight adopts the XML-based formats proposed in Chapter 4 to exchange information with other tools, which ensures that an application that wants to communicate with Twilight can be written in several languages, not only in Java. For example, one can write a monitoring tool for Java programs using a script language like Perl or Python. Finally, we must note that, although we developed Twilight having performance analysis in mind, one could also use Twilight for implementing other instrumentation-based algorithms, like dynamic detection of race conditions [124] or likely invariants [34].

7

Aksum

Aksum has been designed to be a multi-experiment analysis tool, to a high degree independent of hardware and programming paradigm; it provides the user with a uniform and highly customizable interface to instrument an application, access and analyze performance data relative to several experiments, define how experiments are generated and executed, control the end of the search process, and define the search output. Once this info has been provided (or the default values have been accepted), Aksum automatically conducts performance analysis without any user interference

7.1. Architecture

Figure 18 depicts Aksum's architecture. The *user portal*, illustrated throughout this chapter, provides a user-friendly way of input the data necessary for the search. The *experiment engine*, described in Section 7.2, launches the experiments considering the platform where the application will run. The *instrumentation and monitoring engine*, described in Section 7.3, is responsible for monitoring and instrumenting the application independently of the language or paradigm utilized; it relies on an *instrumentation and monitoring system* to instrument the user's *application* and generate raw performance data, which is processed and stored in the *experiment data repository,* where the experiment engine also stores data. The *search engine*, detailed in Section 7.4, coordinates the entire search process and, using the data in the experiment data repository, tries to detect performance problems (called *performance properties*) in the application. The user-provided data, which influence the search process, flow from the user portal to the Search engine, while the output of the search process flows from the Search Engine to the user portal.

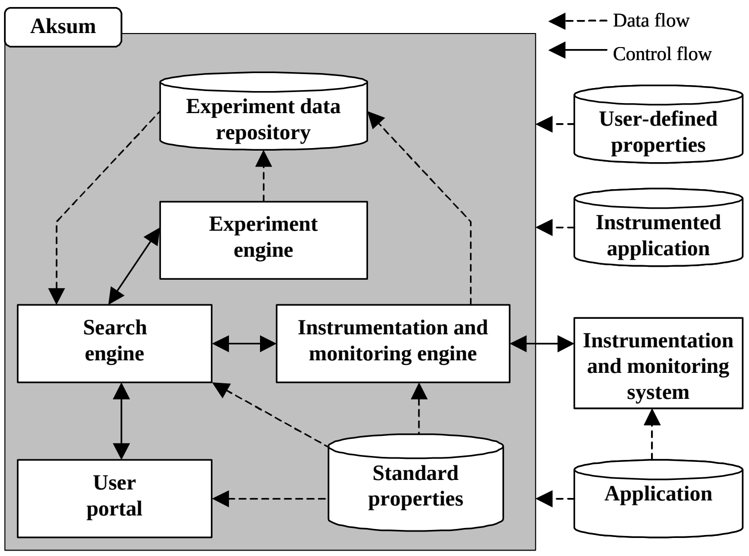


Figure 18. Aksum's architecture

Currently, we use SCALEA [147] and Twilight (Chapter 6) as *instrumentation and monitoring systems*. SCALEA is responsible for instrumenting Fortran programs, while Twilight for the instrumentation of Java programs. We also use the abstract syntax tree generated by the front end of VFC [8], so that we can traverse the structure of Fortran programs and inform SCALEA which code regions must be instrumented.

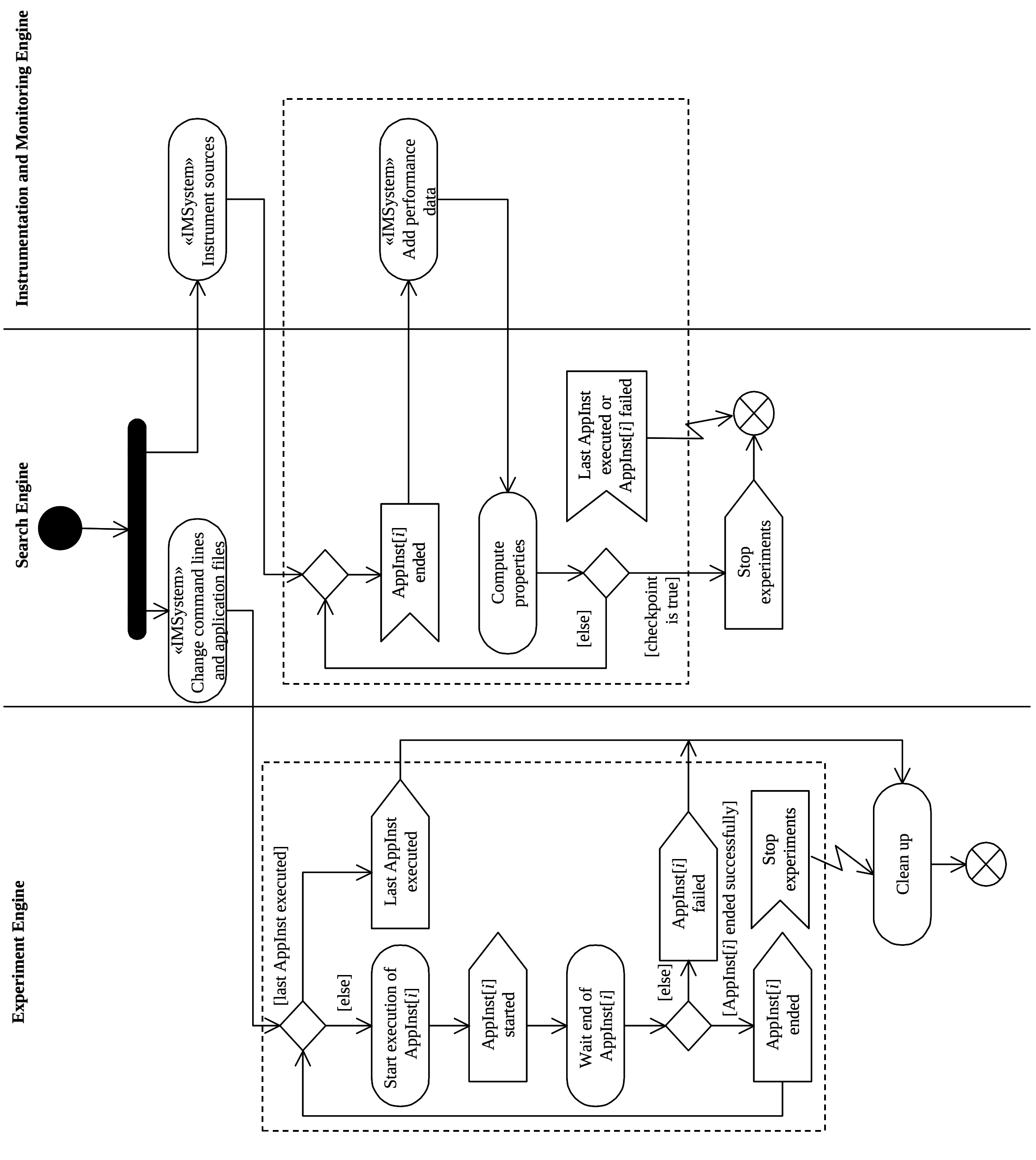


Figure 19. Interaction between the engines of Aksum (static instrumentation). The stereotype «IMSystem» denotes an activity delegated to the Instrumentation and Monitoring System

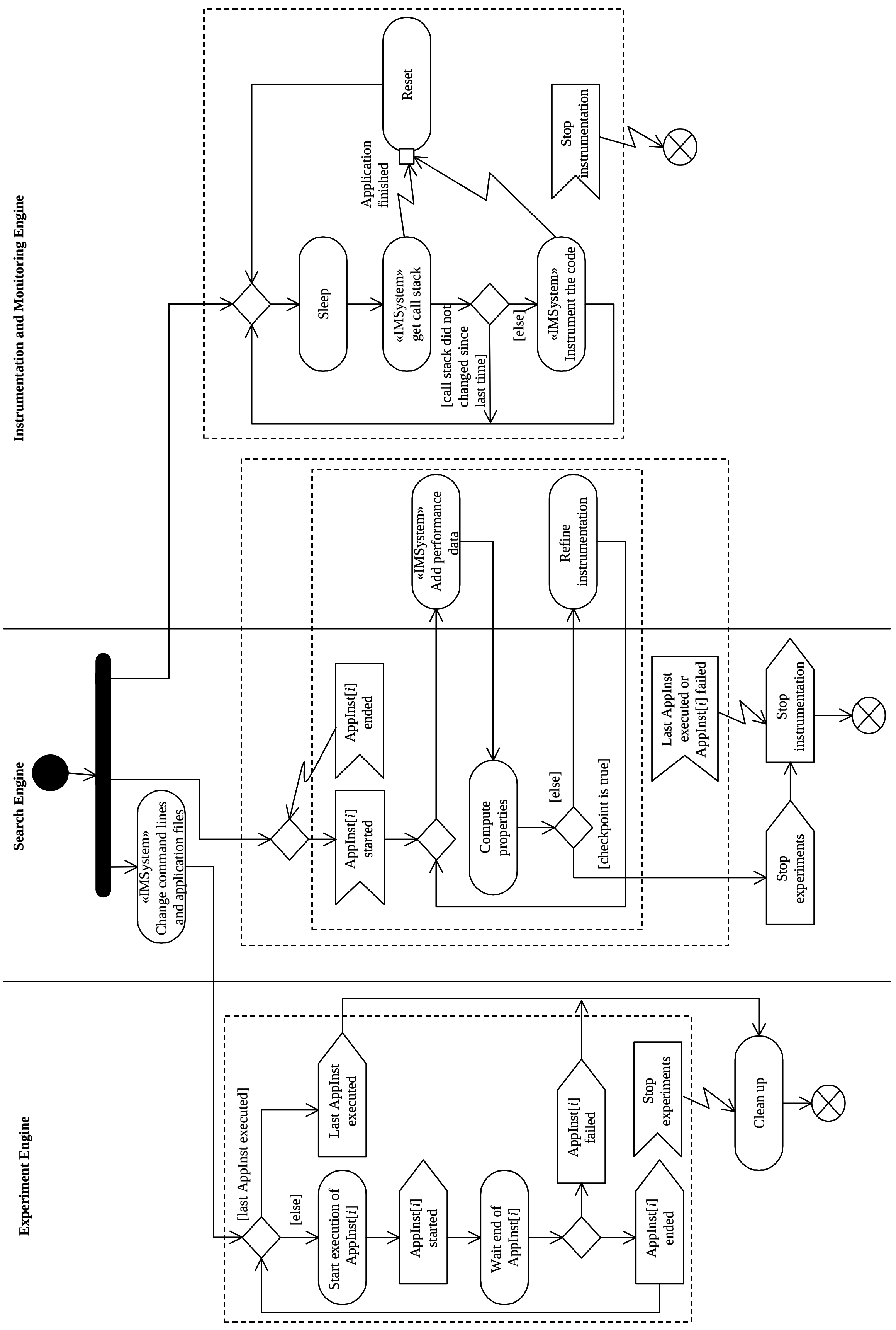


Figure 20. Interaction between the engines of Aksum (dynamic instrumentation). The stereotype «IMSystem» denotes an activity delegated to the Instrumentation and Monitoring System

Aksum uses JavaPSL (Chapter 5) to define and customize performance properties in a systematic and portable way. Aksum has several pre-defined performance properties (such as inefficiency or load imbalance), stored in the *Standard properties* repository, but the user may also define and store new properties in a *User-defined properties* repository. Instances of performance properties found in an application can be grouped, filtered, and displayed in several dimensions as well as plotted on charts.

Aksum may request the instrument of the application before the compilation and execution (static instrumentation), in which case the instrumentation remains the same during the entire application analysis, or it may request the instrumentation while the application is running (dynamic instrumentation), in which case instrumentation is added on demand.

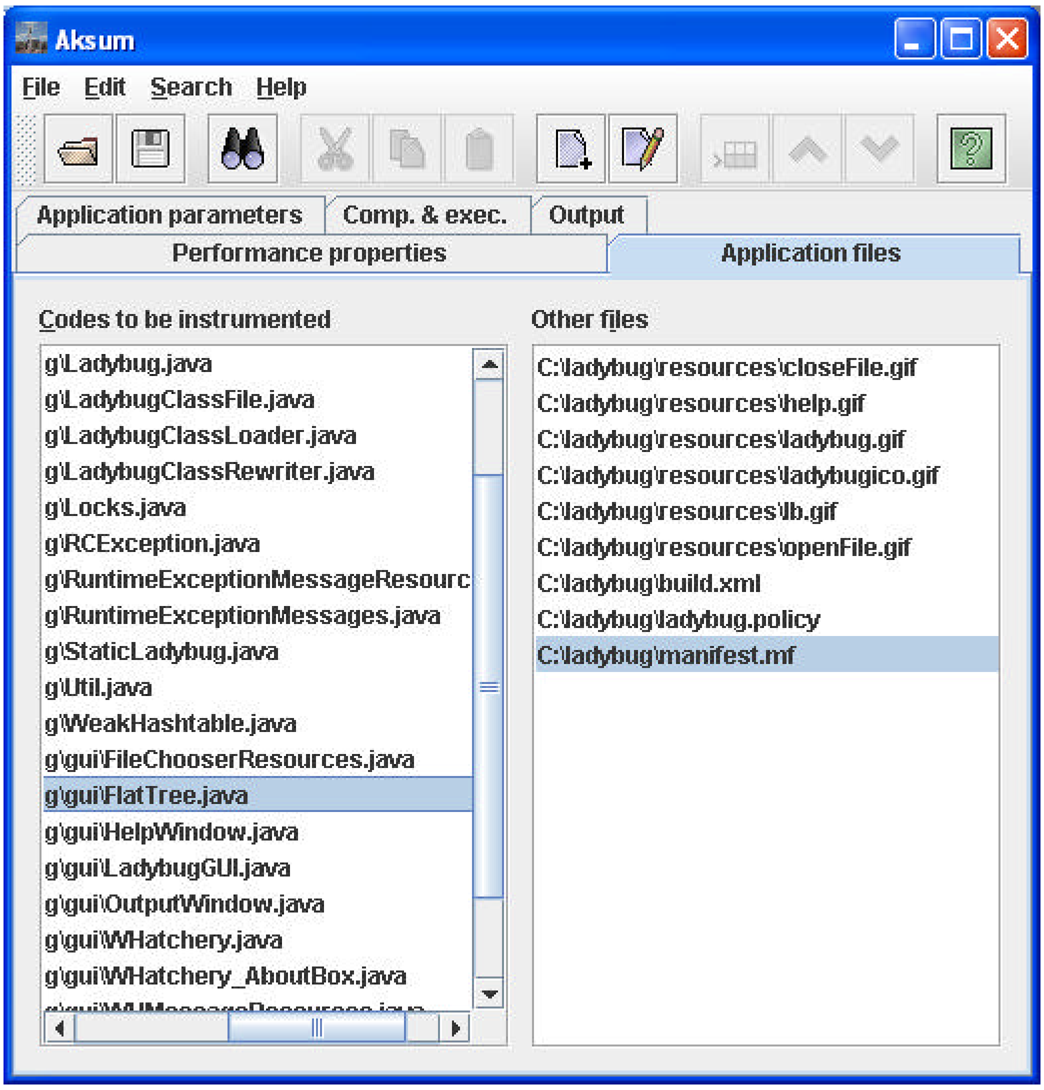
Figure 19 depicts, using UML Activity Diagrams [119], the interaction between the search, experiment, and instrumentation and monitoring engine during the search using static instrumentation; Figure 20 shows the interaction between the engines for dynamic instrumentation. In order to improve the throughput, Aksum creates other threads of control not shown in the diagram; for instance, the Instrumentation and Monitoring Engine sends the request for the call stacks of several processes in parallel. In the rest of this chapter we describe how the search process in Aksum and how the engines and the user portal work together to carry out the performance analysis of an application.

7.2. The Experiment Engine

The experiment engine of Aksum is responsible for launching experiments. This section defines what exactly an experiment is for Aksum, and how experiments can be automatically launched for different input parameters.

7.2.1. Application Files, Command Lines and Directories

An application consists of various files–denoted *application files* in the remainder of this chapter–which are divided into *instrumentable* and *noninstrumentable* files. Instrumentable files are source codes that must be statically instrumented for performance metrics (overheads and timing information) whereas non-instrumentable files refer to source codes the user does not want to be instrumented and any other files necessary to execute an application (e.g., makefiles, scripts, input files, executable files). Files shared among applications (for instance, libraries) may or may not be included as application files. The inclusion of application files in Aksum is shown in Figure 21.



g\Ladybug.java
g\LadybugClassFile.java
g\LadybugClassLoader.java
g\LadybugClassRewriter.java
g\Locks.java
g\RCException.java
g\RuntimeExceptionMessageResourc
g\RuntimeExceptionMessages.java
g\StaticLadybug.java
g\Util.java
g\WeakHashtable.java
g\gui\FileChooserResources.java
g\gui\FlatTree.java
g\gui\HelpWindow.java
g\gui\LadybugGUI.java
g\gui\OutputWindow.java
g\gui\WHatchery.java
g\gui\WHatchery_AboutBox.java

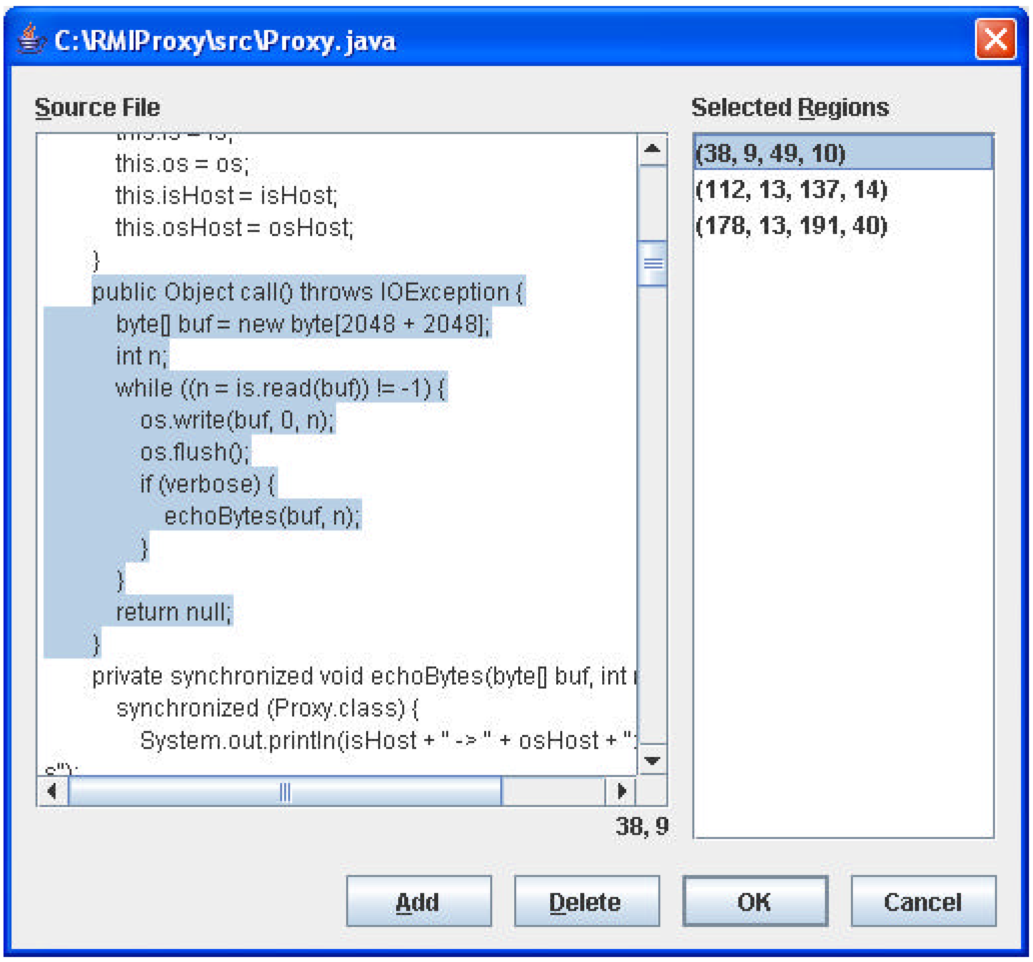
C:\ladybug\resources\closeFile.gif
C:\ladybug\resources\help.gif
C:\ladybug\resources\ladybug.gif
C:\ladybug\resources\ladybugico.gif
C:\ladybug\resources\lb.gif
C:\ladybug\resources\openFile.gif
C:\ladybug\build.xml
C:\ladybug\ladybug.policy
C:\ladybug\manifest.mf

Figure 21. Adding instrumentable and non-instrumentable application files to Aksum

For instrumentable files, the user can also select the code regions that should be analyzed (see Figure 22). If not specified, then Aksum assumes that the entire file must be analyzed. Although instrumentable files make more sense when Aksum instruments the application statically (Section 7.3.1), they also have a meaning in dynamic analysis, denoting files that the user knows beforehand that must be instrumented. We explore further this case in Section 7.3.2.

The user may specify, for each application file, the phases of the experiment where the file is needed. The experiment engine needs to know if a file is needed to start the compilation, the execution, both or neither. Files needed to start the compilation or the execution need special care, as described in Section 7.2.5.

The user must also provide the compilation and execution command lines, as well as the directory where the application is compiled and the directory where it is executed. The user may input command lines as usual, that is, as they are normally input without Aksum, as long as they do not refer to any application file using absolute paths. Absolute paths referring to application files in the compilation command line should be made relative to the directory where the application is compiled, and absolute paths in the execution command line need to be made relative to the directory where the application is executed. Alternatively, the procedure shown in Section 7.2.4 may also be used.



```
        this.is = is;
        this.os = os;
        this.isHost = isHost;
        this.osHost = osHost;
    }
    public Object call() throws IOException {
        byte[] buf = new byte[2048 + 2048];
        int n;
        while ((n = is.read(buf)) != -1) {
            os.write(buf, 0, n);
            os.flush();
            if (verbose) {
                echoBytes(buf, n);
            }
        }
        return null;
    }
    private synchronized void echoBytes(byte[] buf, int 
        synchronized (Proxy.class) {
            System.out.println(isHost + " -> " + osHost + ":
s");
```

```
(38, 9, 49, 10)
(112, 13, 137, 14)
(178, 13, 191, 40)
```

38, 9

Figure 22. Selection of regions to analyze

7.2.2. Application Input Parameters

An application input parameter defines a string that should be replaced in some or all of the application files and in the execution and compilation command lines before the application is compiled and executed. An application input parameters *v* is defined by the quintuplet (*name(v)*, *searchString(v)*, *valueList(v)*, *filSet(v)*, *semantics(v)*), where:

- *name* is a unique name that identifies the parameter *v*;
- *searchString* represents the string to be substituted;
- *valueList* denotes the list of values the search string will be replaced with;
- *fileSet* describes the set of application files in which the search string will be searched and replaced; and
- *semantics* indicates if the parameter is machine or problem-size related (or neither of them).

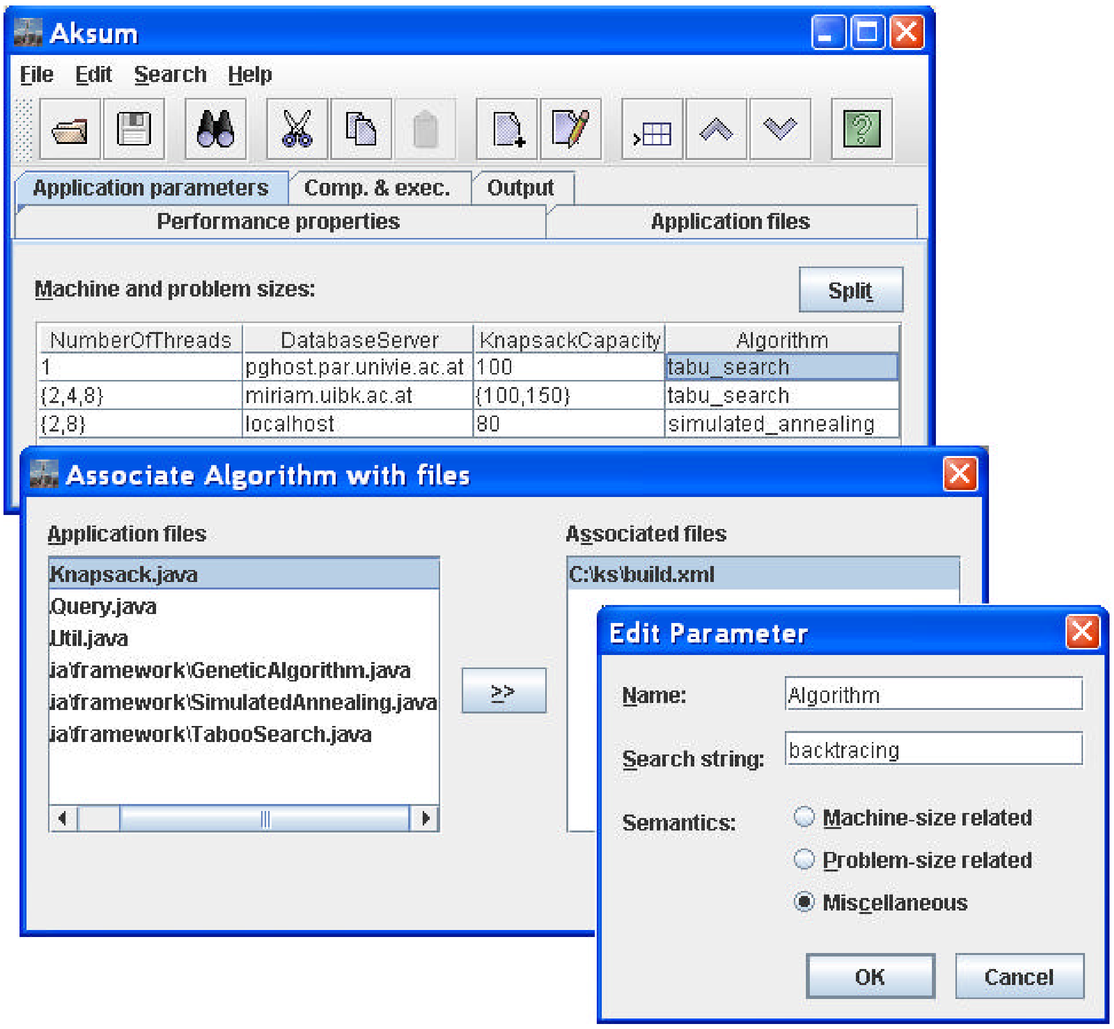
If, for all input parameters, every *search string* is replaced in the associated *file set* with one of the values in the *value lists*, the resulting set of files is called an *application instance*. Formally:

Definition 7.1. Let (v_1,…,v_n) be the list of application input parameters. A set of application files is an *application instance* denoted $AppInst(s_{v_1},\ldots,s_{v_n})$ iif $\forall i$, $1 \le i \le n$, the string *searchString*(v_i) has been substituted by a string s_{v_i} of *valueList*(v_i) in every file of *fileSet*(v_i)

The generation, compilation, and execution of an application instance is called an *experiment*.

Figure 23 shows how the user defines input parameters using the user portal. Four input parameters are shown: *NumberOfThreads*, *DatabaseServer*, *KnapsackCapacity*, and *Algorithm*. Specifically, for the parameter *Algorithm*, we have:

- *name*=Algorithm
- *searchString*=backtracking
- *valueList*=(tabu_search,tabu_search,simulated_annealing)
- *fileSet*={C:\ks\build.xml}
- *semantics*=miscellaneous (neither machine nor problem-size related)



Machine and problem sizes:

NumberOfThreads	DatabaseServer	KnapsackCapacity	Algorithm
1	pghost.par.univie.ac.at	100	tabu_search
{2,4,8}	miriam.uibk.ac.at	{100,150}	tabu_search
{2,8}	localhost	80	simulated_annealing

Application files

Knapsack.java
Query.java
Util.java
ja\framework\GeneticAlgorithm.java
ja\framework\SimulatedAnnealing.java
ja\framework\TabooSearch.java

Associated files

C:\ks\build.xml

Name: Algorithm

Search string: backtracing

Semantics: Machine-size related [✓ Miscellaneous] Problem-size related

Figure 23. Definition of input parameters

7.2.3. Generation of Application Instances

The experiment engine generates application instances according to the following policies:

- Let ($v_1,\ldots,v_n$) be the list of input parameters, and ($s_1,\ldots,s_n$) elements in *valueList*(v_1), . . . , *valueList*(v_n), respectively. $AppInst(s_{v_1},\ldots,s_{v_n})$ is created iff $position(s_{v_i}) = position(s_{v_j})$ ∀i, j: 1 ≤ i j ≤ n, where $position(s_{v_i})$ denotes the position of an element s_{v_i} in *valueList*(v_i).
- Let $w_1 = AppInst(s_{v_1},\ldots,s_{v_n})$ and $w_2 = AppInst(s'_{v_1},\ldots,s'_{v_n})$. Then w_1 is generated (and later on executed) before w_2 iif ∃k such that $1 \le k \le n$ and $position(s_{v_i}) \le position(s'_{v_j})$ ∀i, 1 ≤ i ≤ k. This option enables the specification of an order for the generation and execution of application instances, which can be important when defining checkpoints (see Section 7.4).

For instance, given the parameters

(p, “-mp 2”, (“-mp = 1”, “-mp = 4”), {myScript.sh}, machine-size related)

(q, “THREADS = 1”, (“THREADS = 2”, “THREADS = 4”), {myScript.sh}, machine-size related)

then two application instances are created, the first with the strings “-mp 2” and “THREADS = 1” substituted in the file myScript.sh respectively by “-mp 1” and “THREADS = 2”, and the second instance with them replaced with “-mp 4” and “THREADS = 4”.

For convenience, a value between braces has a special meaning in a value list, denoting a set of values. For instance, the value list ("-mp = {1, 2, 4, 6, 8}") is equivalent to ("-mp = 1", "-mp = 2", "-mp = 4", "-mp = 6", "-mp = 8"). In particular, the string “*a*:*b*:*c*” inside braces, where *a*, *b*, and *c* are real numbers, is a special shortcut for “$a_0, \ldots, a_z$”, where $z = \lfloor (b-a) \div c \rfloor$ and ∀j, 0 ≤ j ≤ z, $a_j = a + j * c$. The previous example could also have been written as ("-mp = {1, 2:8:2}").

Elements in the value list of an input parameters may also refer to other input parameters using their names prefixed with a dollar sign ($). Given a list of input parameters ($v_1,\ldots,v_n$), if the *k*-th element of *valueList*(v_i) refers to variable v_j using the string *$name($v_j$)*, then, if $i < j$, the string *$name($v_j$)* is substituted by the *k*-th element of *valueList*(v_j). For example, suppose that five application instances must be generated. The file *computation.c* must have the string MS replaced with 10 in the first application instance, 20 in the second, and so on. The file *Computation.java* also, but the string to be replaced is RS. We can use the input parameters:

(p, “MS”, (“{10:50:10}”), {computation.c}, problem-size related)

(q, “RS”, (“$p”), {Computation.java}, problem-size related)

These parameters are equivalent to:

(p, “MS”, (“10”,“20”,“30”,“40”,“50”), {computation.c}, problem-size related)

(q, “RS”, (“$p”,“$p”,“$p”,“$p”,“$p”), {Computation.java}, problem-size related)

And these parameters are equivalent to:

(p, “MS”, (“10”,“20”,“30”,“40”,“50”), {computation.c}, problem-size related)

(q, “RS”, (“10”,“20”,“30”,“40”,“50”), {Computation.java}, problem-size related)

The use of sets and references in value lists may greatly simplify the definition of input parameters. Moreover, both compilation and execution command lines may refer to any input parameter using its name. For example, for an execution command line like “*run.sh $p*”, the real execution command line used when executing the third application instance would be “*run.sh 30*” (assuming the parameter *p* as above).

7.2.4. Storing Application Instances

Application instances need to be stored in the file system. The experiment engine creates a directory for all application instances and stores each instance in its own subdirectory, where the necessary directory tree is also created. For example, for the ninth application instance, the file */home/joe/prototype/main.c* will be stored (after the input parameters have been substituted) under the directory <*appInstancesDir*>*/9/home/joe/prototype* (with the name *main.c*), where <*appInstancesDir*> is the directory that the experiment engine created for all application instances.

For a smooth compilation and execution of an application instance, any file in the application being analyzed should use relative references (that is, references relative to the location of the file itself) to refer to other files in the application. If this is impossible or unwanted, the user must create an additional input parameter and associate it to any file that uses absolute references, so that they are replaced with the correct directories. The value list of this input parameter must have only one element, namely the string *$absExpDir* followed by the absolute directory name to be replaced. *$absExpDir* is a special string that the experiment engine replaces, when generating the files for the *k*-th application instance, with <*appInstancesDir*>*/k*. For example, if the file */home/jane/app/script.sh* refers to other application files in the directory */home/jane/app* always using */home/jane/app*, the following parameter may be used.

(p, “/home/jane/app”, (“$absExpDir/home/jane/app”), {/home/jane/app/script.sh}, neither problem nor machine-size related)

The extra input parameter guarantees that the correct files in each application instance are used, and not the original ones. *$absExpDir* is also allowed (and replaced) in the compilation and execution command lines.

On the other hand, references to files that do not belong to the application (like libraries) need to be absolute.

7.2.5. When the Instrumentation Takes Place

If Aksum is configured to do static instrumentation, then the source files must be instrumented before the generation of application instances begins, as the instrumented sources and not the original ones, will be used in the generation (static instrumentation is described in Section 7.3.1).

Regardless of which instrumentation mode is used, static or dynamic, both the compilation and the execution command lines may need to be changed, for example in order to link the application with instrumentation libraries. Therefore, the experiment engine asks the instrumentation and monitoring system to change the compilation and execution command lines accordingly. This is not enough, however, as commands to compile and execute the application may be spread also in scripts and other application files, and therefore the search engine also provided all application files marked as needed to start the compilation or execution (see Section 7.2.1) to the instrumentation and monitoring system, which examines and modify them if needed. These modifications are described in Section 7.5.

7.2.6. Compilation and Execution

Using the compilation command line provided by the user and modified by the instrumentation and monitoring system, the experiment manager replaces any special string (see Sections 7.2.3 and 7.2.4) with the associated value and, finally, compiles the application instance. If the compilation fails (detected by examining the return value of the compilation), the experiment manager aborts immediately.

Similarly, the execution command line (already changed by the instrumentation and monitoring system as well as by the experiment manager) is used to execute the application instance. If the execution fails, the experiment manager may, depending on the user’s choice:

- abort
- try to execute the same application instance again, or
- just skip this application instance and go to the next experiment.

Moreover, if the execution fails, the experiment engine uses an optional termination command line, which can clean up everything left by an execution that finishes abnormally. This is specially necessary in multi-experiment analysis, where the next experiment may start only if the last one did not leave any rubbish behind, like zombie processes and open sockets.

The experiment engine notifies the search engine each time an execution ends successfully, and also after the last experiment has finished, successfully or not.

7.3. The Instrumentation and Monitoring Engine

The instrumentation and monitoring engine decides what must be instrumented in the application, without specifying how the instrumentation is done (which is left to the Instrumentation and monitoring system). Aksum has in facts two engines for instrumentation and monitoring, one suitable for dynamic instrumentation and one appropriate for static instrumentation. Both engines are described in the following.

7.3.1. The Engine for Static Instrumentation

This engine decides what must be instrumented in the source files of an application. It receives a neutral representation of the application structure created by the instrumentation and monitoring system (Section 7.5), analysis its *units* (methods in the case of Java, procedures and the main program in the case of Fortran) and decides which code regions need to be instrumented.

A unit has zero or more blocks selected for analysis. Each block is selected as described in Section 7.2.1. In the simplest case, nothing in the unit is selected for analysis, and the unit is just skipped. The second simplest case occurs when the entire unit is selected for analysis, that is, there is only one block, covering the whole unit.

The engine instruments all blocks selected for analysis in an application which are “big enough”, that is, that have *N* or more statements, where *N* is a user-defined number (the best value for *N* is discussed in Chapter 9). Inside each selected block, we instrument:

- any loop that is not nested in other loop;
- any loop that is nested in other loops but is not the only nested statement;
- subroutine calls and method invocations, but only if the subroutine called or method invoked has *N* statements or more.

If any code region begins in a block but ends outside, the block is “rounded up” to include the end of the code region.

Suppose, for instance, that the following Java method is completely selected for analysis, and that this method, as well as the method *createMatrix*, has more than *N* statements, but the methods *someComputation* and *println* do not.

```
         void compute() {
    B      int[][] matrix = createMatrix();
           int sum = 0 ;
           for(int i = 0; i < L; i++) {
             for(int j = 0; j < C; j++) {
 A   C         sum += someComputation(matrix[i][j]);
             }
           }
           System.out.println(sum);
         }
```

The code region A is instrumented because the method *compute* has more than *N* statements; code region B, an invocation of a method with more than *N* statements, is also instrumented. Code region C is a loop not nested in any other loop, so it is instrumented too. The loop nested in C is not instrumented, as it is the only statement nested in C. The invocations of *someComputation* and *println* are also not instrumented, as these methods have less than *N* statements.

After the application has been instrumented, the instrumentation engine notifies the experiment engine that the generation of application instances may start.

7.3.2. The Engine for Dynamic Instrumentation

This engine decides what must be instrumented in the application *during* its execution by analyzing the files executed. The engine asks periodically for the call stacks of the execution and instruments the units (methods or subroutines) found, according to the following algorithm:

```
While the application instance is running
  Sleep some time
  Ask for the call stacks of all processes that make up the
    execution
  Ask for a view, in each process, of the application structure
    containing the units found in the call stacks
  Merge the views obtained from different processes in a single view
  For each unit in the application and in each process
    Instrument the unit itself
    [Instrument all code regions in the unit]
```

The time the engine sleeps varies according to the behavior of the application. At the beginning, the engine asks frequently for the call stacks in expectation of finding “hot spots”, that is, units where the application spends most of the time. As long as the call stacks do not change, the “time to sleep” increases at each step. If new units appear in the call stack, the “time to sleep” is reset to a small value and the engine starts again to ask for call stacks frequently.

Providing the call stacks and a view of the application, as well as doing the real instrumentation, is responsibility of the Instrumentation and Monitoring System (Section 7.5). As the engine needs to merge application views from different processes, the elements in the view need to be comparable for equality, as detailed in Section 7.5.2.

It deserves note that, when instrumenting dynamically Java applications, the instrumentation may not become effective immediately. When a method is instrumented, only its next executions will use the instrumented version; invocations in the call stack at the time the instrumentation was inserted will still use the old version of the method. Although Java allows to pop invocations off the stack, this is a problematic approach as the global state of the application (for example, open files or global variables) cannot be reverted to what it was before the method was invoked. As this postponed instrumentation may pose a problem for long-running methods, Aksum allows the user to specify, under “instrumentable files”, those files that must be instrumented as soon as the execution begins, so that the first execution of the method already uses the instrumented version.

The engine may operate in two modes: *refining* or *non-refining*. In refining mode, only methods are initially instrumented. Later, if the search engine (described in Section 7.4) finds a performance problem with some method, it informs the instrumentation engine that the analysis in that method needs to be refined, and only then the code regions in the method are instrumented. Firstly, only code regions immediately nested in the problematic method will be instrumented; if performance problems are found in any of the newly instrumented code region, the instrumentation engine is informed again and repeats the refinement process for the problematic code regions (see column “Search Engine” in Figure 20). The use of call stacks and refinement is the same approach adopted in more recent versions of Paradyn [89, 118]

While the dynamic instrumentation used in Paradyn becomes effective immediately, the instrumentation inserted in a method in Java becomes active only for the next executions of this method, that is, refining mode may be inadequate for Java applications with long-running methods. For this reason, Aksum offers also the non-refining mode where, for each method in a class, all code regions found in the method (besides the method itself) are instrumented. This approach would be excessively intrusive for static instrumentation, but it makes sense for dynamic instrumentation because only methods found in the call stack are selected for instrumentation. Furthermore, the representation of the application generated from binary files, which is probably the case when doing dynamic instrumentation, is considerably less rich than that generated from the sources, there not having many code regions inside a method to instrument. The non-refining mode can go even further and asks for the instrumentation of methods that *may* be invoked in the future, which are found by recursively examining the code of the methods in the stack. Note, however, that this search is inherently inexact, as in general the real method invoked is determined only when the invocation occurs. A potential problem with the both refining and non-refining mode is that several code regions (like *get* and *set* methods) are executed often but have an insignificant execution time. For this reason, it is possible to specify in Aksum when a code region is too small to be instrumented. By default, Aksum considers a code region “small” if it contains less than five code regions immediately nested and these code regions either are also small or have a nesting level less than two.

Every time an application instance finishes the execution, the engine repeats the steps above to the new instance, until all instances have been executed.

7.4. The Search Engine

The Search engine coordinates the search process; it seeks out performance properties in the application using the data generated by the other engines and stored in the Experiment data repository.

Properties are hierarchically organized into tree structures called *property hierarchies*, which are used to tune and prune the search for performance properties. For example, one may assume that, if an application is efficient, there is no need to compute its load imbalance. This assumption can be encoded in a specific property hierarchy by placing the property *LoadImbalance* under the property *Inefficiency*. Another example would be the definition of a property hierarchy without any communication properties when it is known that the application is encoded as an OpenMP code and runs on a shared memory machine.

Each node in the property hierarchy represents a performance property and is described by two elements:

- *Performance property name*: the name of the performance property associated with this node; the property definition is stored in a property repository (defined by the user or provided by Aksum).
- *Threshold*: a value that is compared against the severity value of each instance of the property represented by this node; if the severity value is greater than or equal to this value, then the property instance is *critical* and will be included in the list of *critical* properties.

Figure 24 shows a property hierarchy with six properties, and how the property *LoadImbalance* is customized. There are four standard property hierarchies provided by Aksum, covering message passing, shared memory, mixed parallel programs, and distributed Java programs, but the user can define and store new property hierarchies from scratch or based on these predefined hierarchies. The reference code region for every property node in the predefined property hierarchies is per default set to the main program.

The process of searching for performance properties usually finishes when all application instances have been executed. In addition, Aksum supports the definition of checkpoints to stop the search for properties before the end of the last experiment. A checkpoint is a Boolean function defined as follows:

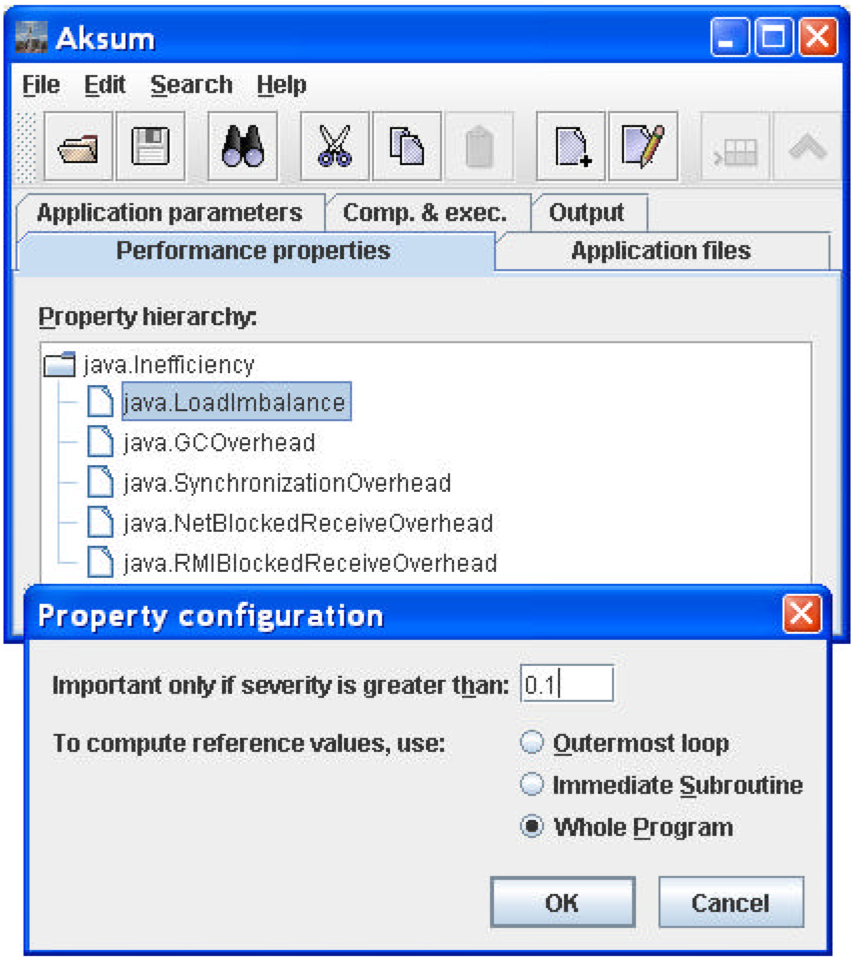
op(*severity*(*property*, *code region*, *number of experiments*)) *relop value*

where *op* $\in$ {maximum, minimum, average, standard deviation}and

relop $\in$ $\{>, \geq, <, \leq, =, \neq\}$. *Any property* and *any code* region are also valid values for *property* and *code region*.

The following checkpoint, for instance, means that the search must stop if the severity of the any property in any code region is greater than 0.6.

maximum(severity(any property, any code region, 1)) > 0.6



Property hierarchy:

- java.Inefficiency
  - java.LoadImbalance
  - java.GCOverhead
  - java.SynchronizationOverhead
  - java.NetBlockedReceiveOverhead
  - java.RMIBlockedReceiveOverhead

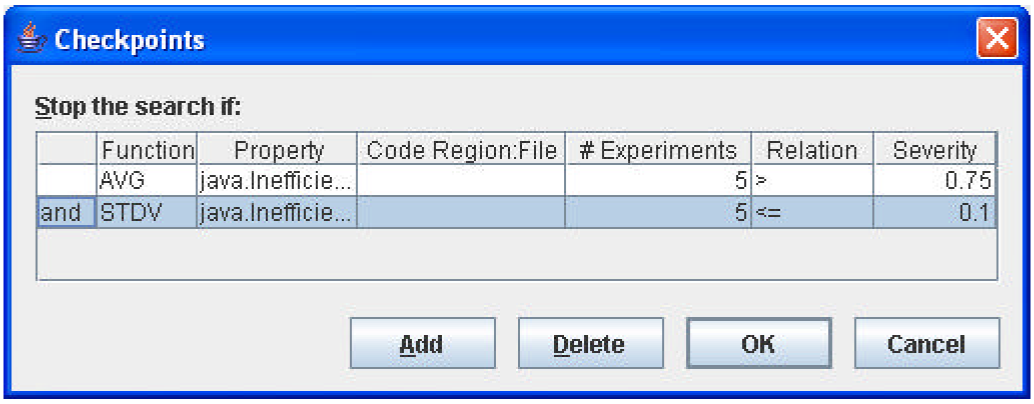
Property configuration

Important only if severity is greater than: 0.1

To compute reference values, use: ( ) Outermost loop ( ) Immediate Subroutine [✓ Whole Program]

Figure 24. Property hierarchy and property customization

Figure 25 shows a checkpoint definition that stops the search process if the average inefficiency for the entire program in the last 5 experiments is above 0.75 with standard deviation less than or equal to 0.1.



Stop the search if:

	Function	Property	Code Region:File	# Experiments	Relation	Severity
	AVG	java.Inefficie...		5	>	0.75
and	STDV	java.Inefficie...		5	<=	0.1

Figure 25. Checkpoint definition; the code region is blank, which means "any code region"

7.5. The Instrumentation and Monitoring System

Tools for instrumenting and monitoring applications have different requirements, inputs, and outputs. This section describes the extra layer that needs to be added to an instrumentation and monitoring tool so that it can communicate with Aksum. This layer, called IM-interface in the rest of this section, isolates all details specific for a given tool, so that the other engines of Aksum can work unaware of the kind of tool used, the platform where the application runs, and even the language in which the application has been written.

An instrumentation and monitoring tool to which an IM-interface has been added constitutes an instrumentation and monitoring system. We currently have two instrumentation and monitoring systems; one connects Aksum to SCALEA [147], the other to Twilight (Chapter 6). In the following, we describe the operations an IM-interface needs to add to an instrumentation and monitoring tool so that the last can be used in Aksum.

7.5.1. Modification of Command Lines and Application Files

The IM-interface must analyze the command lines which compile and start the application and change them accordingly so as to satisfy the requirements of the instrumentation and monitoring system utilized. Aksum gives the IM-interface not only the compilation and execution command lines, but also any files needed to compile and start the application (see Section 7.2.5), like scripts.

The IM-interface does the best effort to recognize compilation, linkage and execution commands in the files given; this means that there is no guarantee that the IM-interface will find all command lines that need to be change or that it will not mistakenly change a line in the file that should not be changed. The following rules are used to detect the commands of interest in an application file:

- Build files (Twilight)

Build files are XML documents used by Ant [3], a popular tool for managing Java projects. Any valid XML document whose root element is *project* is considered to be a build file. Similarly, any element *javac* in a build file is considered an element that compiles the application, and any element *java* is considered an element that executes the application. This rule accords with [3] and with all applications we analyzed that make use of Ant. For example, the following line, which compiles all sources in directory *myApp*, is recognized as a compilation command in a build file:

```
<javac includes="myApp/*.java"/>
```

- Other files (SCALEA, Twilight)

The IM-interface looks for lines that match against either a compilation pattern, a linkage pattern or a execution pattern. We have determined some common patterns based on the applications we have analyzed; for example, the following pattern, written as a regular expression [46], is used in the IM-interface for SCALEA to detect a compilation command line for Fortran programs:

```
\s*(f90|pgf90|f77|g77|\$FC|\$\(FC\)|\$\{FC\}|\$F90|\$\(F90\)|\$\{F90\
}|\$\(F77\)|\$\{F77\})(.*)\s+-c\s+
```

This pattern detects a call to a Fortran compiler (like *$(F77)* or *f90*) with the switch *–c*. The following lines, for instance, would be detected as compilation command lines:

```
f77 –c hello.f
${F90} –mt -O5 –c hello.f90
```

The IM-interfaces for both SCALEA and Twilight allow the user to overwrite the compilation, linkage and execution patterns.

After a compilation, linkage or execution command line is recognized, it is modified according to the requirements of the underlying instrumentation and monitoring tool.

7.5.2. Creation of an Application View

The IM-Interface must provide a view of the application to be analyzed that is to the highest degree language and platform independent and, at the same time, detailed enough to allow a deep and fast analysis. This view is called in Aksum the program tree. For Fortran programs, we use the abstract syntax tree generated by the VFC front end to build the program tree, while for Java programs, we use the SIR generated with Twilight.

Each node in the program tree is labeled so as to identify the structure it represents in the application. The following structures, which are enough for the analysis that Aksum carries out, may be represented in a program tree:

- modules (Fortran)
- classes (Java), labeled *modules* in a program tree;
- programs (Fortran)
- methods *public static void main(String[])* in Java, labeled *program* in a program tree;
- procedures (Fortran), labeled *subroutine* in a program tree;
- methods (Java), labeled *subroutine* in a program tree;
- loops;
- procedure calls (Fortran), labeled *call* in a program tree;
- method invocations (Java), labeled *call* in a program tree;
- conditionals;
- assignments;
- complex conditionals (switch in Java, SELECT in Fortran);
- where (Fortran);
- try, catch, finally (Java);
- jumps;
- OMP loops and parallel loops (Fortran);
- critical sections (synchronized methods and blocks in Java).

Besides its label, a node may contain a name (when representing a module or a class) or a reference to another node (when representing a method invocation or procedure call). Figure 26 shows a program tree representing a small Java class.

```
class C {
   public static void
   main(String[] args) {
      int a = 1;
      int b;
      while (a < 5) {
         b += a * a;
         f(b);
      }
   }

   static synchronized void
   f(int n) {
      ...
   }
}
```

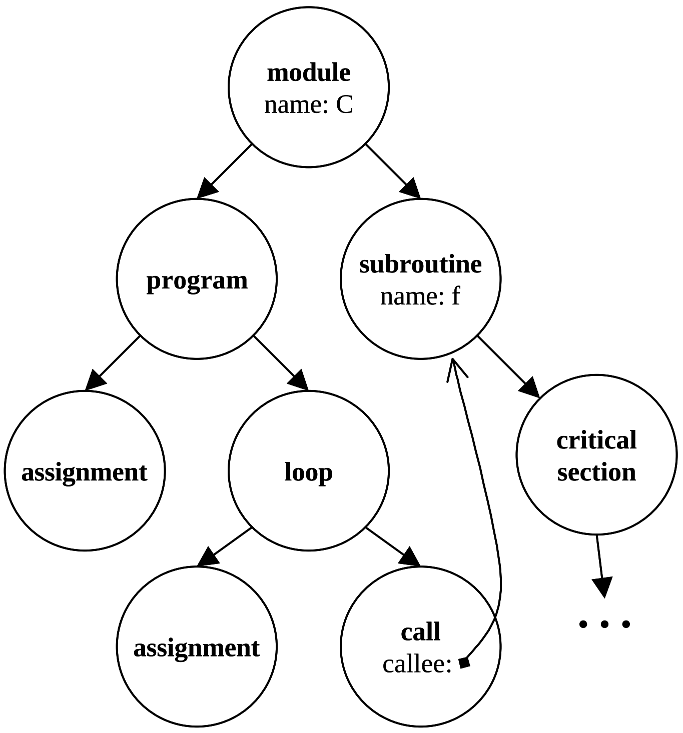


Figure 26. Example of a program tree for a Java class

An IM-interface for a static instrumentation system (that is, one that instruments the sources before they are compiled) receives, as input for creating the program tree, the instrumentable files that compound the application (see Section 7.2.1). An IM-interface for dynamic instrumentation receives as input a process that participates in the application execution and a set of subroutines from which the program tree will be generated (the process and the subroutines are obtained from a call stack, described below). In both cases, the output is the program tree. For the dynamic case, the program tree will contain at least the subroutines given as input; in addition, the IM-interface must guarantee that, given two nodes *n* and *m*, where *n* belongs to the application tree of process *p* and *m* to the application tree of process *q*, then *n.hashCode()* = = *m.hashCode()* and *n.equals(m)* if, and only if, *n* and *m* refers to the same code region in the application. This is needed because Aksum must merge the application trees obtained from different processes in a single tree (see Section 7.3.2).

7.5.3. Providing Call Stacks

The IM-interface for a dynamic instrumentation system (like Twilight) must be able to provide the call stacks of every process participating in the application execution. Each process may have several threads, and each thread has at every instant a call stack. Therefore, when asked to provide call stacks, the IM-interface outputs a set of pairs $\langle p, s_p \rangle$, where each pair represents a process *p* that participates in the execution of the application and the call stacks s_p of all threads in process *p*. Each call stack s_p of process *p* is itself a set of pairs $\langle t, c_t \rangle$, where each pair represents the call stack c_t of thread *t* in process *p*. Finally, each call stack c_t is represented as a list of subroutines $(r_1, r_2, \ldots, r_n)$, where subroutine r_i is called from subroutine r_{i-1} $\forall\, i$, $1 < i \le n$.

7.5.4. Insertion of Probes in the Application

The IM-interface needs to insert probes in the application in order to measure the metrics Aksum needs for the performance analysis. The IM-engine accepts as input for this operation the regions to be instrumented, the performance properties (described in Section 7.4) that will be used in the analysis and, for dynamic instrumentation, also the process where the probes must be inserted. The regions are the output of a request for the application view, while the process is the output of a request for the stack traces.

The IM-interface loads the performance properties given and parses them in order to determine which metrics the property needs. After that, the IM-interface uses the instrumentation and monitoring tool to add the probes needed to measure those metrics.

7.5.5. Generation of Experiment-related Data in JavaPSL Format

The profiles generated by the instrumentation tool must be converted to the JavaPSL format (Chapter 5), which is used in Aksum. The IM-interface parses the profiles and creates instances of the classes describing experiment-related data explained in Section 5.1.

We recall, also from Section 5.1, that there must be a subclass of *RegionSummary* specific for a given instrumentation and monitoring tool, that is, a subclass that can represent the metrics that the tool can measure (or a subset of them). We created subclasses for Twilight and SCALEA which can provide a map with the values for all metrics these tools can measure. For example, in order to get the garbage collection time of a region summary *r* generated with Twilight, one must write:

```
r.getTimingMetrics().get(GC_TIME)
```

7.6. Summary

This chapter presented Aksum, a highly flexible and customizable multi-experiment performance analysis tool that can automatically conduct a set of experiments and detect the performance bottlenecks in these experiments. A particular unique feature of Aksum is its ability to search for performance problems in multiple experiments, whereas most existing tools restrict their analysis to single experiments. As we will see in Chapter 9, the output of Aksum can be presented at various levels of detail and summarized into line charts so as to immediately guide the user to the most critical performance properties detected. In addition, properties can be freely added to or removed from Aksum; the user can specify properties by using JavaPSL (described in Chapter 5).

8

A Learning Agent for Performance Analysis

While the analysis techniques used in Aksum are effective to find performance problems, our ad hoc approach lacked a more formal model that could be used to explain the decisions taken during the analysis and to justify their correctness. We wanted to use a well-established theory to model the performance analysis problem, but we also wanted a theory which, when implemented, performed as good as or better than the implementation we already had.

We chose reinforcement learning to model the performance analysis problem for two reasons: The trial-and-error nature of reinforcement learning resembles closely the empirical character of performance analysis, and, differently of other forms of learning, no expert teacher is required to tell the agent the correct actions to take.

In this chapter, we provide a short background on agents and reinforcement learning, and describe the modeling of performance analysis problems as reinforcement learning problems.

8.1. Background

Starting in 1952, Arthur Samuel employed many ideas of reinforcement learning used today to write programs for checkers [204]; his work is regarded as the earliest successful research on machine learning. Although reinforcement learning is almost so old as the study of learning in artificial intelligence, it was not an especially active area in research until its revival in the early 1980s. In fact, the field of reinforcement learning as studied today dates only from the late 1980s. In the following, we define some important concepts needed to understand our work on reinforcement learning in performance analysis.

8.1.1. Agents

Russel and Norvig [121] define agents as “anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.” Agents also have goals; in particular, a rational agent is one that, based on what its sensors perceive and on the knowledge it has, takes the right action, that is, the action expected to help the agent to achieve its goal.

A *learning* agent, as shown in Figure 27, has, besides sensors and effectors, also a performance element, a learning element, a problem generator, and a critic. The *performance element* decides, based on the perceived state of the environment, the action that should be taken in order to achieve the agent’s goal. The *learning element* defines the agent’s goals using the knowledge from the performance element about which actions were taken in the past and some feedback about their effect, determining how the agent can achieve a better performance in the future. Using an external set of performance standards, the *critic* evaluates the success of the actions, which is needed because the agent has no means to know if the response of the environment is good or not. Finally, the *problem generator* suggests ways of expand the knowledge available.

As an example, consider a simple agent playing chess that needs to decide what to do for a given board configuration. Perhaps the agent has already seen this configuration several times before in other games, and knows that, in seventy percent of those games, moving the queen one square left ended with the agent's victory. As the agent's goal is winning the game, there is some evidence that moving the queen one square left is a good move. The problem generator, however, may suggest that the queen should now move forward, because the agent still does not know the consequences of this move and because there is no guarantee that moving the queen left will lead to the victory anyway. After having moved the queen forward and waited the adversary's move, the sensors realize that agent was checkmated, so the critic translated this information as "bad move". The learning agent uses the translated information to update the knowledge, so that the next time the queen is not moved forward for that board configuration and, perhaps, also for similar board configurations.

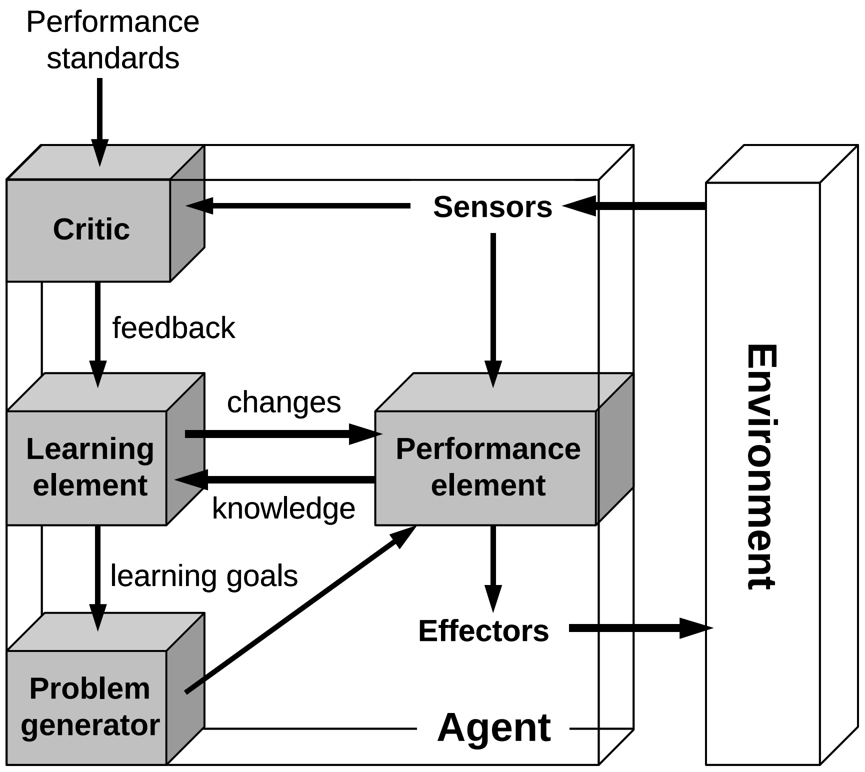


Figure 27. General model of learning agents [137]

8.1.2. Reinforcement Learning

The more widely studied forms of machine learning, like artificial neural networks and decision trees, are based on the concept of supervised learning, a technique where the agent cannot act autonomously but must be trained by a teacher that tells the right action (output) to take at a given situation (input). From the presented training data (input/output pairs), the agent must be able to generalize and predict the correct output for any unseen inputs.

Reinforcement learning is a form of machine learning where the agents are never told the right actions to select given the perceived environment's state. For each action taken, the agent receives a scalar feedback from the environment, called reward or reinforcement, which is the only measure of how good or bad the action was. Consequently, the agent must learn the right action to take by trial-and-error, and learning and evaluation of the system occur concomitantly.

Reinforcement learning is seen as a class of problems and not as a set of techniques –any method suitable for solving a reinforcement learning problem is considered a reinforcement learning method. In a reinforcement learning problem, the agent and the environment interact at each step t of a sequence of discrete time steps; the agent perceives through its sensors the environment's state $s_t \in S$, where S is the set of possible states, and selects action $a_t \in A(s_t)$, where $A(s_t)$ represents the set of actions that can be performed when in state s_t. At the next time step, the agent receives a reward $r_{t+1} \in \mathbb{R}$ and the environment's state changes to s_{t+1}. The interactions between agent and environment are shown in Figure 28. The continuous case is also possible, but we will not deal with it here.

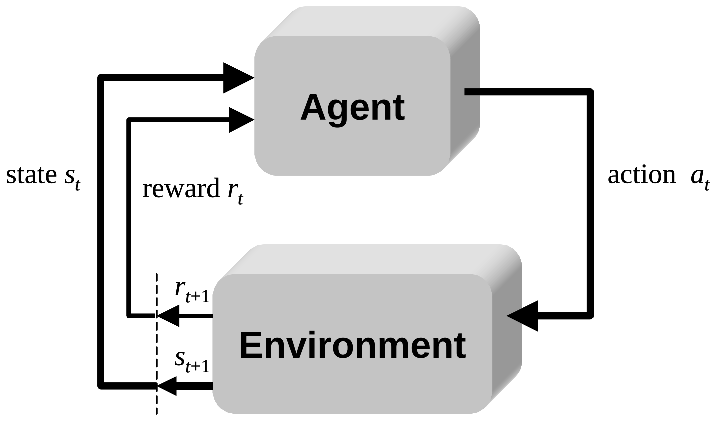


Figure 28. Agent-environment interactions in reinforcement learning [137]

The following example dialogue, taken from [77], presents an intuitive way of understanding the interaction between the agent and its environment. As this example shows, the environment may be non-deterministic, that is, when applied to the same state, the same action may produce different rewards and lead to different states (like action 2 in state 65).

Environment:	You are in state 65. You have 4 possible actions.
Agent:	I'll take action 2.
Environment:	You received a reinforcement of 7 units. You are now in state 15. You have 2 possible actions.
Agent:	I'll take action 1.
Environment:	You received a reinforcement of -4 units. You are now in state 65. You have 4 possible actions.
Agent:	I'll take action 2
Environment:	You received a reinforcement of 5 units. You are now in state 44. You have 5 possible actions
⋮	⋮

A policy π is a mapping from each state s ∈ S and action $a \in A(s)$ to the probability π(s, a) of selecting the action a when in state s. The agent's goal is choosing a policy that maximizes the expected *return* value, where the return R_t is some function of the reward sequence after time step t. For example, one could use the sum of the rewards until the final step T as return:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_T$$

For some problems, however, there is no clear final step, and the interaction between agent and environment continues infinitely. For these cases, one may use the *discounted return*, which is the infinite summation:

$$R_t = \sum_{k=0}^{\infty} g^k r_{t+k+1}$$

The parameter γ, $0 \le \gamma < 1$, is called *discount rate*. It can be seen not only as a mathematical trick to bound the sum (if the sequence $\{r_k\}$ is bounded), but also as a way to assign the importance of future rewards: values near to 1 puts more emphasis on future rewards than values near to 0. Discounted return can also be used for the finite case.

As we said, the agent must maximize the *expected* return value, since the environment may be non-deterministic. If (r_{t+1}, s_{t+1}) depends only on (s_t, a_t), that is, if one needs only the last state and an action to predict the possible outcome (reward and state transition), we say that the environment satisfies the Markov property and call it a Markov Decision Process, or MDP. In addition, if the number of possible states and actions is finite, we say that the environment is a finite MDP, for which much of the theoretical work in reinforcement learning has been written. The performance for a reinforcement learning system with the Markov property is better, and therefore many problems where the environment does not satisfy the Markov property are modeled as approximations of MDPs.

For an MDP, we will use $P_{ss'}^{a}$ to denote the probability of changing to state s' if action a is executed when in state s, that is:

$$P_{ss'}^{a} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}.$$

In the same way, $R_{ss'}^{a}$ is used to denote the expected reward of executing action a when in state s if the state changes to s':

$$R_{ss'}^{a} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\}.$$

We also define the state-value function $V^{\pi}(s)$ as the expected return when the environment is in state s and the agent follows then policy π, and we define the action-value function $Q^{\pi}(s, a)$ as the expected return when the environment is in state s, the agent takes action a, and then starts following policy π. Assuming discounted returns, and using $E_{\pi}\{\ \}$ to denote the expected value if the agent follows policy π, $V^{\pi}(s)$ and $Q^{\pi}(s, a)$ can be defined formally for MDPs as:

$$V^{p}(s) = E_{p}\{R_t \mid s_t = s\} = E_{p}\left\{\sum_{k=0}^{\infty} g^k r_{t+k+1} \,\middle|\, s_t = s\right\},$$

$$Q^p(s,a) = E_p\left\{R_t \middle| s_t = s, a_t = a\right\} = E_p\left\{\sum_{k=0}^{\infty} g^k r_{t+k+1} \middle| s_t = s, a_t = a\right\}.$$

We say that a policy π is better than or equal to a policy π' (denoted as π ≥ π') if $V^{\pi}(s) \geq V^{\pi'}(s)\ \forall s \in S$. There is at least one policy, called optimal policy, that is better than or equal to any other policy. All optimal polices are denoted π^*; they share the same state-value function V^* (called the optimal state-value function) and the same action-value function Q^* (called the optimal action-value function). V^* and Q^* are defined as:

$$V^*(s) = \max_p V^p(s), \forall s \in S \text{ and}$$

$$Q^*(s) = \max_p Q^p(s,a), \forall s \in S, a \in A(s).$$

It can also be proven [137] that:

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + gV^*(s')\right].$$

The last equation expresses how the values of a state and its successor states are related. In a finite MDP with *N* states, there will be *N* equations in *N* unknowns. Such an equation system, called Bellman optimality equation, has a unique solution. If $P^a_{ss'}$ and$R^a_{ss'}$ are know, one can in principle solve the system, determining V^* and, consequently, the optimal policy π^*: any policy that assigns a non-null probability to the actions that lead to the next best state.

An equation the express the relationship between the value of a state and the values of its successor states can also be derived for $Q^*(s, a)$:

$$Q^*(s,a) = \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + g \max_{a'} Q^*(s',a')\right].$$

Determining the optimal policy is even easier when the values for $Q^*(s, a)$ are known: for each state *s*, one just needs to select the action for which $Q^*(s, a)$ is maximal.

Although it may seem that the reinforcement learning problem is now solved, several assumptions made in this section limit the utility of the solution just presented. First, the number of states must be small enough, which is not true for many real-world problems (the game of checkers, for example, has an estimated number of 10^{18} legal states [125]). Second, the dynamics of the environment ($P^a_{ss'}$ and$R^a_{ss'}$) must be known. Finally, the environment satisfies the Markov property. "Reinforcement learning can be clearly understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions" [137].

8.1.3. Techniques for Solving Reinforcement Learning Problems

Many techniques have been proposed to solve reinforcement learning problems. In this section we show two of them, Sarsa [120, 143] and one-step Q-learning [155], which are the techniques we implemented in our framework for solving reinforcement learning problems (described in Appendix D). We also discuss policies for selecting actions.

Let s_t be the environment's state at step t, and a_t the action the agent selects. The action generates a reward r_{t+1} and changes the environment's state to s_{t+1} at the next time step, when the agent chooses the action a_{t+1}. The rule for updating $Q(s_t, a_t)$ using the Sarsa algorithm is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + a\left[r_{t+1} + l\,Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

while the rule for updating $Q(s_t, a_t)$ using one-step Q-learning is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + a\left[r_{t+1} + l \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

In both equations, a is a small constant step-size parameter and, under certain conditions, it can be shown that the learning action-value function Q does converge to optimal action-value function Q^* in both algorithms. Note that Q-learning does not depend on the action selected at step t+1 to update the learning action-value function.

The question left is: Which action should be selected at time step t, that is, which policy should the agent follow? A natural choice could be the *greedy* policy, which always selects the action with the largest expected reward. Always being greedy, however, may prevent the agent from exploring the environment and discovering actions that may produce better rewards. At the other extreme, we could choose a totally *random* policy, which does explore the environment but never exploits the knowledge acquired during the exploration.

One solution for the dilemma between exploiting and exploring is being most of the time greedy, but with a small probability ε selecting a random action. This policy, called ε-greedy policy, is popular but suffers from a problem: it explores all actions with equal probability, that is, the fact that some actions are more likely to bring good rewards than others is totally ignored during the exploration.

Another solution, called Softmax policy, ranks the actions according to their expected rewards, attributing higher probabilities to actions with higher expected rewards. If the Gibbs-Boltzmann distribution is used, we have that the probability of selection action a from the set A of possible actions when at state s is:

$$\Pr(a) = \frac{e^{Q(s,a)/t}}{\sum_{b \in A} e^{Q(s,b)/t}}$$

where τ is a parameter called temperature. Smaller values for the temperature increase the greediness of action selection, while higher values increase its randomness.

One convergence condition for the Sarsa algorithm is that the followed policy converges in the limit to the greedy distribution, while the convergence criteria for Q-learning is independent of the policy chosen. For this reason, Sarsa is called an *on-policy* algorithm, and Q-learning an *off-policy* algorithm.

8.2. Modeling Performance Analysis as a Reinforcement Learning Problem

Performance analysis can be seen as a reinforcement problem where the agent's goal is to find in a short time many performance problems and with as little as possible interference in the application's behavior. Because reinforcement learning is based on trial and error, it would take the agent too much time to learn the right actions for a given state if the performance analysis were post-mortem and the instrumentation static, since the reward would come only after the application finished executing. For this reason, we modeled only the dynamic performance analysis as a reinforcement learning problem.

As usual, the first challenge when modeling real-world problems is deciding which elements are significant when solving the problem and which are not. An excessive number of variables added to the problem definition may slow down the resolution: the agent has more signals to perceive and process, and it may take some time until the agent finally realizes that a variable has little or no significance for the problem. On the other hand, a model represents a type of biased knowledge, where the agent is told which signals can be safely ignored according to someone's point of view (points of view, however, are not always right). Another challenge is converting the result of actions to a scalar value that the agent can use as reinforcement, which may also contain a biased view of the problem and therefore will be transferred to the agent.

This section discusses how we modeled the problem of (dynamic) performance analysis and the rationale behind our model.

8.2.1. Definitions

The following definitions will be used when modeling performance analysis as a reinforcement learning problem. Because our focus was the analysis of Java programs, we will often use the terms *class*, *method*, and *method invocation*, although these terms could be substituted with similar ones from other program paradigms (like "module", "subroutine" and "subroutine call").

Definition 8.1. Let C be the set of classes used by an application, and $M(c)$ the set of methods of class $c \in C$. A *stack trace* of the application is a sequence $[m_1, m_2, \ldots, m_k]$ of method invocations, where $\forall j,\ 1 < j \leq k,\ m_j \in \bigcup_{c \in C} M(c)$ and m_{j-1} was invoked by method m_j.

At every instant of the application execution, each live thread has a non-empty stack trace. Let E be the set of all pairs (*stack trace*, *thread*) that can be observed during the application execution, and E_i the set of pairs that are observed at some instant i; for a single pair $(e = [m_1, m_2, \ldots, m_k], t) \in E_i$, and C and $M(c)$ as defined above, we have the following notations and definitions:

- $Methods(e) = \{m \mid \exists m,j\ (1 \leq j \leq k \wedge m = m_j\}$ are the methods in the stack trace e.
- $Classes(e) = \{c \mid \exists m(m \in M(c) \cap Methods(e))\}$ are the classes in the stack trace e.
- $Classes^*(E_i) = \bigcup_{(h,t) \in E_i} Classes(h)$ are the classes in all stack traces of E_i.
- The size of E_i is defined as $Size(E_i) = (\overline{x}, s)$, where $\overline{x}$ is the average length of the stack traces in E_i and s the standard deviation. We say that $Size(E_u) = (\overline{x}_u, s_u)$ overlaps $Size(E_v) = (\overline{x}_v, s_v)$ if and only if:

  $[\overline{x}_u - s_u, \overline{x}_u + s_u] \cap [\overline{x}_v - s_v, \overline{x}_v + s_v] \neq \varnothing$
- The average number of classes is defined as $NC(E_i) = (\overline{y}, r)$, where $\overline{y}$ is the average cardinality of $Classes^*(h)$ for all stack traces h in E_i and s the standard deviation. We say that $NC(E_u) = (\overline{y}_u, r_u)$ overlaps $NC(E_v) = (\overline{y}_v, r_v)$ if and only if:

$$[\overline{y_u} - r_u, \overline{y_u} + r_u] \cap [\overline{y_v} - r_v, \overline{y_v} + r_v] \neq \emptyset$$

Definition 8.2. Let *SummarySet* denote the set $\mathbb{N}\times\mathbb{N}\times\mathbb{R}^2\times\mathbb{R}^2$. Let *P* be the set of all processes that make up an application execution at instant *i*, and *E* and E_i as defined above. *StackSummary*: $E \rightarrow$ *SummarySet* is defined as
StackSummary$(E_i) = (|P|, |Classes^*(E_i)|, Size(E_i), NC(E_i))$

Moreover, we say that the stack traces observed at instant *u* and *v* are similar if $Size(E_u)$ overlaps $Size(E_v)$ and $NC(E_u)$ overlaps $NC(E_v)$.

A stack summary condenses important information about the stack traces at some instant *i*. As we will see later, an agent will be able to take a decision for a stack trace that has never been before based on a similar stack trace.

Definition 8.3. Let *StabilityLevel* $\subset \mathbb{N}$ be a finite set such that $0 \in$ *StabilityLevel*. The function *StackStability*: $\mathbb{N} \rightarrow$ *StabilityLevel* is defined as:
StackStability$(x) = y \Leftrightarrow x \geq y \wedge \forall w \in$ *Stability* $(w > y$ Þ $w > x)$

StackStability maps any natural number *x* to the highest value in *StabilityLevel* that is less than or equals to *x*. This function is used to describe the stability level of stack traces given the time the stack trace has not changed. For example, if *StabilityLevel* = {0, 15000, 30000, 60000, 120000} and the methods in the stack trace have not changed in the last 95000 milliseconds, *StackStability* will be 60000. Since asking for the stack traces introduces overhead in the application execution, we can use the fact that a stable stack trace is not likely to change in order to avoid the overhead of constantly asking for stack traces.

Definition 8.4. Let *r* be a probed code region, *T* a time interval, *C* the (estimated) time the probe execution adds to the execution time of *r*, and *K* an amount of time that is large enough to consider *C* as insignificant. Let *y* be the number of times *r* was executed during the time interval *T* and *x* the average execution time during the same interval, $T \geq x$. We define the probe effect *PE*: $\mathbb{N}\times\mathbb{N} \rightarrow [0,1]$ as:

$$PE(x,y) = \begin{cases} \frac{1}{T}\left(\frac{x-K}{C-K}\right)xy \text{ if } C \leq x \leq K \\ 0 \text{ if } x > K \\ \min\left(1, \frac{C}{T}\text{y}\right) \text{ if } x < C \end{cases}$$

The function *PE* models the interference of the probe in the execution of a code region: the shorter the execution time of *r* and the more often *r* is executed, the larger becomes the probe effect. It is easy to see that *y* cannot be greater than *T*/*x* and that $\forall x, y_1, y_2\ (y_2 > y_1 \Rightarrow f(x,y_2) \geq f(x,y_1))$. Therefore:

$$\min(PE(x,y)) = PE(x,0) = 0$$

$$\max\left(PE(x,y)\right)=\max\left(PE\left(x,\frac{T}{x}\right)\right)=\begin{cases}\frac{x-K}{C-K} \text{ if } C \le x \le K \\ 0 \text{ if } x > K \\ 1 \text{if } x < C \\ \end{cases}$$

Consequently, $0 \le PE(x,y) \le 1$.

If there is a probe b in a code region, we denote the execution time, as measured by the probe b, as *time*(b), and the number of times the code region (and the probe) was executed as *count*(b). Thus we can define the probe effect in terms of probes as

ProbeEffect(b) = *PE*(*time*(b),*count*(b))

Definition 8.5. Let c be an instrumentable code region in the application. If c is a method (and consequently is not nested in any instrumentable code region), we say that the level of c, denoted as *level*(c), is 0. Otherwise, *level*(c) = 1+*level*(p), where p is the code region where c is immediately nested. The set of code regions in a method m with level n is denoted as *CodeRegions*(m, n).

Using the definition of level, we can define the size and the depth of a method. Given a method m, its size, denoted as *size*(m), is the cardinality of the set *CodeRegions*(m, 1). The depth of m, denoted as *depth*(m), is the value g such that *CodeRegions*(m,g+1) is empty and *CodeRegions*(m,g) is not, that is, *depth*(m) is the maximum level among the code regions in the method. These definitions are useful when a method is instrumented, as we will see later.

Definition 8.6. Let B be the set of probes in an application, r a code region containing a probe, and $t = [t_0, t_1]$ a time interval. Let W_0 denote the properties found in r at time t_0, and W_1 the properties found in r at time t_1. *Inactivity*: $B \rightarrow \mathbb{N}$ is defined as:

$$\mathit{Inactivity}(b)=\begin{cases}0 \text{ if } W_0 = W_1 \\ t_1 - t_0 \text{ otherwise}\end{cases}$$

In other words, *Inactivity* measures for how much time the set of properties in a code region has not been updated.

8.2.2. States, Actions, and Rewards

Assume that the set *StackStability* (Definition 8.3) and the constants T, C, and K (Definition 8.4) are defined, and let *SummarySet* denote the set $\mathbb{N}\times\mathbb{N}\times\mathbb{R}^2\times\mathbb{R}^2$ (Definition 8.2). The set of states in the performance analysis problem is defined as:

S = *StackStability* ∪ *SummarySet* ∪{*RepentanceState*, *RemovalState*}

An element of *StackStability* will be called a *StabilityState*, and an element of *SummarySet* will be called a *SummaryState*.

The set of actions in the performance analysis problem is defined as:

A = { *Repent*, *DoNothing*, *RequestStackTrace* } ∪
{ *InstrumentCodeRegion*(s, d) | (s, d) ∈ $\mathbb{N}^2$ } ∪
{ *RemoveProbe*(t, f) | (t, f) ∈ $\mathbb{N}^2$ }

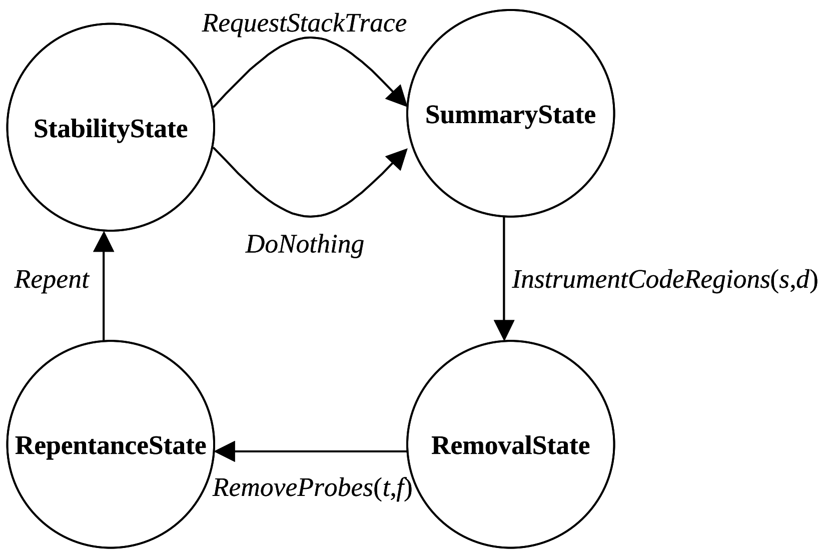


Figure 29. Transition graph for the performance analysis problem

The actions the agent chooses for every state are rewarded (or punished). In the following, we describe which actions are allowed for each state, the effect of that action in the environment, and the rewards received. We will use names starting with *REWARD_* and *PENALTY_* to describe positive and negative constants values used as rewards; concrete values for these constants, as well as for *StackStability* and the constants *T*, *C*, and *K* are shown in Section 9.4. The transition graph for the performance analysis problem is shown in Figure 29.

- *RequestStackTrace*

Action allowed when at some *StackStability* state. The stack traces of the application are retrieved, and *StackStability* and *StackSummary* are computed (the values will be used to determine the next *SummaryState* and *StabilityState*). The reward for this action is *PENALTY_STACK_REQUEST*, if the stack traces are equal the previous stack traces, or *PENALTY_STACK_REQUEST* + *REWARD_STACK_CHANGED*, if they are not. Note that the agent will be punished for having requested the stack traces (because the request introduces overhead in the execution), but the punishment will be reduced by a positive reward if the stack has changed, because the change can potentially bring more knowledge to the agent.

- *DoNothing*

Action allowed at any *StabilityState*, which just causes the transition to a *SummaryState* and has reward 0.

- *InstrumentCodeRegions*(*s*, *d*)

An action allowed in the *SumamryState* which instruments each code region *c* in each method *m* of the application if, and only if, *c* has never been instrumented and the both following conditions are true:

- *size*(*m*) ≥ *s* or *depth*(*m*) ≥ *d* or there exists a code region *c*' that invokes *m*, and a performance property was found in *c*'
- *level*(*c*) = 0, or a performance property was found in *p*, where *p* is the code region where *c* is immediately nested.

The environment keeps an internal representation *AppStruct* of the application structure (classes, methods, code regions). Before starting the instrumentation, the environment checks if *AppStruct* contains enough information about the methods found in the last stack trace retrieved, that is, if the code regions of each method in the stack trace are present in *AppStruct*. If not, then *AppStruct* is updated.

Both updating *AppStruct* and instrumenting the application are a source of overhead in the application execution (note that we are not talking yet about the instrumentation overhead, but about the overhead of inserting instrumentation). The reward for this action is *PENALTY_UPDATE_STRUCT* + PENALTY_INSTRUMENTATION if *AppStruct* had to be updated, or PENALTY_INSTRUMENTATION otherwise.

- *RemoveInstrumentation*(*i*,*f*)

Action allowed at the *RemovalState*, which removes any probe *b* in the application if any of the following conditions is true:

- $Inactivity(b) \geq i$, or
- $ProbeEffect(b) \geq f$

The reward for this action is *PENALTY_REMOVE_INSTRUMENTATION* if some probe was removed.

- *Repent*

Repent evaluates how good or bad the agent is doing its task, being the only action allowed at the *Repentance state*. For each code region *c* with probe *b*, let $G_{old}(b)$ be the set of performance properties that were found in *c* and which were already present in *c* the last time the action *Repent* was executed. Let $G_{new}(b)$ be the new properties found in *c* and which are also present in the code region where *c* is nested, and let $G^*_{new}(b)$ be the new properties which are not. *PropertyReward*(*b*) is defined as:

$$\begin{aligned} PropertyReward(b) = \\ & G_{old}(b) \times REWARD\_OLD\_PROPERTY + \\ & G_{new}(b) \times (REWARD\_NEW\_PROPERTY + extra) \\ & G^*_{new}(b) \times (REWARD\_NEW\_PROPERTY^* + extra) \end{aligned}$$

where

$$\begin{aligned} & REWARD\_NEW\_PROPERTY^* > \\ & REWARD\_NEW\_PROPERTY > \\ & REWARD\_OLD\_PROPERTY \end{aligned}$$

and *extra* = REWARD_EXPLANATION_FOUND if *c* has no nested code region, and 0 otherwise.

PropertyReward attributes a larger value to new properties than to old properties, and an even larger value if the property is not present in the parent code region. In a sense, *PropertyReward* measures the degree of “surprise” when a property is found: if a property is found in code region c, we expect to find it again the next time we look at *c*. It will also be no big surprise if the property is found in a code region nested in *c*. On the other hand, a property found in *c* which was not present in the parent of *c* appeared “out of nothing” and represents new possibilities of exploration. The *extra* argument is only an extra reward given to new properties found in a code region where the instrumentation cannot be refined, which means that the cause of a performance problem must be in that code region, and not in a nested code region.

Now, we define the utility of probe b as:

$$ProbeUtility(b) = PropertyReward(b) - \max\{PropertyReward(b), REWARD\_NEW\_PROPERTY^{*}\} \times ProbeEffect(b)$$

ProbeUtility uses the probe effect as a discount factor for the property reward. Note that, in the worst case, the probe effect can even nullify the value of the properties found. At the same time, *REWARD_NEW_PROPERTY** also acts as a negative reinforcement for code regions where few or no properties are found.

Finally, the reward of the *Repent* action is computed as

$$\frac{\sum_{b \in B} ProbeUtility(b)}{|B|}$$ where B is the set of all probes.

8.2.3. Similarities and Biases

A problem the agent must face while acting is what to do when a state that has never been seen before is encountered. Our generic framework for solving reinforcement learning problems (shown in Appendix D) provides support for *similar states*, which are states used when the agent has no basis to make an informed decision about the best action to choose. In this case, the agent will select the best action for one of the similar states (instead of choosing randomly an action), which is called *generalization*. The agent must know, however, what makes two states "similar."

Since the *RepentanceState* and the *RemovalState* are unique, we just need to consider *StabilityStates* and *SummaryStates*. Nevertheless, *StabilityStates* are themselves a measure of similarity, because they classify the time the stack has not changed as some "stability level" (see Definition 8.3), while *SummaryStates* already define the concept of similarity (using the average size of stack traces and the average number of classes in the stack). Therefore, it is straightforward to find similar states in the model we defined.

Sometimes, however, there is no similar state, and the agent must select an action without any information about the possible outcome. While this is not a problem in theoretical reinforcement learning, which under certain conditions usually guarantees the a solution is found provided that a state is visited infinitely often, in practice it is useful to integrate some knowledge in the agent that informs which actions are more likely to bring good rewards. This integrated knowledge is called a bias.

- Action *InstrumentCodeRegions*(s_0, d_0) is more likely to bring a good reward than action *InstrumentCodeRegions*(s_1, d_1) if either $s_0 > s_1$ and $d_0 >= d_1$ or $s_0 = s_1$ and $d_0 > d_1$.

  Rationale: Instrumenting small methods is likely to introduce a larger probe effect, so it makes sense to instrument first larger methods.

  If D and S are both bounded subsets of $\mathbb{N}$, and if *InstrumentCodeRegion*(s, d) is defined only if $(s, d) \in S \times D$, then a bias can alternatively (but not equivalently) be expressed as:

  $$bias(s,d) = \frac{(\max\{D\} - d) + (\max\{S\} - s)}{(\max\{D\} - \min\{D\}) + (\max\{S\} - \min\{S\})}$$

  We will say that *InstrumentCodeRegions*(s_0, d_0) is more likely to bring a good reward than *InstrumentCodeRegions*(s_1, $d1$) if $bias(s_0, d_0) > bias(s_1, d_1)$. This definition makes the biases of all *InstrumentCodeRegion* actions comparable to each other, allowing the creation of a totally ordered set of *InstrumentCodeRegions* actions.

- Action *RemoveInstrumentation*(t_0, f_0) is more likely to bring a good reward than action *RemoveInstrumentation* (t_1, f_1) if either $t_0 > t_1$ and $f_0 >= f_1$ or $t_0 = t_1$ and $f_0 > f_1$.

  Rationale: The longer the time a probe has been inactive, the less likely is finding a performance property in the probed code region, and the larger the probe effect is, the more likely a bad reward in the future is. Therefore, it makes sense first to remove probes that have been inactive for more time or those that introduced larger probe effects.

  If *RemoveInstrumentation*(i, f) is defined only if $(i, f) \in I \times F$, where I and F are bounded sets, then a totally ordered set of *RemoveInstrumentation* actions can be created in the same way as described for *InstrumentCodeRegion* actions.

8.3. Summary

In this chapter, we provided a background on reinforcement learning and demonstrated how it can be used to model the performance analysis problem in order to provide a foundation to the decisions taken during the analysis process. We also defined formally some concepts used in performance analysis, like the probe effect or the state of a stack trace.

Reinforcement learning greatly simplifies the design and implementation of a performance analysis tool, as one just needs to define actions, without having to define the “search algorithm” itself, which is any of the algorithms for solving the reinforcement learning problem. This simplicity makes it easier to add new actions. Moreover, values for many parameters that must be manually adjusted by the user in the traditional analysis algorithm (like minimum size a method must have to be instrumented) are dynamically adjusted over the search process through the trial-and-error mechanism of reinforcement learning.

On the other hand, reinforcement learning introduces a few other parameters that must be set by the user, like the constants α and λ (see Section 8.1.3). Another drawback is that all theoretical guarantees in reinforcement learning are asymptotically true: States must be visited and actions must be executed an infinite number of times. Therefore, in order to speed up the reinforcement learning process, it is necessary to introduce biases that indicate the paths likely to lead faster to the agent’s goal.

In Section 9.4, we show experimental results that confirm the utility of our approach.

9

Experiments

In this chapter, we demonstrate the efficacy of our approach for detecting performance problems with seven different applications, three written in Fortran and four written in Java:

- a material science kernel (see Section 9.1.1);
- a photonic application (see Section 9.1.2);
- a financial application (see Section 9.1.3);
- a distributed graph algorithm (see Section 9.2);
- a distributed backtracking framework (see Section 9.3);
- a game (see Section 9.4.1);
- a distributed database (see Section 9.4.2).

Details about each application and the execution environment are described under the respective sections. We used Aksum to carry out the performance analysis and either SCALEA (for Fortran applications) or Twilight (for Java applications) to instrument the code and to collect performance metrics during the application's execution. We used the performance properties described in Section 5.3, except for those that had no support of the respective instrumentation tool. For example, neither instrumentation tool can measure overhead due to creation of threads, therefore we did not use the property *ThreadInitializationOverhead*,

9.1. Static Analysis of Fortran Applications

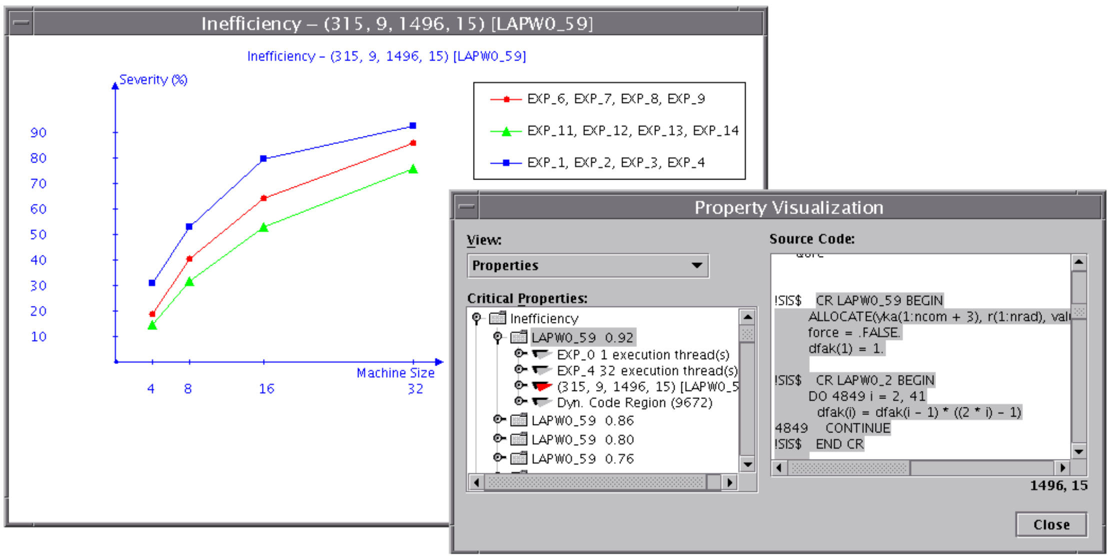
This section presents the performance analysis for three Fortran applications; the experiments were all conducted on an SMP cluster with 16 quad nodes connected through Fast Ethernet. Aksum itself was executed on a Sun workstation Ultra 1/170.

9.1.1. LAPW0 Material Science Application

LAPW0 [10] calculates the effective potential of the Kohn-Sham eigen-value problem. Implemented as a Fortran 90 MPI code, it has been examined with four problem sizes (representing 8, 16, 32 and 64 atoms) and five machine configurations (1, 4, 8, 16 and 32 CPUs). Due to a lack of memory, however, the last problem size could not be executed with only one CPU. Each line shown in the charts of this section refers to the execution of LAPW0 for a single problem size and different machine sizes.

For all problem sizes, the code inefficiency increases for larger machine sizes (see Figure 30), and this is the most critical aspect of LAPW0 (which Aksum indicates by placing the property *Inefficiency* at the top of the critical property list, as shown in Figure 31). The highest values for this property, however, appear only in the main program, which suggests that no code region is alone responsible for the inefficient behavior of LAPW0. In fact, Aksum shows that the causes for the inefficiency cannot be explained by any property in particular, but it lies in the combination of several properties:

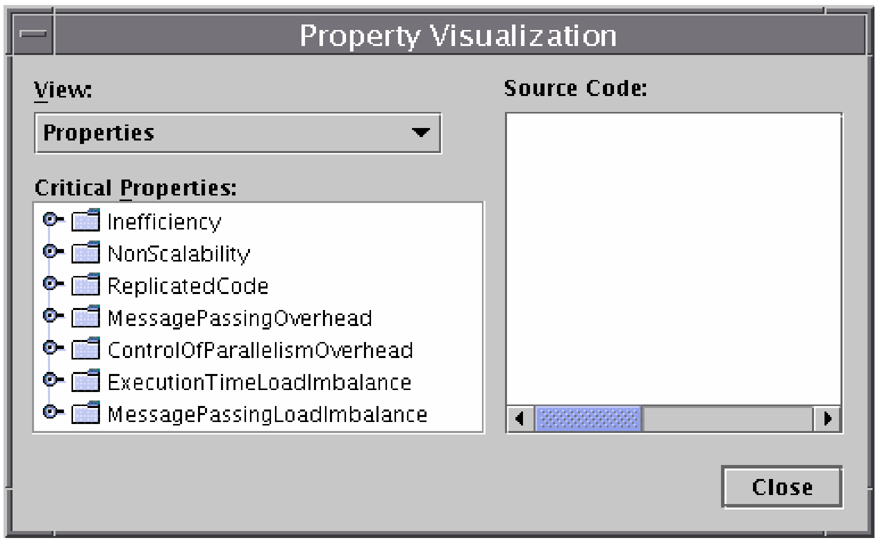
- *ReplicatedCode* (Figure 32)
  LAPW0 contains several replicated code regions that are not parallelized and executed by all CPUs. The cumulative effect of these regions (LAPW0_35, LAPW0_10, LAPW0_38, LAPW0_15 and LAPW0_47) stands out in the main program (LAPW0_59), responding for about 25% of the execution time.
- *MessagePassingOverhead* (Figure 33)
  This severity of this property also increases with larger machine sizes and, since message passing is cumulative, the worse values always appear in the main program.
- *ControlOfParallelismOverhead* (Figure 34)
  Again, a cumulative property whose value stands out in the main program.
- *ExecutionTimeLoadImbalance* (Figure 34)
  The code distributes a set of atoms onto the processes, but for the problem sizes tested it is not always possible to distribute them equally onto a set of CPUs. Besides the main program (LAPW0_59), the code regions most affected by this distribution are LAPW0_28, LAPW0_25, LAPW0_40 and LAPW0_43.


Inefficiency – (315, 9, 1496, 15) [LAPW0_59]
Inefficiency – (315, 9, 1496, 15) [LAPW0_59]
Severity (%)
90
80
70
60
50
40
30
20
10
Machine Size
4
8
16
32
EXP_6, EXP_7, EXP_8, EXP_9
EXP_11, EXP_12, EXP_13, EXP_14
EXP_1, EXP_2, EXP_3, EXP_4
Property Visualization
View:
Properties
Critical Properties:
Source Code:
1496, 15
Close

```
Inefficiency
  LAPW0_59  0.92
    EXP_0 1 execution thread(s)
    EXP_4 32 execution thread(s)
    (315, 9, 1496, 15) [LAPW0_5
    Dyn. Code Region (9672)
  LAPW0_59  0.86
  LAPW0_59  0.80
  LAPW0_59  0.76
```

```
!SIS$   CR LAPW0_59 BEGIN
       ALLOCATE(yka(1:ncom + 3), r(1:nrad), valu
       force = .FALSE.
       dfak(1) = 1.

!SIS$   CR LAPW0_2 BEGIN
       DO 4849 i = 2, 41
         dfak(i) = dfak(i - 1) * ((2 * i) - 1)
4849    CONTINUE
!SIS$   END CR
```

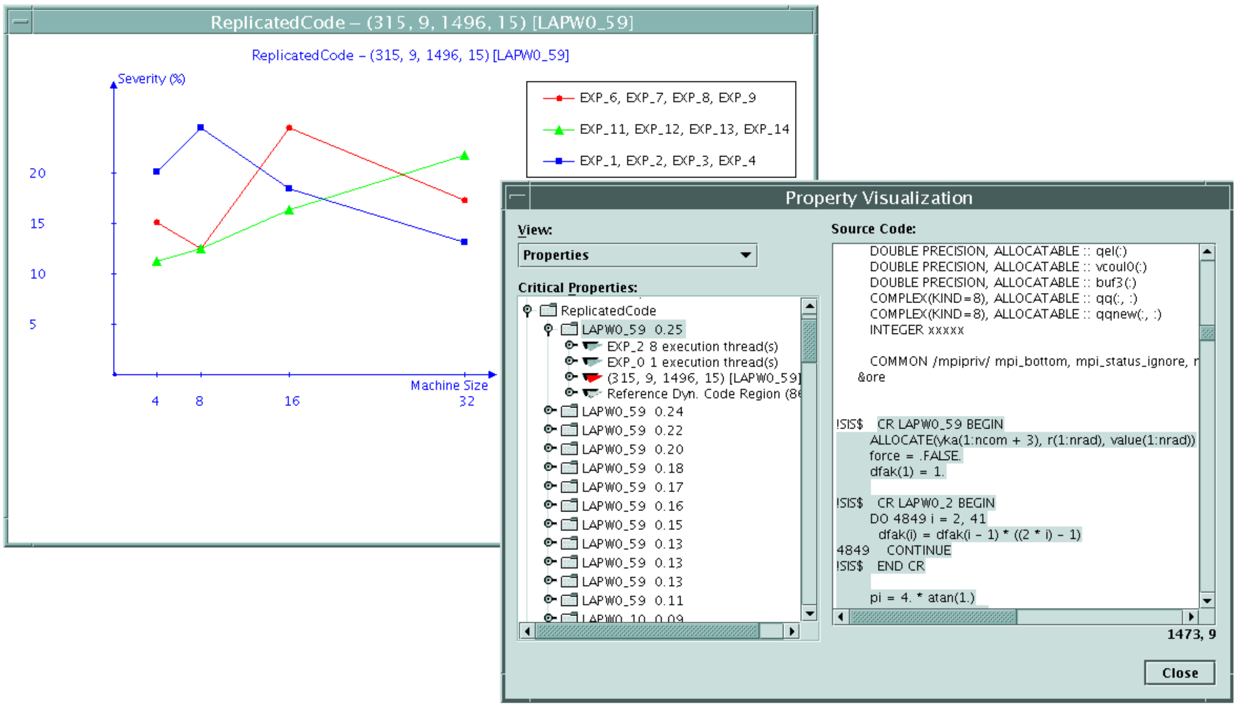
Figure 30. Inefficiency in LAPW0



Critical Properties:

- Inefficiency
- NonScalability
- ReplicatedCode
- MessagePassingOverhead
- ControlOfParallelismOverhead
- ExecutionTimeLoadImbalance
- MessagePassingLoadImbalance

Figure 31. Properties found in LAPW0

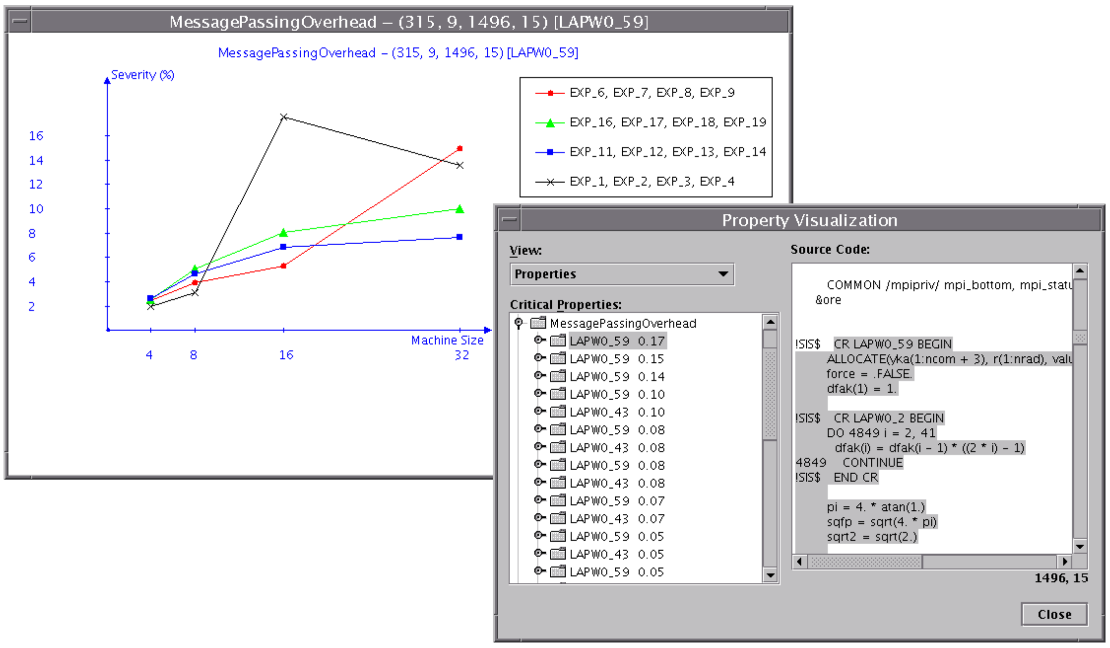

ReplicatedCode – (315, 9, 1496, 15) [LAPW0_59]
ReplicatedCode – (315, 9, 1496, 15) [LAPW0_59]
Severity (%)
20
15
10
5
4
8
16
32
Machine Size
EXP_6, EXP_7, EXP_8, EXP_9
EXP_11, EXP_12, EXP_13, EXP_14
EXP_1, EXP_2, EXP_3, EXP_4
Property Visualization
View:
Properties
Source Code:
1473, 9
Close

Critical Properties:

```
ReplicatedCode
  LAPW0_59  0.25
    EXP_2 8 execution thread(s)
    EXP_0 1 execution thread(s)
    (315, 9, 1496, 15) [LAPW0_59]
    Reference Dyn. Code Region (8
  LAPW0_59  0.24
  LAPW0_59  0.22
  LAPW0_59  0.20
  LAPW0_59  0.18
  LAPW0_59  0.17
  LAPW0_59  0.16
  LAPW0_59  0.15
  LAPW0_59  0.13
  LAPW0_59  0.13
  LAPW0_59  0.13
  LAPW0_59  0.11
  LAPW0_10  0.09
```

```
      DOUBLE PRECISION, ALLOCATABLE :: qel(:)
      DOUBLE PRECISION, ALLOCATABLE :: vcoul0(:)
      DOUBLE PRECISION, ALLOCATABLE :: buf3(:)
      COMPLEX(KIND=8), ALLOCATABLE :: qq(:, :)
      COMPLEX(KIND=8), ALLOCATABLE :: qqnew(:, :)
      INTEGER xxxxx

      COMMON /mpipriv/ mpi_bottom, mpi_status_ignore, r
     &ore


!SIS$  CR LAPW0_59 BEGIN
      ALLOCATE(yka(1:ncom + 3), r(1:nrad), value(1:nrad))
      force = .FALSE.
      dfak(1) = 1.

!SIS$  CR LAPW0_2 BEGIN
      DO 4849 i = 2, 41
        dfak(i) = dfak(i - 1) * ((2 * i) - 1)
4849    CONTINUE
!SIS$  END CR

      pi = 4. * atan(1.)
```

Figure 32. ReplicatedCode property in LAPW0


MessagePassingOverhead – (315, 9, 1496, 15) [LAPW0_59]
MessagePassingOverhead – (315, 9, 1496, 15) [LAPW0_59]
Severity (%)
16
14
12
10
8
6
4
2
4
8
16
32
Machine Size
EXP_6, EXP_7, EXP_8, EXP_9
EXP_16, EXP_17, EXP_18, EXP_19
EXP_11, EXP_12, EXP_13, EXP_14
EXP_1, EXP_2, EXP_3, EXP_4
Property Visualization
View:
Properties
Source Code:
Close

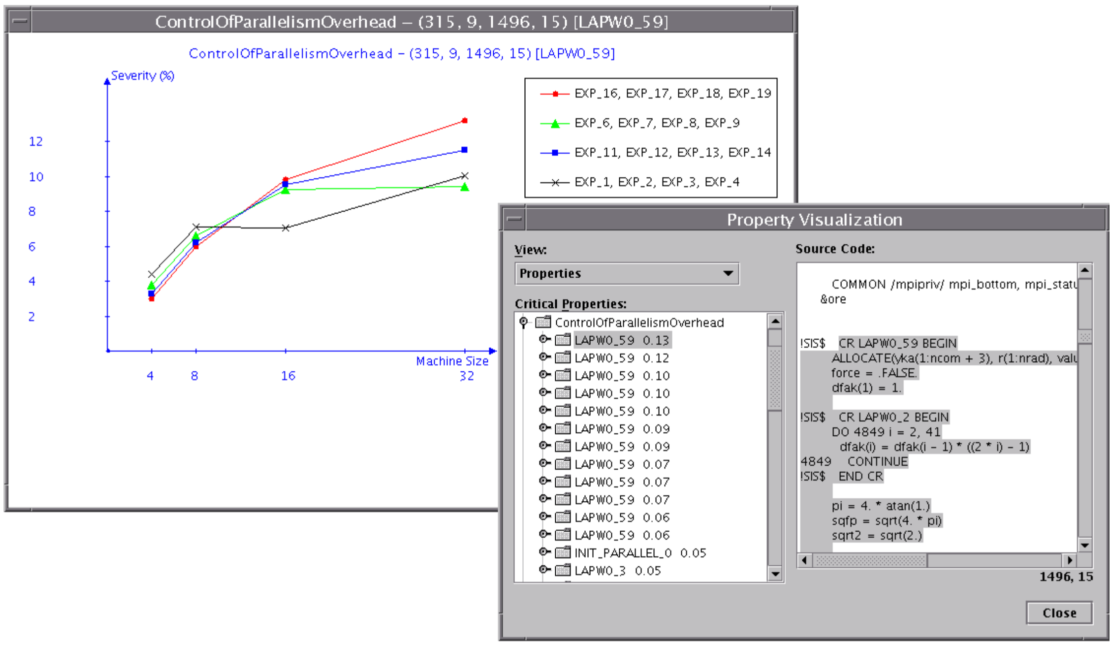
Critical Properties:

```
MessagePassingOverhead
  LAPW0_59  0.17
  LAPW0_59  0.15
  LAPW0_59  0.14
  LAPW0_59  0.10
  LAPW0_43  0.10
  LAPW0_59  0.08
  LAPW0_43  0.08
  LAPW0_59  0.08
  LAPW0_43  0.08
  LAPW0_59  0.07
  LAPW0_43  0.07
  LAPW0_59  0.05
  LAPW0_43  0.05
  LAPW0_59  0.05
```

```
      COMMON /mpipriv/ mpi_bottom, mpi_statu
     &ore

!SIS$   CR LAPW0_59 BEGIN
      ALLOCATE(yka(1:ncom + 3), r(1:nrad), valu
      force = .FALSE.
      dfak(1) = 1.

!SIS$   CR LAPW0_2 BEGIN
      DO 4849 i = 2, 41
        dfak(i) = dfak(i - 1) * ((2 * i) - 1)
4849    CONTINUE
!SIS$   END CR

      pi = 4. * atan(1.)
      sqfp = sqrt(4. * pi)
      sqrt2 = sqrt(2.)
```

1496, 15

Figure 33. Overhead caused by message passing in LAPW0


ControlOfParallelismOverhead – (315, 9, 1496, 15) [LAPW0_59]
ControlOfParallelismOverhead – (315, 9, 1496, 15) [LAPW0_59]
Severity (%)
12
10
8
6
4
2
4
8
16
32
Machine Size
EXP_16, EXP_17, EXP_18, EXP_19
EXP_6, EXP_7, EXP_8, EXP_9
EXP_11, EXP_12, EXP_13, EXP_14
EXP_1, EXP_2, EXP_3, EXP_4
Property Visualization
View:
Properties
Source Code:
Close

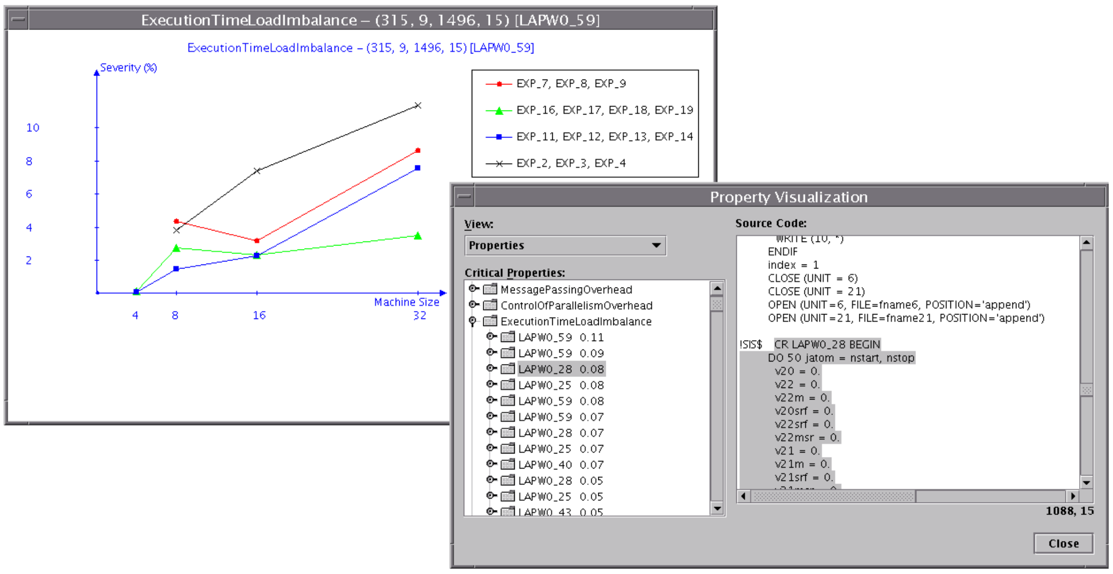
Critical Properties:

```
ControlOfParallelismOverhead
  LAPW0_59  0.13
  LAPW0_59  0.12
  LAPW0_59  0.10
  LAPW0_59  0.10
  LAPW0_59  0.10
  LAPW0_59  0.09
  LAPW0_59  0.09
  LAPW0_59  0.07
  LAPW0_59  0.07
  LAPW0_59  0.07
  LAPW0_59  0.06
  LAPW0_59  0.06
  INIT_PARALLEL_0  0.05
  LAPW0_3  0.05
```

```
      COMMON /mpipriv/ mpi_bottom, mpi_statu
     &ore

!SIS$   CR LAPW0_59 BEGIN
      ALLOCATE(yka(1:ncom + 3), r(1:nrad), valu
      force = .FALSE.
      dfak(1) = 1.

!SIS$   CR LAPW0_2 BEGIN
      DO 4849 i = 2, 41
        dfak(i) = dfak(i - 1) * ((2 * i) - 1)
4849    CONTINUE
!SIS$   END CR

      pi = 4. * atan(1.)
      sqfp = sqrt(4. * pi)
      sqrt2 = sqrt(2.)
```

1496, 15

Figure 34. Overhead caused by control of parallelism in LAPW0


ExecutionTimeLoadImbalance – (315, 9, 1496, 15) [LAPW0_59]
ExecutionTimeLoadImbalance – (315, 9, 1496, 15) [LAPW0_59]
Severity (%)
10
8
6
4
2
4
8
16
32
Machine Size
EXP_7, EXP_8, EXP_9
EXP_16, EXP_17, EXP_18, EXP_19
EXP_11, EXP_12, EXP_13, EXP_14
EXP_2, EXP_3, EXP_4
Property Visualization
View:
Properties
Critical Properties:
Source Code:
1088, 15
Close

```
MessagePassingOverhead
ControlOfParallelismOverhead
ExecutionTimeLoadImbalance
    LAPW0_59  0.11
    LAPW0_59  0.09
    LAPW0_28  0.08
    LAPW0_25  0.08
    LAPW0_59  0.08
    LAPW0_59  0.07
    LAPW0_28  0.07
    LAPW0_25  0.07
    LAPW0_40  0.07
    LAPW0_28  0.05
    LAPW0_25  0.05
    LAPW0_43  0.05
```

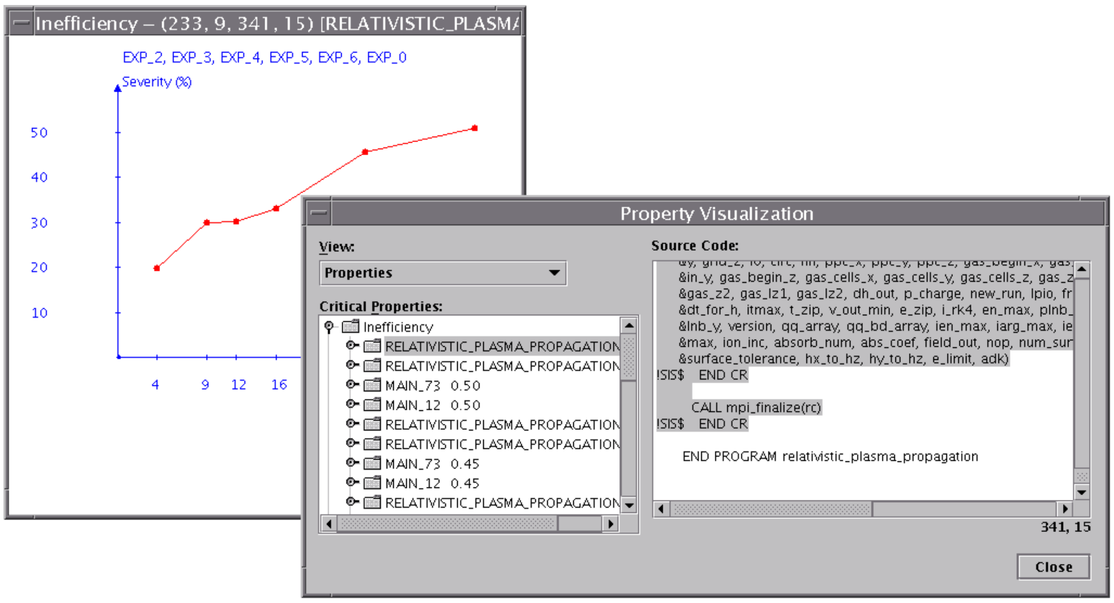
```
          WRITE (10, *)
        ENDIF
        index = 1
        CLOSE (UNIT = 6)
        CLOSE (UNIT = 21)
        OPEN (UNIT=6, FILE=fname6, POSITION='append')
        OPEN (UNIT=21, FILE=fname21, POSITION='append')

!SIS$   CR LAPW0_28 BEGIN
        DO 50 jatom = nstart, nstop
          v20 = 0.
          v22 = 0.
          v22m = 0.
          v20srf = 0.
          v22srf = 0.
          v22msr = 0.
          v21 = 0.
          v21m = 0.
          v21srf = 0.
```

Figure 35. **ExecutionTimeLoadImbalance property in LAPW0**

9.1.2. 3D-PIC

The 3D-Particle-In-Cell [48] is an application written in Fortran90 and MPI that simulates the ultrashort laser-plasma interaction in a three dimensional geometry. It can presently run with seven different problem sizes (1, 4, 9, 12, 16, 25 and 36 CPUs).

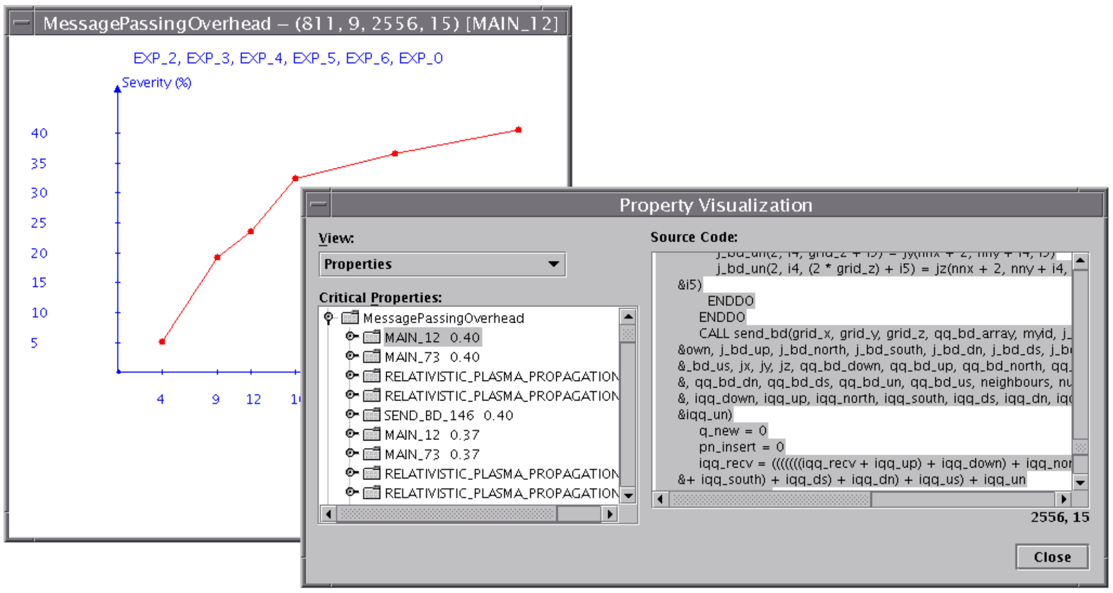
Aksum’s analysis show that the inefficiency of this code tends to grow for larger problem sizes (Figure 36), which is caused mainly by the high message passing overhead in subroutine SEND_BD, invoked from within the loop marked as MAIN_12 (Figure 37). Aksum also directs the user to the most time-consuming MPI call (Figure 38).



```
Inefficiency
    RELATIVISTIC_PLASMA_PROPAGATION
    RELATIVISTIC_PLASMA_PROPAGATION
    MAIN_73  0.50
    MAIN_12  0.50
    RELATIVISTIC_PLASMA_PROPAGATION
    RELATIVISTIC_PLASMA_PROPAGATION
    MAIN_73  0.45
    MAIN_12  0.45
    RELATIVISTIC_PLASMA_PROPAGATION
```

```
     &in_y, gas_begin_z, gas_cells_x, gas_cells_y, gas_cells_z, gas_z
     &gas_z2, gas_lz1, gas_lz2, dh_out, p_charge, new_run, lpio, fr
     &dt_for_h, itmax, t_zip, v_out_min, e_zip, i_rk4, en_max, plnb_
     &lnb_y, version, qq_array, qq_bd_array, ien_max, iarg_max, ie
     &max, ion_inc, absorb_num, abs_coef, field_out, nop, num_sur
     &surface_tolerance, hx_to_hz, hy_to_hz, e_limit, adk)
!SIS$   END CR

        CALL mpi_finalize(rc)
!SIS$   END CR

      END PROGRAM relativistic_plasma_propagation
```

Figure 36. **Inefficiency in 3D-PIC**

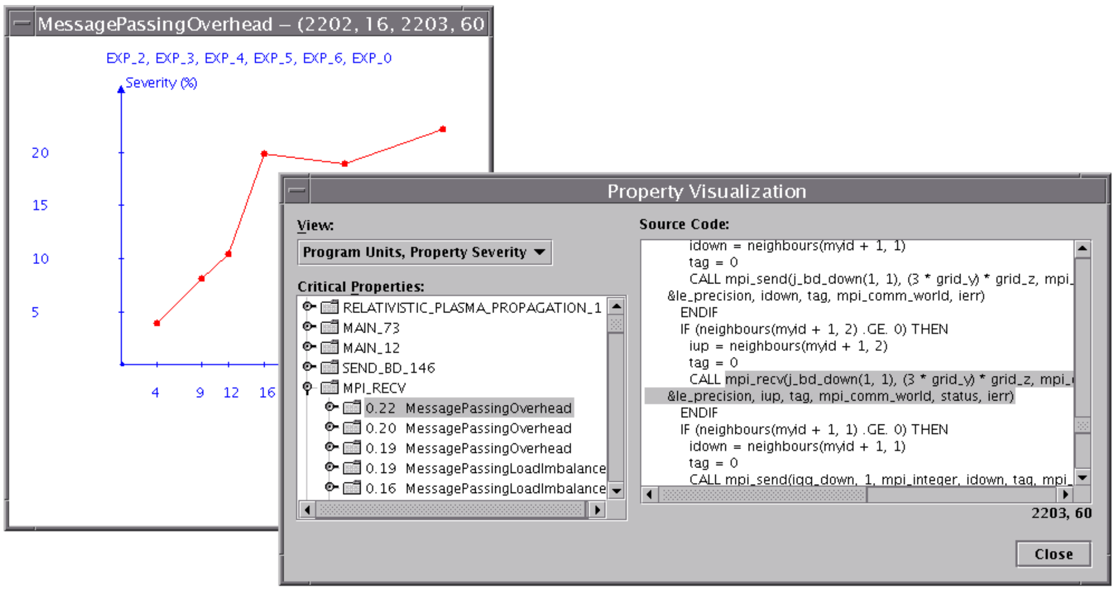


```
MessagePassingOverhead
  MAIN_12  0.40
  MAIN_73  0.40
  RELATIVISTIC_PLASMA_PROPAGATION
  RELATIVISTIC_PLASMA_PROPAGATION
  SEND_BD_146  0.40
  MAIN_12  0.37
  MAIN_73  0.37
  RELATIVISTIC_PLASMA_PROPAGATION
  RELATIVISTIC_PLASMA_PROPAGATION
```

```
          j_bd_un(2, i4, (2 * grid_z) + i5) = jz(nnx + 2, nny + i4,
&i5)
        ENDDO
      ENDDO
      CALL send_bd(grid_x, grid_y, grid_z, qq_bd_array, myid, j_
&own, j_bd_up, j_bd_north, j_bd_south, j_bd_dn, j_bd_ds, j_b
&_bd_us, jx, jy, jz, qq_bd_down, qq_bd_up, qq_bd_north, qq_
&, qq_bd_dn, qq_bd_ds, qq_bd_un, qq_bd_us, neighbours, nu
&, iqq_down, iqq_up, iqq_north, iqq_south, iqq_ds, iqq_dn, iq
&iqq_un)
      q_new = 0
      pn_insert = 0
      iqq_recv = (((((((iqq_recv + iqq_up) + iqq_down) + iqq_nor
&+ iqq_south) + iqq_ds) + iqq_dn) + iqq_us) + iqq_un
```

2556, 15

Figure 37. Overhead caused by message passing in 3D-PIC



MessagePassingOverhead – (2202, 16, 2203, 60
EXP_2, EXP_3, EXP_4, EXP_5, EXP_6, EXP_0
Severity (%)
20
15
10
5
4
9
12
16
Property Visualization
View:
Program Units, Property Severity
Critical Properties:
Source Code:
Close

```
RELATIVISTIC_PLASMA_PROPAGATION_1
MAIN_73
MAIN_12
SEND_BD_146
MPI_RECV
  0.22  MessagePassingOverhead
  0.20  MessagePassingOverhead
  0.19  MessagePassingOverhead
  0.19  MessagePassingLoadImbalance
  0.16  MessagePassingLoadImbalance
```

```
        idown = neighbours(myid + 1, 1)
        tag = 0
        CALL mpi_send(j_bd_down(1, 1), (3 * grid_y) * grid_z, mpi_
&le_precision, idown, tag, mpi_comm_world, ierr)
      ENDIF
      IF (neighbours(myid + 1, 2) .GE. 0) THEN
        iup = neighbours(myid + 1, 2)
        tag = 0
        CALL mpi_recv(j_bd_down(1, 1), (3 * grid_y) * grid_z, mpi_
&le_precision, iup, tag, mpi_comm_world, status, ierr)
      ENDIF
      IF (neighbours(myid + 1, 1) .GE. 0) THEN
        idown = neighbours(myid + 1, 1)
        tag = 0
        CALL mpi_send(iqq_down, 1, mpi_integer, idown, tag, mpi_
```

2203, 60

Figure 38. Most time consuming MPI call in 3D-PIC

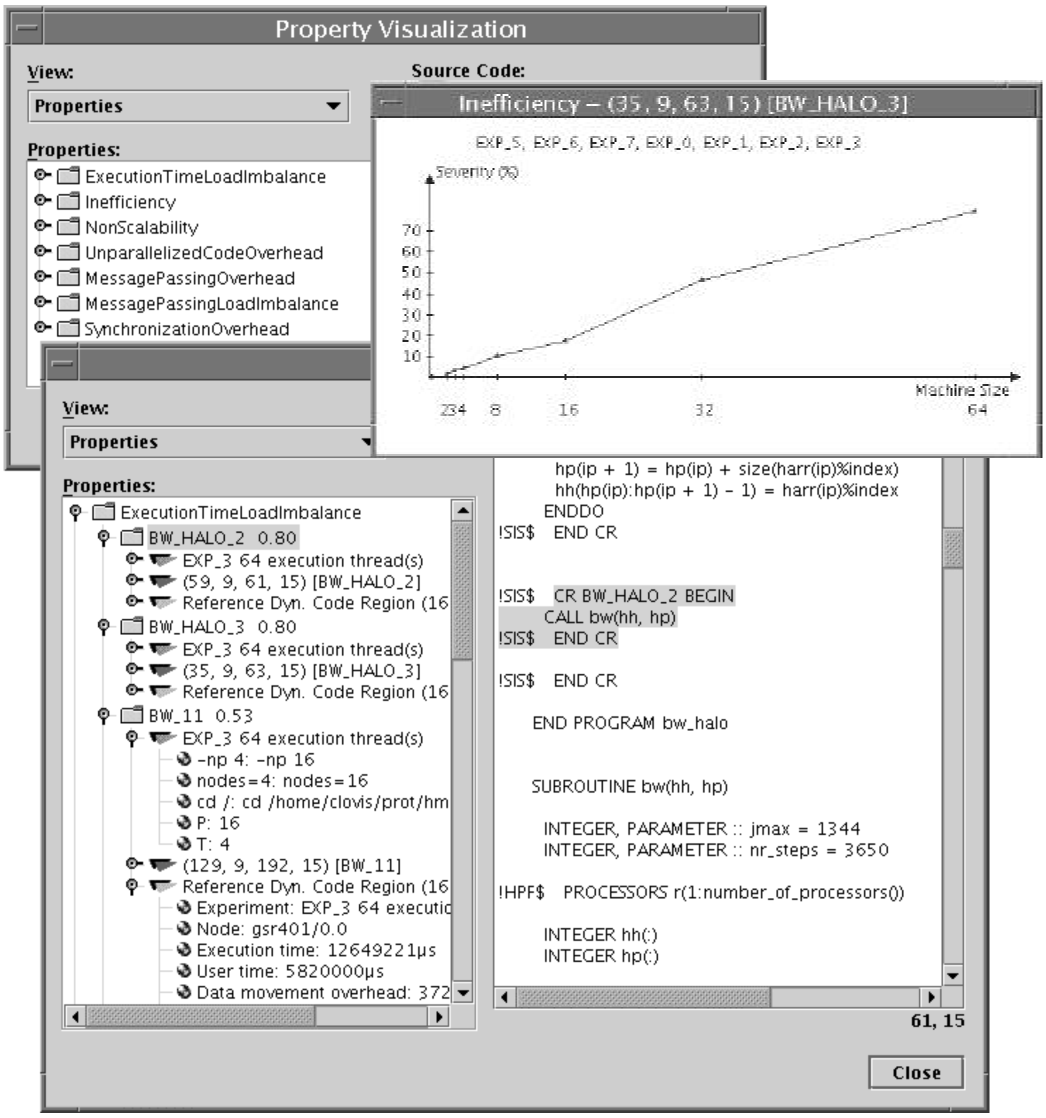
9.1.3. Backward pricing

The backward pricing application implements the backward induction algorithm to compute the price of an interest rate dependent product [29]. The backward induction algorithm has been implemented as an HPF code based on which the VFC compiler [8] generates a mixed OpenMP/MPI code.

Based on the user provided input data, the search engine of Aksum automatically determines that seven performance properties in the property hierarchy are critical for this code (see Figure 39), where the properties are presented in ascending order of severity. As usual, the user portal displays initially only the property names for those instances whose severity is above the user-defined threshold (we set it to 0.01). The property instances can be shown by expanding each property name. For every instance the corresponding program unit and severity value is indicated. In the backward pricing application, the most serious performance property is

ExecutionTimeLoadImbalance, which has an instance that holds for the main (entire) program with severity value 0.80 (see the entry BW_HALO_3 0.80). The same property holds for the sub-region of the main program indicated by the entry BW_HALO_2 0.80.

The severity of the *ExecutionTimeLoadImbalance* property instances for the entire application increases with the number of execution threads (not shown in Figure 39), from 0.01 for 2 CPUs to 0.80 for 64 CPUs. This behavior also explains the increasing severity values for the *Inefficiency* property (varying from 0.05 for 2 CPUs to 0.79 for 64 CPUs – see the Inefficiency diagram in the upper right window of Figure 39) and the poor scaling behavior (the severity of property *NonScalability* is 0.56 for the main program). All other properties in the property hierarchy have lower severity values (*SynchronizationOverhead*: 0.01, *MessagePassingOverhead*: 0.17 with 64 CPUs, for the other machine sizes 0.00).



```
ExecutionTimeLoadImbalance
Inefficiency
NonScalability
UnparallelizedCodeOverhead
MessagePassingOverhead
MessagePassingLoadImbalance
SynchronizationOverhead
```

```
ExecutionTimeLoadImbalance
  BW_HALO_2  0.80
    EXP_3 64 execution thread(s)
    (59, 9, 61, 15) [BW_HALO_2]
    Reference Dyn. Code Region (16
  BW_HALO_3  0.80
    EXP_3 64 execution thread(s)
    (35, 9, 63, 15) [BW_HALO_3]
    Reference Dyn. Code Region (16
  BW_11  0.53
    EXP_3 64 execution thread(s)
      -np 4: -np 16
      nodes=4: nodes=16
      cd /: cd /home/clovis/prot/hm
      P: 16
      T: 4
    (129, 9, 192, 15) [BW_11]
    Reference Dyn. Code Region (16
      Experiment: EXP_3 64 executic
      Node: gsr401/0.0
      Execution time: 12649221µs
      User time: 5820000µs
      Data movement overhead: 372
```

```
        hp(ip + 1) = hp(ip) + size(harr(ip)%index)
        hh(hp(ip):hp(ip + 1) - 1) = harr(ip)%index
      ENDDO
!SIS$   END CR


!SIS$   CR BW_HALO_2 BEGIN
      CALL bw(hh, hp)
!SIS$   END CR

!SIS$   END CR

     END PROGRAM bw_halo


     SUBROUTINE bw(hh, hp)

      INTEGER, PARAMETER :: jmax = 1344
      INTEGER, PARAMETER :: nr_steps = 3650

!HPF$   PROCESSORS r(1:number_of_processors())

      INTEGER hh(:)
      INTEGER hp(:)
```

Figure 39. Snapshots of Aksum's property visualization for the Backward Pricing application

The main program calls the subroutine BW, which calls subroutine COMPUTE_SLICE. As the properties Inefficiency and *NonScalability* are not critical for COMPUTE_SLICE, and since the critical instances of these properties have always approximately the same value for both the main program and the subroutine BW, we conclude that performance tuning should mainly concentrated on subroutine BW.

9.2. Static Analysis of a Java Application

We conducted a variety of experiments with All-Pairs Shortest Path, an application that finds the length of the shortest path between every pair of vertices of a graph. It was built using DPPEJ, the IBM framework for distributed Java applications [28]. DPPEJ is built on the top of RMI; its API "provides conceptually similar functionality to the one-sided communication APIs available in the Messaging Passing Interface (MPI) standard."

The experiments are a combination of different machine and problem sizes. For machine sizes, we used 1, 4, 7, 9, and 11 DPPEJ daemons (a DPPEJ daemon is one thread running on one JVM). For problem sizes, we chose 500, 1000, and 1500 vertices.

Code and data are transferred to each daemon at the beginning of the execution using the DPPEJ API (which uses RMI). Each daemon runs the same code with a subset of the data, and one daemon combines the solutions found in each daemon, generating the global solution.

The experiments were executed using a heterogeneous set of nodes:

- Four Pentium III Xeon nodes (operating system: Linux);
- Three Sun Blade 1500 workstations (operating system: Solaris);
- One AMD Athlon PC (operating system: Windows 2000);
- One Pentium 4 notebook (operating system: Windows XP);
- Two Sun Blade 150 workstations (operating system: Solaris).

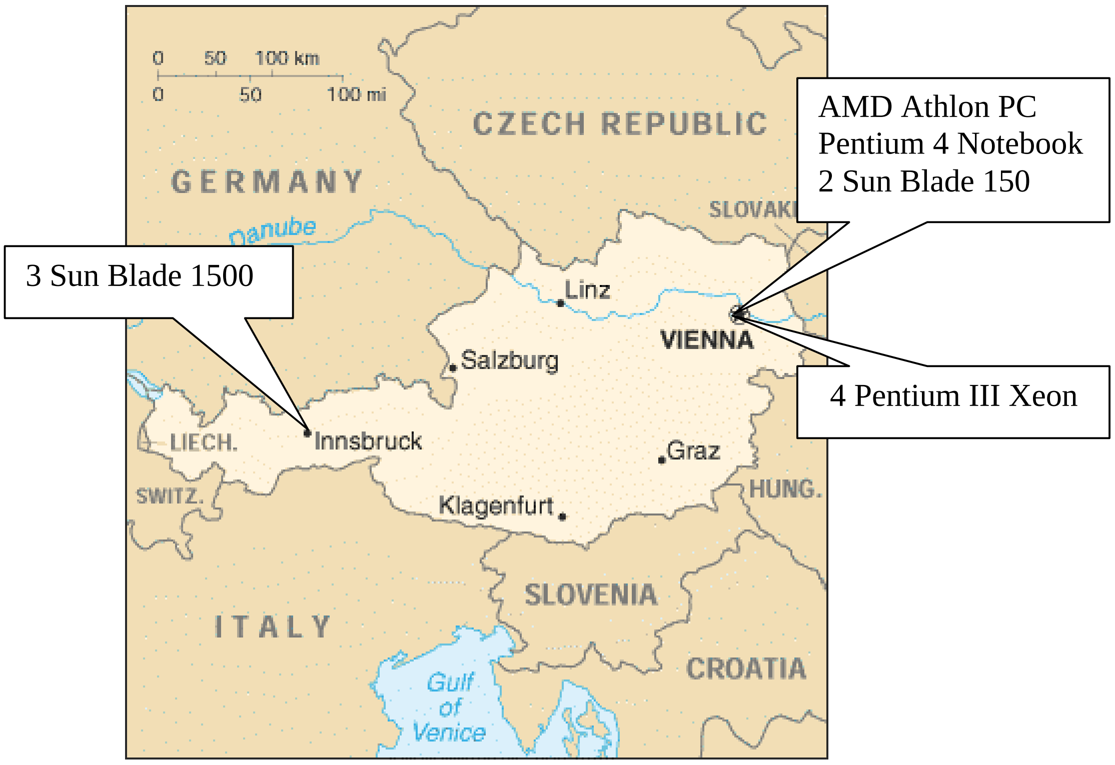


Figure 40. Nodes and sites used when analyzing DPPEJ

The Sun Blade 1500 workstations are located in the city of Innsbruck, Austria, while the other nodes are located in Vienna. These two cities are approximately 600 kilometers apart. In Vienna, the Linux nodes make up one site, and the remaining machines another site. This is represented in Figure 40.

Figure 41 shows the problematic code regions that Aksum found. Each code region is displayed with its type (e.g. LOOP represents while, do… while, and for loops, and TRY represents the try…catch…finally constructs), the name of the source code where the code region is (the source code itself is also displayed, although Figure 41 does not show it), and the position of the code region in the source (start line, start column, end line, end column). The last three numbers correspond to the minimum, average, and maximum severity values for the performance properties found in that code region. Under each code region, Aksum shows the performance properties found and, under each property, the problem sizes used in the experiment where the performance property was found. Finally, below each problem size, the experiments that used that problem size are shown, where the three last numbers always indicate the minimum, average, and maximum values for the severity values in the nodes below in the tree. The nodes in the output tree can be reorganized so as to show, for instance, first the properties, then the experiments, and then the code regions. Note also that the number of threads shown for each experiment corresponds to the DPPEJ daemons plus the main thread (the one that only distributes the data).

The code region BLOCK in the file ASPTest.java corresponds to the main program. As this region only waits for the end of the computation, it can be ignored.

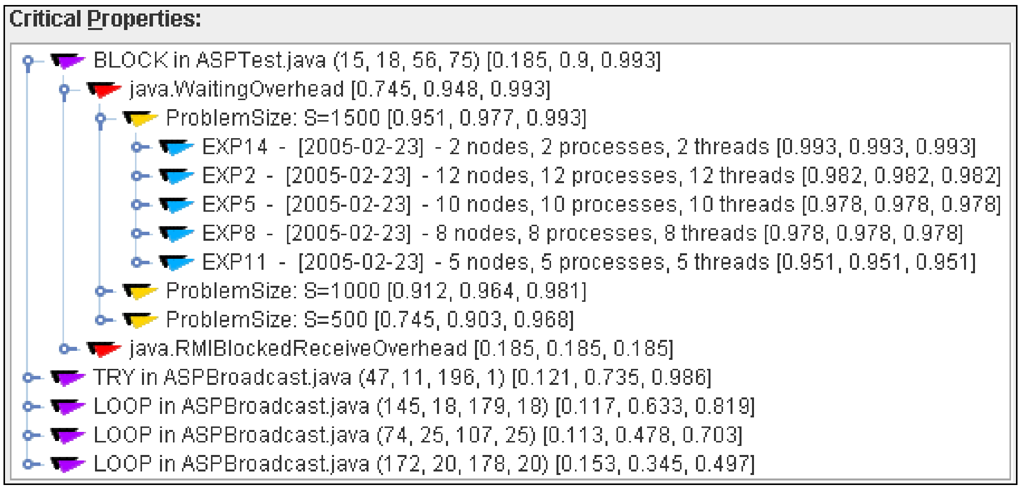

```
Critical Properties:
BLOCK in ASPTest.java (15, 18, 56, 75) [0.185, 0.9, 0.993]
  java.WaitingOverhead [0.745, 0.948, 0.993]
    ProblemSize: S=1500 [0.951, 0.977, 0.993]
      EXP14 - [2005-02-23] - 2 nodes, 2 processes, 2 threads [0.993, 0.993, 0.993]
      EXP2 - [2005-02-23] - 12 nodes, 12 processes, 12 threads [0.982, 0.982, 0.982]
      EXP5 - [2005-02-23] - 10 nodes, 10 processes, 10 threads [0.978, 0.978, 0.978]
      EXP8 - [2005-02-23] - 8 nodes, 8 processes, 8 threads [0.978, 0.978, 0.978]
      EXP11 - [2005-02-23] - 5 nodes, 5 processes, 5 threads [0.951, 0.951, 0.951]
    ProblemSize: S=1000 [0.912, 0.964, 0.981]
    ProblemSize: S=500 [0.745, 0.903, 0.968]
  java.RMIBlockedReceiveOverhead [0.185, 0.185, 0.185]
TRY in ASPBroadcast.java (47, 11, 196, 1) [0.121, 0.735, 0.986]
LOOP in ASPBroadcast.java (145, 18, 179, 18) [0.117, 0.633, 0.819]
LOOP in ASPBroadcast.java (74, 25, 107, 25) [0.113, 0.478, 0.703]
LOOP in ASPBroadcast.java (172, 20, 178, 20) [0.153, 0.345, 0.497]
```

Figure 41. Properties found

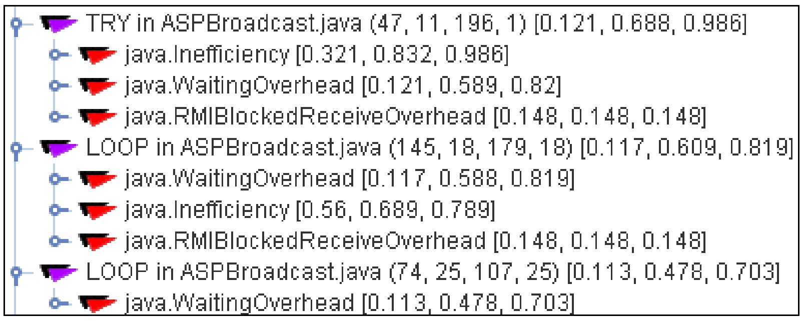

```
TRY in ASPBroadcast.java (47, 11, 196, 1) [0.121, 0.688, 0.986]
  java.Inefficiency [0.321, 0.832, 0.986]
  java.WaitingOverhead [0.121, 0.589, 0.82]
  java.RMIBlockedReceiveOverhead [0.148, 0.148, 0.148]
LOOP in ASPBroadcast.java (145, 18, 179, 18) [0.117, 0.609, 0.819]
  java.WaitingOverhead [0.117, 0.588, 0.819]
  java.Inefficiency [0.56, 0.689, 0.789]
  java.RMIBlockedReceiveOverhead [0.148, 0.148, 0.148]
LOOP in ASPBroadcast.java (74, 25, 107, 25) [0.113, 0.478, 0.703]
  java.WaitingOverhead [0.113, 0.478, 0.703]
```

Figure 42. Properties found - details

The performance properties found in the code region TRY in the file ASPBroadcast.java are detailed in Figure 42: *Inefficiency*, *WaitingOverhead*, and *RMIBlockedOverhead*. Based on that we can conclude:

- Adding more nodes does not speed up the execution of the application very much (property *Inefficiency*). In particular, high values like 0.986 indicate that the parallel execution time is almost the same as the sequential execution time.
- The time spent waiting for something done in other threads is significant (property *WaitingOverhead*).
- Some time is also spent blocked waiting for the result of remote method invocations (property *RMIBlockedReceiveOverhead*).

The loop at (145, 18, 179, 18) is nested in the code region TRY described above. By inspecting the source code (not shown), it is possible to see that this loop is executed by only one DPPEJ daemon, which is responsible for collecting the output of the other DPPEJ daemons. Figure 42 shows that the values for the severity are near those values in the TRY region, which means that this loop is the real culprit.

The loop at (74, 25, 107, 25) is executed by all DPPEJ daemons except one. The property *RMIBlockedReceiveOverhead* was not found in this loop, which reflects the fact that this loop does not wait for the computation done in other daemons, as the previous loop does. The property *Inefficiency* was not found, as we set it to compare experiments only with the sequential version (and the sequential version does not execute this loop, only the previous one).

Finally, the properties in the loop at (172, 20, 178, 20), which is also nested in the code region TRY, are shown in details in Figure 43. This loop is inefficient, and the property *LoadImbalance*, also found in this loop, indicates that the reason for the inefficiency is a poor work distribution. Note that the loop is inefficient only for the experiments using eleven DPPEJ daemons; these are exactly the only experiments that use the slower nodes. In fact, by inspecting the source code, we see that the distribution of data among the DPPEJ daemons is static and does not take into account the speed of the nodes processing the computations (that is, slow nodes receive the same amount of data as fast nodes).

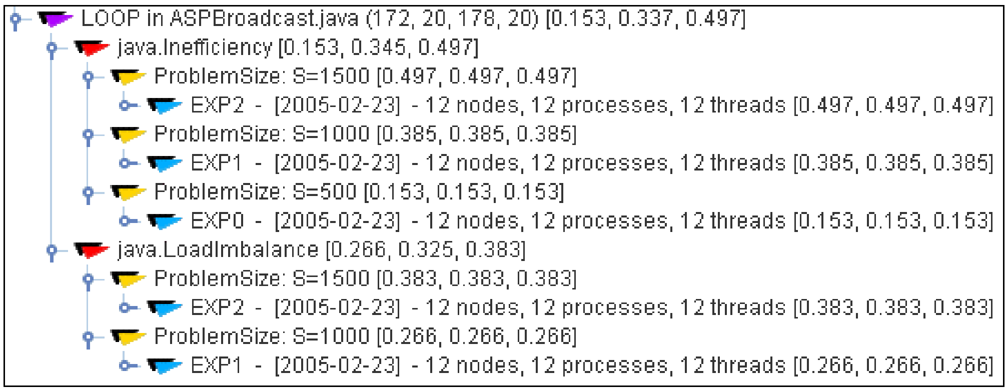

```
LOOP in ASPBroadcast.java (172, 20, 178, 20) [0.153, 0.337, 0.497]
    java.Inefficiency [0.153, 0.345, 0.497]
        ProblemSize: S=1500 [0.497, 0.497, 0.497]
            EXP2 - [2005-02-23] - 12 nodes, 12 processes, 12 threads [0.497, 0.497, 0.497]
        ProblemSize: S=1000 [0.385, 0.385, 0.385]
            EXP1 - [2005-02-23] - 12 nodes, 12 processes, 12 threads [0.385, 0.385, 0.385]
        ProblemSize: S=500 [0.153, 0.153, 0.153]
            EXP0 - [2005-02-23] - 12 nodes, 12 processes, 12 threads [0.153, 0.153, 0.153]
    java.LoadImbalance [0.266, 0.325, 0.383]
        ProblemSize: S=1500 [0.383, 0.383, 0.383]
            EXP2 - [2005-02-23] - 12 nodes, 12 processes, 12 threads [0.383, 0.383, 0.383]
        ProblemSize: S=1000 [0.266, 0.266, 0.266]
            EXP1 - [2005-02-23] - 12 nodes, 12 processes, 12 threads [0.266, 0.266, 0.266]
```

Figure 43. Properties found - details

The instrumentation overhead is very reasonable. For a problem size of 1500 vertices, the instrumentation overhead using the four Pentium nodes was 34%, and for the entire set of nodes 11%.

9.3. Dynamic Analysis of a Java Application

We analyzed a distributed application built using JavaSymphony, a “high-level Java-based programming paradigm for parallel and distributed systems … built as a Java class library that allows the user to control parallelism, load balancing and locality at a high level. The communication between two different machines is built on the top of RMI.” [76] In the application analyzed, one must find all different possibilities to move a number of *n* identical objects along *m* consecutive positions. The problem is solved using a backtracking algorithm where the sub-problems obtained at various steps are dynamically distributed among the nodes used in the application, that is, idle nodes receive additional sub-problems from busy nodes. Details about the problem and the backtracking algorithm can be found in [41].

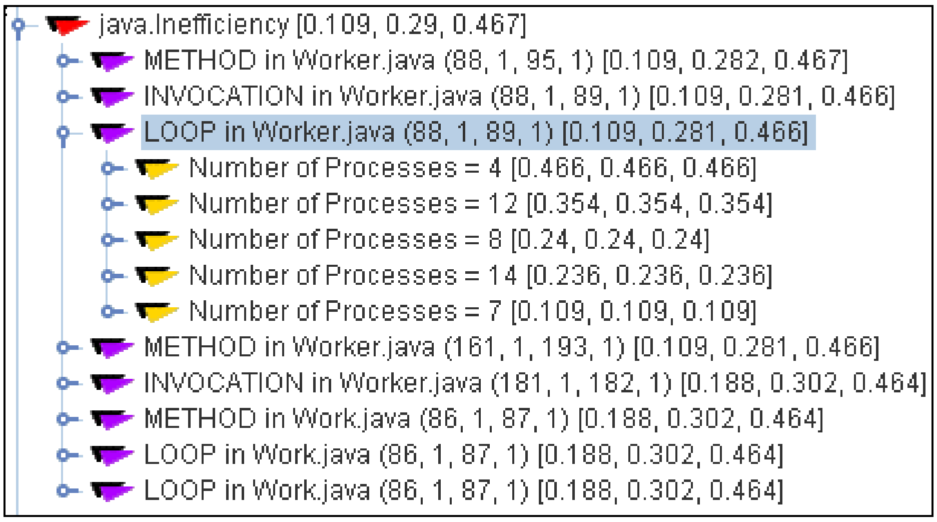
The experiments were executed using a heterogeneous set of nodes:

- Four Sun Blade 1500 workstations (operating system: Solaris);
- Three Sun Blade 1500 workstations (operating system: Solaris);
- One AMD Athlon PC (operating system: Windows 2000);
- Four Pentium III Xeon nodes (operating system: Linux);
- Two Sun Blade 150 workstations (operating system: Solaris).

The second set of Sun Blade 1500 workstations is located in the city of Innsbruck, Austria, while the other nodes are located in Vienna. The experiments were executed with four, seven, eight, twelve and fourteen nodes, added in the same order as listed above. The problem size chosen has five objects and six positions, which yields 701149020 solutions. The application has long-running methods that are executed only once, so Aksum had to run in non-refining mode. All communication between Twilight and Aksum was compressed, and a proxy between Aksum and some of the nodes (the workstations in Innsbruck and the Pentium III nodes) was responsible for forwarding request and response documents through firewalls.

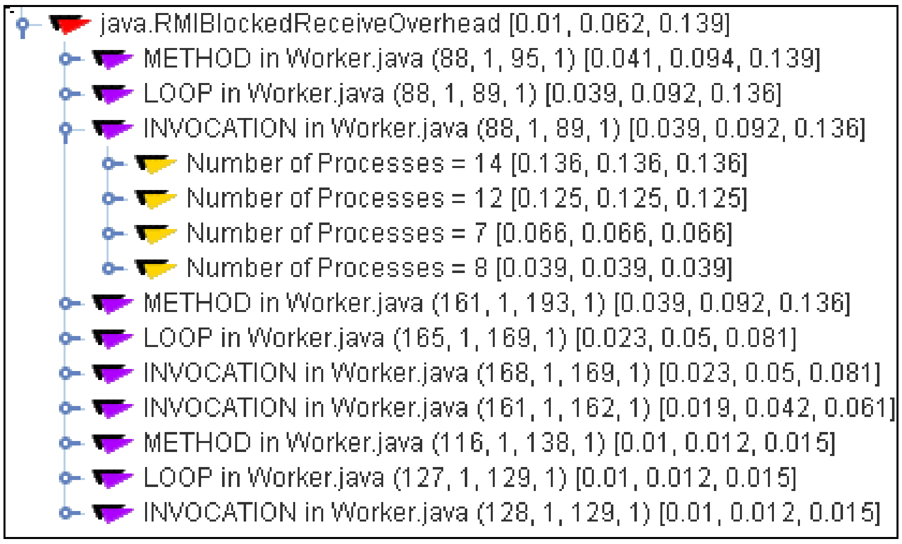
The first performance problem we noticed was the presence of unidentified overhead in several code regions. Unidentified overhead is a property in the repository of standard properties that computes the difference between the wall clock time and the sum of CPU time, waiting time (as defined by the metric THREAD_WAITED_TIME in Section 6.4), and time blocked in network operations. For one of the code regions this was a curious result, considering that this code region should just wait until all the nodes finish their tasks, that is, the wall clock time and the value of the metric THREAD_WAITED_TIME should be almost the same and no unidentified overhead should be detected. Looking at the code, we located a bug in that code region that caused the thread executing it to spin in a loop instead of leaving the CPU. This bug had a negative impact on the execution time, because the node where this code region was executed also had to solve part of the problem. By fixing the bug, the unidentified overhead disappeared in several regions (those that were actively computing the solutions for the problem) and changed to waiting overhead in the others (those that were waiting the answer computed in other nodes), and the execution time of the application decreased accordingly; in the sequential case, for example, it was cut by half, from 20.6 minutes to 10.1 minutes in a Sun Blade 1500 workstation.

Further analysis showed that the application has an inefficiency problem, that is, adding more nodes does not yield the expected performance gain, although this problem tends to decrease as the number of nodes added increase. Although the work is distributed to the nodes dynamically, we detected that the main cause for the inefficiency is the load imbalance. A small but noticeable time blocked waiting for the answer to a remote method invocation (property RMIBlockedReceive) was also detected for larger number of nodes. Figure 44, Figure 45, and Figure 46. show the properties found: *Inefficiency*, *RMIBlockedReceiveOverhread*, and *LoadImbalance*.



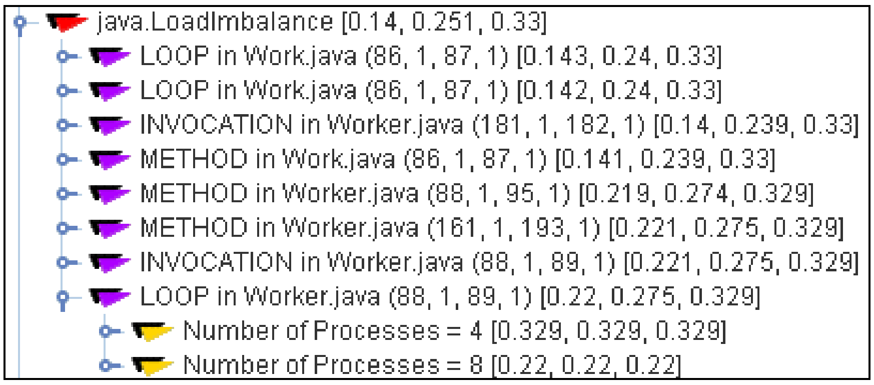
```
java.Inefficiency [0.109, 0.29, 0.467]
  METHOD in Worker.java (88, 1, 95, 1) [0.109, 0.282, 0.467]
  INVOCATION in Worker.java (88, 1, 89, 1) [0.109, 0.281, 0.466]
  LOOP in Worker.java (88, 1, 89, 1) [0.109, 0.281, 0.466]
    Number of Processes = 4 [0.466, 0.466, 0.466]
    Number of Processes = 12 [0.354, 0.354, 0.354]
    Number of Processes = 8 [0.24, 0.24, 0.24]
    Number of Processes = 14 [0.236, 0.236, 0.236]
    Number of Processes = 7 [0.109, 0.109, 0.109]
  METHOD in Worker.java (161, 1, 193, 1) [0.109, 0.281, 0.466]
  INVOCATION in Worker.java (181, 1, 182, 1) [0.188, 0.302, 0.464]
  METHOD in Work.java (86, 1, 87, 1) [0.188, 0.302, 0.464]
  LOOP in Work.java (86, 1, 87, 1) [0.188, 0.302, 0.464]
  LOOP in Work.java (86, 1, 87, 1) [0.188, 0.302, 0.464]
```

Figure 44. Inefficiency in the code analyzed



```
java.RMIBlockedReceiveOverhead [0.01, 0.062, 0.139]
  METHOD in Worker.java (88, 1, 95, 1) [0.041, 0.094, 0.139]
  LOOP in Worker.java (88, 1, 89, 1) [0.039, 0.092, 0.136]
  INVOCATION in Worker.java (88, 1, 89, 1) [0.039, 0.092, 0.136]
    Number of Processes = 14 [0.136, 0.136, 0.136]
    Number of Processes = 12 [0.125, 0.125, 0.125]
    Number of Processes = 7 [0.066, 0.066, 0.066]
    Number of Processes = 8 [0.039, 0.039, 0.039]
  METHOD in Worker.java (161, 1, 193, 1) [0.039, 0.092, 0.136]
  LOOP in Worker.java (165, 1, 169, 1) [0.023, 0.05, 0.081]
  INVOCATION in Worker.java (168, 1, 169, 1) [0.023, 0.05, 0.081]
  INVOCATION in Worker.java (161, 1, 162, 1) [0.019, 0.042, 0.061]
  METHOD in Worker.java (116, 1, 138, 1) [0.01, 0.012, 0.015]
  LOOP in Worker.java (127, 1, 129, 1) [0.01, 0.012, 0.015]
  INVOCATION in Worker.java (128, 1, 129, 1) [0.01, 0.012, 0.015]
```

Figure 45. Overhead caused by RMI



```
java.LoadImbalance [0.14, 0.251, 0.33]
  LOOP in Work.java (86, 1, 87, 1) [0.143, 0.24, 0.33]
  LOOP in Work.java (86, 1, 87, 1) [0.142, 0.24, 0.33]
  INVOCATION in Worker.java (181, 1, 182, 1) [0.14, 0.239, 0.33]
  METHOD in Work.java (86, 1, 87, 1) [0.141, 0.239, 0.33]
  METHOD in Worker.java (88, 1, 95, 1) [0.219, 0.274, 0.329]
  METHOD in Worker.java (161, 1, 193, 1) [0.221, 0.275, 0.329]
  INVOCATION in Worker.java (88, 1, 89, 1) [0.221, 0.275, 0.329]
  LOOP in Worker.java (88, 1, 89, 1) [0.22, 0.275, 0.329]
    Number of Processes = 4 [0.329, 0.329, 0.329]
    Number of Processes = 8 [0.22, 0.22, 0.22]
```

Figure 46. Load imbalance in the code analyzed

9.4. Dynamic Analysis Using a Learning Agent

We analyzed two applications using the reinforcement learning approach described in Chapter 8. The constant values used are shown in Table 6.

Defined in	*Name*	*Value*
Section 8.1.3	Policy	ε-greedy
	ε	0.1
	Agent algorithm	Sarsa
	γ	0.9
	α	0.1
Section 8.2.1, Definition 8.3	StabilityLevel	{ 2000, 5000, 10000, 30000, 60000 };
Section 8.2.1, Definition 8.4	T	1500 ms
	C	1 ms
	K	40 ms
Section 8.2.2	PENALTY_STACK_REQUEST	-3
	REWARD_STACK_CHANGED	+2
	PENALTY_UPDATE_STRUCT	-5
	PENALTY_INSTRUMENTATION	-5
	PENALTY_REMOVE_INSTRUMENTATION	-5
	REWARD_OLD_PROPERTY	+5
	REWARD_NEW_PROPERTY	+10
	REWARD_NEW_PROPERTY*	+15
	REWARD_EXPLANATION_FOUND	+5
Section 8.2.3	S	{1,2,3,4,5,6}
	D	{1,2,3,4,5}
	I	{45 s, 75 s, 105 s}
	F	{0.35,0.55,0.75}

Table 6. Parameter values for the performance analysis using a reinforcement learning agent

9.4.1. Tempest 1000

Tempest 1000 [24] is a highly interactive Java applet with several objects moving on the screen. This applet, shown in Figure 48, could not be analyzed using our traditional technique for dynamic analysis, because the perturbation created by the probes inserted rendered the game irresponsive. In contrast, using the reinforcement learning agent, the speed (or at least the perceived speed, which is what is important in an interactive application) did not change except for occasional, short freezes, which became less and less often as the run went on.

The source code of Tempest 1000 is not available, so we decided to include also the AWT API in the analysis, which is responsible, among others, for painting graphics and images and for processing events (like mouse and window events). The AWT API has a public and a private part, the public part being composed of the packages *java.awt.**, and the private of vendor-specific packages, both of which available for Sun’s Java distribution.

We ran two experiments: The first used only one node, running both the code and acting as I/O device, while the second experiment (shown in Figure 47) used two nodes, one running the game code and the other acting as a graphic display where input events were also generated. Aksum was always executed on a different node.

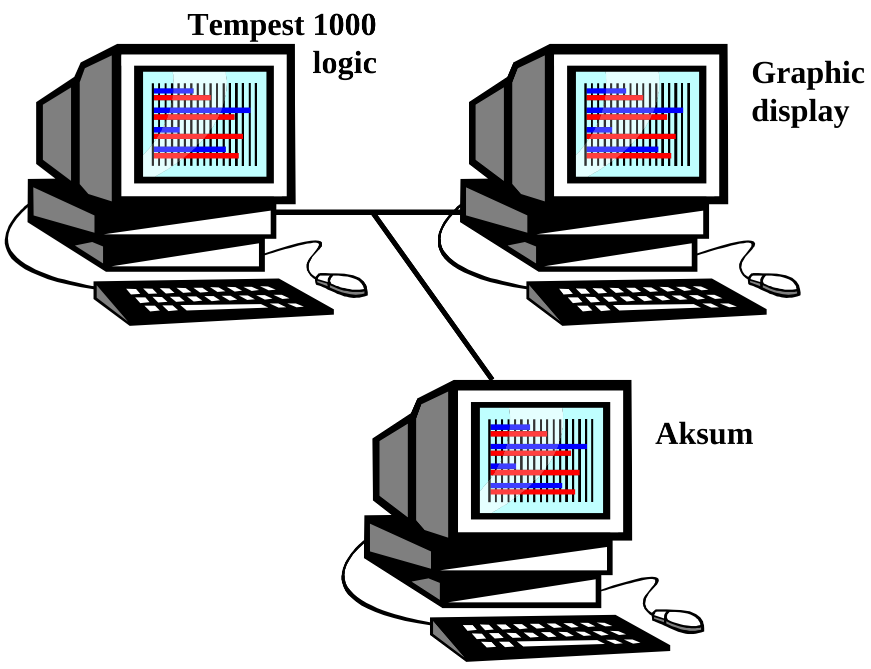


Figure 47. Aksum analyzing Tempest 1000

For the first experiment, no property could be found; for the second, however, Aksum found several regions with a reasonable synchronization overhead, all of them related to event processing (like the methods *handleEvent* and *dispatchEvent*) and screen painting (like the methods *paint* and *drawStuff*), as Figure 49 shows. Since every access to the I/O device must be synchronized, and considering that, when using two nodes, the place where inputs and outputs are generated is not the same place where they are processed, we conjectured that the reason for the synchronization overhead is that the larger amount of time needed to process events remotely makes it more likely to have two concomitant access to the I/O device (for example, the mouse pointer moves while the game's screen is updated).

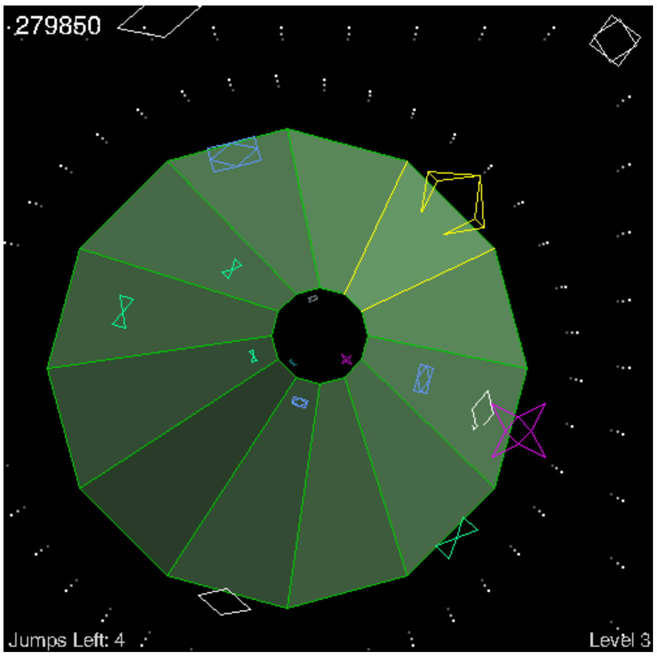
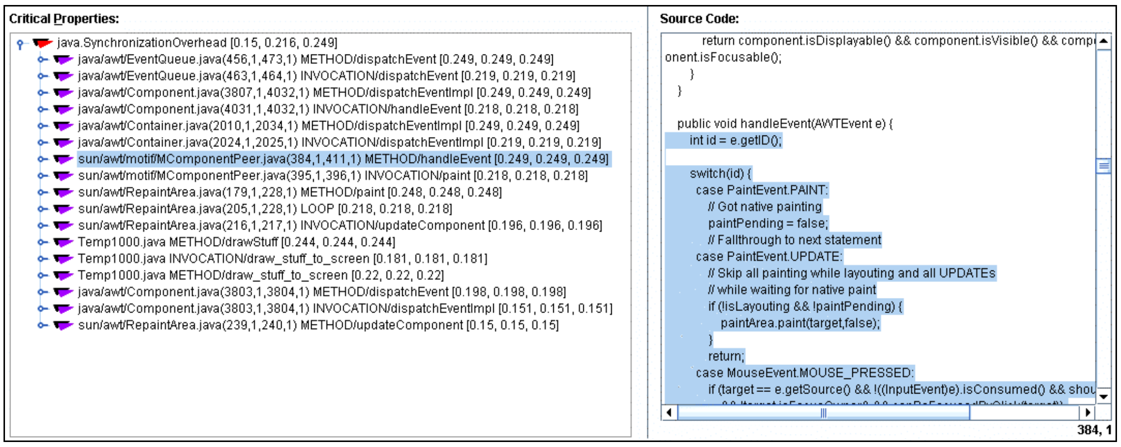


Figure 48. Snapshot of Tempest 1000 with several moving objects


Critical Properties:

```
java.SynchronizationOverhead [0.15, 0.216, 0.249]
    java/awt/EventQueue.java(456,1,473,1) METHOD/dispatchEvent [0.249, 0.249, 0.249]
    java/awt/EventQueue.java(463,1,464,1) INVOCATION/dispatchEvent [0.219, 0.219, 0.219]
    java/awt/Component.java(3807,1,4032,1) METHOD/dispatchEventImpl [0.249, 0.249, 0.249]
    java/awt/Component.java(4031,1,4032,1) INVOCATION/handleEvent [0.218, 0.218, 0.218]
    java/awt/Container.java(2010,1,2034,1) METHOD/dispatchEventImpl [0.249, 0.249, 0.249]
    java/awt/Container.java(2024,1,2025,1) INVOCATION/dispatchEventImpl [0.219, 0.219, 0.219]
    sun/awt/motif/MComponentPeer.java(384,1,411,1) METHOD/handleEvent [0.249, 0.249, 0.249]
    sun/awt/motif/MComponentPeer.java(395,1,396,1) INVOCATION/paint [0.218, 0.218, 0.218]
    sun/awt/RepaintArea.java(179,1,228,1) METHOD/paint [0.248, 0.248, 0.248]
    sun/awt/RepaintArea.java(205,1,228,1) LOOP [0.218, 0.218, 0.218]
    sun/awt/RepaintArea.java(216,1,217,1) INVOCATION/updateComponent [0.196, 0.196, 0.196]
    Temp1000.java METHOD/drawStuff [0.244, 0.244, 0.244]
    Temp1000.java INVOCATION/draw_stuff_to_screen [0.181, 0.181, 0.181]
    Temp1000.java METHOD/draw_stuff_to_screen [0.22, 0.22, 0.22]
    java/awt/Component.java(3803,1,3804,1) METHOD/dispatchEvent [0.198, 0.198, 0.198]
    java/awt/Component.java(3803,1,3804,1) INVOCATION/dispatchEventImpl [0.151, 0.151, 0.151]
    sun/awt/RepaintArea.java(239,1,240,1) METHOD/updateComponent [0.15, 0.15, 0.15]
```

Source Code:

```
            return component.isDisplayable() && component.isVisible() && comp
onent.isFocusable();
        }
    }

    public void handleEvent(AWTEvent e) {
        int id = e.getID();

        switch(id) {
          case PaintEvent.PAINT:
            // Got native painting
            paintPending = false;
            // Fallthrough to next statement
          case PaintEvent.UPDATE:
            // Skip all painting while layouting and all UPDATEs
            // while waiting for native paint
            if (!isLayouting && !paintPending) {
                paintArea.paint(target,false);
            }
            return;
          case MouseEvent.MOUSE_PRESSED:
            if (target == e.getSource() && !((InputEvent)e).isConsumed() && shou
```

384, 1

Figure 49. Synchronization overhead found in Tempest 1000

9.4.2. One$DB

One$DB [103] is an open source version of Daffodil DB, a commercial Java database. Our experiment with One$DB consisted in inserting several records in a simple set of tables, the structure of which is shown in Figure 50. The experiment used four nodes, one acting as a database server, and the other three as clients issuing concurrently SQL statements to update the database (one hundred INSERT statements for each table in the database for each client). We analyzed only the performance of the clients and, as usual, Aksum was executed on node that was not used in the experiment.

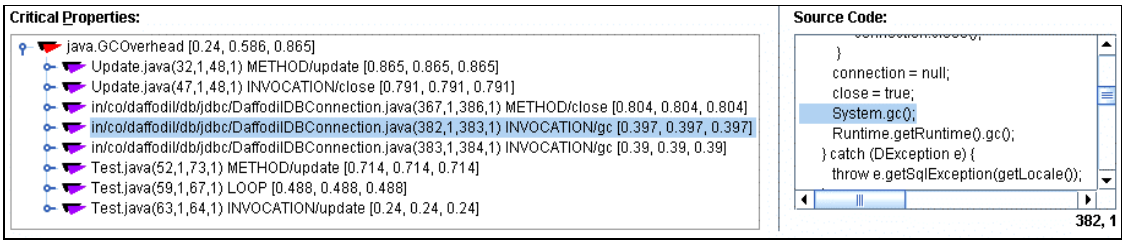
During the analysis, the only problem found was an excessive garbage collection overhead. Figure 51 shows that one of our classes, called *Test*, contains a method called *update*. This method, which extends from line 52 to 73 in the source file, contains a loop, which contains an invocation to method *update* in other of our classes, called *Update*. This invocation concentrates all garbage collection overhead. Now, looking at class *Update*, we see that it has a method called *update*, which invokes the method *close* in class *DaffodilDBConnection*, which belongs to the One$DB API. Again, the garbage collection overhead is concentrated in this single invocation. In the method *close* of class *DaffodilDBConnection*, we can see that the garbage collection overhead is concentrated in the invocations to the method *gc*: *System.gc* (line 382 in the source file) and *Runtime.getRuntime().gc* (line 383 in the source file), one following the other. *System.gc* and *Runtime.getRuntime().gc* are two equivalent ways of explicitly invoking the garbage collector.

Overhead due to garbage collection is inherent to Java, but garbage collection is usually performed transparently and at steps; an explicit invocation causes a very complex and expensive algorithm to run even when this is not needed, and causes the application to pause while the collection is performed. Actually, Sun is very clear about explicit calls to the garbage collection: “Don't call System.gc(). The system will make the determination of when it's appropriate to do garbage collection and generally has the information necessary to do a much better job of initiating a garbage collection” [45].

Since even one single explicit invocation is already objectionable, the use of two consecutive invocations is unjustifiable, being so useful as ordering a list twice. In fact, after having removed both invocations and recompiled the One$DB library, we reduced the execution time of our test application in 83% (from 146 seconds to 25 seconds). Note that garbage collection still occurs, but the decision on *when* it is initiated is left entirely to the Java virtual machine, which, as our results show, can really make a better decision.

```
CREATE TABLE Customer (
    id       INTEGER AUTOINCREMENT PRIMARY KEY,
    name     VARCHAR(20) NOT NULL,
    address  VARCHAR(20) NOT NULL,
    delivery VARCHAR(20) NOT NULL
)

CREATE TABLE Product (
    id    INTEGER AUTOINCREMENT PRIMARY KEY,
    name  VARCHAR(20) NOT NULL
)

CREATE TABLE Supply (
    id        INTEGER AUTOINCREMENT PRIMARY KEY,
    productId INTEGER REFERENCES Product(id)
)

CREATE TABLE ProductOrder (
    id         INTEGER AUTOINCREMENT PRIMARY KEY,
    productId  INTEGER REFERENCES Product(id),
    clientId   INTEGER REFERENCES Customer(id),
    orderDate  DATE NOT NULL
)
```

Figure 50. SQL statements describing the structure of the tables used in the experiment; AUTOINCREMENT is an extension in One$DB, meaning that the default value for the field comes from a sequence number generator kept by One$DB for the table


Critical Properties:

```
java.GCOverhead [0.24, 0.586, 0.865]
    Update.java(32,1,48,1) METHOD/update [0.865, 0.865, 0.865]
    Update.java(47,1,48,1) INVOCATION/close [0.791, 0.791, 0.791]
    in/co/daffodil/db/jdbc/DaffodilDBConnection.java(367,1,386,1) METHOD/close [0.804, 0.804, 0.804]
    in/co/daffodil/db/jdbc/DaffodilDBConnection.java(382,1,383,1) INVOCATION/gc [0.397, 0.397, 0.397]
    in/co/daffodil/db/jdbc/DaffodilDBConnection.java(383,1,384,1) INVOCATION/gc [0.39, 0.39, 0.39]
    Test.java(52,1,73,1) METHOD/update [0.714, 0.714, 0.714]
    Test.java(59,1,67,1) LOOP [0.488, 0.488, 0.488]
    Test.java(63,1,64,1) INVOCATION/update [0.24, 0.24, 0.24]
```

Source Code:

```
    }
    connection = null;
    close = true;
    System.gc();
    Runtime.getRuntime().gc();
  } catch (DException e) {
    throw e.getSqlException(getLocale());
```

382, 1

Figure 51. Garbage collection overhead in One$DB

An excerpt of the analysis performed by the agent when analyzing One$DB (on the nodes names *mary*, *laura* and *grace*) is shown below.

```
Executing <Request stack>
   Requesting stacks
   The stacks changed.
    Reward:  -1,
    next state: SummaryState
        #P=3,#classes=10,Stack size=(6.67±2.49),Classes=(1.10±0.30)

Executing <InstrumentCodeRegions size=5, depth=4>
   Preparing program tree of process 22050@laura
   URIs:  [
     in/co/daffodil/db/jdbc/DBDataSource.class,
     com/daffodilwoods/rmi/server/RmiServerServerSide_Stub.class,
     com/daffodilwoods/rmi/RmiServer.class,
     in/co/daffodil/db/rmi/RmiDaffodilDBDriver.class,
     MyDataSource.class, Test.class, Update.class]
   Preparing program tree of process 1799@grace
   ...
   Preparing program tree of process 5789@mary
   ...
   22050@laura: Instrumenting
   5789@mary: Instrumenting
   instrumented: [
       probe p1: procedure or method getConnection in
         in.co.daffodil.db.rmi.RmiDaffodilDBDriver
         uri=in/co/daffodil/db/rmi/RmiDaffodilDBDriver.java
       probe p2: procedure or method update in
         Test
         uri=Test.java
   ...
   1.094% instrumented
   ...
Reward: -10, next state: RetirementState

Executing <RemoveProbes if inactivity ≥ 30 or probeEffect ≥ 0.55>
Reward: 0, next state: Repentance

Executing <Repent>
   Evaluating probe utility
   probe p4: procedure or method update in Update/uri=Update.java
     invoked 27 times/exec.time=12538
     properties={GCOverhead:4 instances}, 1 new property
   ...
   Probe utility: 5/4-1=0
Reward: -1, next state: StabilityState,stability=2s
...
Executing <Request stack>
   Requesting stacks
   Steps without change in the stacks: 7
   next state: SummaryState
       #P=3,#classes=8,Stack size=(6.00±2.83),Classes=(1.13±0.33)
...
Executing <RemoveProbes if inactivity ≥ 18 or probeEffect ≥ 0.55>
   Removing probe p3: procedure or method main in
     Test/uri=Test.java/never invoked/exec.time=0/properties={}:
     inactive for long time
...
```

10

Conclusion

In this dissertation, we addressed many open problems in the field of performance analysis; we proposed novel solutions for these problems, and showed how our proposals were successfully applied to several practical problems and how they eventually led to improvement in the performance of some applications. In this section, we outline the main contributions of our work and possible directions for future research.

10.1. Contributions

We tackled the problem of integration between instrumentation, monitoring and performance analysis tools by proposing a rich interface for the communication between tools (MIR) and a structural representation for applications written in Java, Fortran, C or C++ (SIR) that capture most of the needs and requirements of performance analysis as it is done today. Since SIR and MIR are language and platform neutral, they reduce the dependency not only on a specific tool but also on particular programming environments or paradigms. Besides having shown several use cases, we defined the rules that map program constructs to SIR elements, and completely defined the grammar of SIR and MIR documents.

We showed data layouts optimized for use in performance analysis and how they can be accessed through Java interfaces and used to specify performance problems in an application. These interfaces, in conjunction with some utility classes, constitute JavaPSL. We demonstrated how JavaPSL can be used to specify simple and complex performance properties, ranging from overheads to inefficiency or load imbalance for a set of heterogeneous nodes. Performance properties provide a normalized value, called severity, which allows them to be compared to each other and, consequently, directs the performance analyst's attention to the most severe problems.

We developed Twilight, a sophisticated instrumentation and monitoring tool that uses the most recent advances in the Java platform to instrument and collect performance data. Twilight was entirely written in Java, which means that, if you can run your Java application on a specific platform, then you can also attach Twilight to monitor the application. Twilight can parse and instrument both source and compiled Java codes, being able to insert probes in and remove probes from an application in execution. In particular, the bytecode parser we wrote for Twilight detects much of the original structure in the source code: packages, classes, methods, method invocations, synchronized blocks, and loops. Finally, Twilight can also profile the Java API and provide interesting metrics, like number of strings created or number of bytes sent due to remote method invocations. Important as well is the fact that Twilight has full support for SIR and MIR.

We created Aksum, a highly flexible and customizable performance analysis tool that automatically conducts a set of experiments and detect the performance bottlenecks in these experiments. The cause-consequence relationship between overheads is used to perform a systematic performance analysis based on overhead classification. Aksum is, to a high degree, independent of hardware and programming paradigm: we showed the performance analysis of programs written in Fortran and Java running on different platforms and with different operating systems.

By using the concept of severity, Aksum can interpret the performance data collected, such that the output naturally guides the user to the most critical performance properties detected. The customization possibilities that Aksum offers virtually allow the creation of one's personal performance analysis tools: input parameters and files that make up an application can be defined and combined; performance properties can be freely added, configured or removed; the end of the search process can be controlled; the output can be grouped, sorted, filtered, and displayed in a multitude of ways. Internally, Aksum can be easily extended to support other instrumentation tools besides those that are currently supported.

We defined numerically concepts like probe effect, inactivity of a probe, and similarity between two stack traces, and then we proposed a way of formally modeling performance analysis as a reinforcement learning problem by showing how the ideas of performance analysis can be mapped to the states, actions, and rewards of a reinforcement learning problem. We integrated our reinforcement learning agent into Aksum, and showed how it can be used to efficiently search for performance problems in an application. Through reinforcement learning, Aksum can adjust automatically several parameters necessary in dynamic analysis, like time between requests for stack traces, size of methods to instrument, and when instrumentation must be removed. Furthermore, the design and implementation of performance analysis techniques based on reinforcement learning is simplified, since there are already many algorithms for solving the generic reinforcement learning problem.

10.2. Future Work

Some issues that have been partially or not addressed at all in our work are possible research directions for the future:

- Trace files
  Trace files constitute an import source of information about the behaviour of an application; nevertheless, they were practically ignored during our work: no performance property that uses trace information was defined, and traces were only mentioned en passant in the chapters about Twilight and SIR/MIR. Trace files also introduce more overhead than profiles; therefore it will be needed to validate the strength of Aksum in the presence of trace files.
- Security
  We never addressed in practice the general problem of security during the development of Aksum and Twilight. However, this is a matter of paramount importance in any tool targeted at distributed systems.
- Reuse of knowledge base
  Our reinforcement learning agent never uses knowledge acquired in previous executions, because we felt that applications behave so differently from each other that it does not make sense to reuse, in the analysis of one application, the knowledge base generated during the analysis of other application. We need to check how true this conjecture is and perhaps insert more variables in our model. Moreover, we can study the influence of other learning techniques, like planning, on our algorithm for reinforcement learning.

A

XML

An *XML document* is a well-formed data object according to the XML specification [36]. An XML document contains at least one *element*, and each element may have a set of *attributes* and may be nested within other elements. The (unique) element in the XML document that is not nested is called the *root element*. For instance, in

```
<staff>
  <employee matr="B001" name="John Doe" marriedTo="A003"/>
  <employee matr="B002" name="John Smith"/>
  <employee matr="A003" name="Jane Doe" marriedTo="B001"/>
</staff>
```

<staff> and *<employee>* are elements, while *matr*, *name* and *marriedTo* are attributes of the element *<employee>*.

A *DTD* (Document Type Definition) is a set of markup declarations that defines the grammar for a class of XML documents. An XML document that has an associated DTD and complies with it is said to be *valid*. For example, the previous example is valid according to the following DTD:

```
<!ELEMENT staff (employee)+> <!-- meaning: a staff element may contain -->
                             <!-- one or more employee elements -->
<!ELEMENT employee EMPTY> <!-- meaning: an employee element may not -->
                          <!-- contain any text or nested element -->
<!ATTLIST employee       <!-- meaning: an employee element: -->
  matr ID #REQUIRED      <!-- must have the attribute matr, a string -->
                         <!-- not used as the value of any other -->
                         <!-- ID-attribute in the same XML document -->
  name CDATA #REQUIRED   <!-- must have the attribute name, a string -->
  marriedTo IDREF #IMPLIED>   <!-- may have the attribute marriedTo, -->
                              <!-- a string with the same value of -->
                              <!-- any other ID-attribute in the -->
                              <!-- same XML document -->
```

DTDs have a limited type capability and a different syntax from XML documents; this motivated the development of *XML schemas*. An *XML schema* is itself an XML document that describes the structure and constrains the contents of XML documents by following the *XML schema language specification* [160]. The corresponding XML schema for the DTD above could be

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- corresponds to ATTLIST employee … in the DTD -->
  <xs:element name="employee">
    <xs:complexType>
      <xs:attribute name="matr" type="xs:ID" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="marriedTo" type="xs:IDREF"/>
    </xs:complexType>
  </xs:element>
  <!-- corresponds to ELEMENT staff (employee)+ in the DTD -->
  <xs:element name="staff">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="employee"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

General rules for translating DTDs to XML schemas can be found in [20]; in addition, the type restrictions shown in Table 7 and Table 8 should be used when converting the DTDs presented in Section 4 to XML schemas.

Element	*Attribute*	*Type in the XML Schema*
location	startLine	nonNegativeInteger
location	startColumn	nonNegativeInteger
location	endLine	nonNegativeInteger
location	endColumn	nonNegativeInteger
location	uri	anyURI
scheduling	chunk	positiveInteger
variable	dimensions	nonNegativeInteger
dimension	index	positiveInteger
dimension	lowerBound	integer
dimension	upperBound	integer

Table 7. Types to be used when converting to XML Schema the elements and attributes of the DTD describing the SIR grammar

Element	*Attribute*	*Type in the XML Schema*
thread	omp-master	boolean
resource	in	anyURI
resource	out	anyURI
snapshot	site	boolean
snapshot	node	boolean
snapshot	process	boolean
snapshot	thread	boolean
snapshot	freeze	boolean
measuring	delivery	nonNegativeInteger
measuring	destination	anyURI
measuring	interval	nonNegativeInteger
measuring	duration	nonNegativeInteger

measurement	value	decimal
instrreq	activate	boolean
instrreq	defaults	boolean
instrreq, ctrlreq	flush	boolean

Table 8. Types to be used when converting to XML Schema the elements and attributes of the DTD describing the MIR grammar

B

DTDs for SIR and MIR

SIR.dtd

```
<!ELEMENT sir (variable*, (group | unit)+)>
<!ATTLIST sir
  language (fortran | java | c | cpp) #REQUIRED
  messagePassing (true|false) #IMPLIED
  sharedMemory (true|false) #IMPLIED
>

<!ELEMENT group (inheritance*, location?, variable*, (group|unit)*)>
<!ATTLIST group
  id ID #REQUIRED
  type (module | package | class | interface) #REQUIRED
  name CDATA #IMPLIED
  internal CDATA #IMPLIED
>

<!ELEMENT inheritance EMPTY>
<!ATTLIST inheritance
  id IDREF #IMPLIED
  name CDATA #IMPLIED
>

<!-- either id or name must be specified -->
<!ELEMENT unit (alias?, location?, variable*, variableRef*,
                (group | unit | codeRegion)*)>
<!ATTLIST unit
  id ID #REQUIRED
  type (function | subroutine | program | method) #REQUIRED
  name CDATA #IMPLIED
  arguments IDREFS #IMPLIED
  virtual (true|false) #IMPLIED
  internal CDATA #IMPLIED
  language (fortran | java | c | cpp) #IMPLIED
  instrumentable (true|false) #IMPLIED
>

<!ELEMENT alias (#PCDATA)>

<!ELEMENT codeRegion (callee?, location?, variable*, variableRef*,
                      scheduling?, (expression | loopControl)*,
                      (codeRegion | group)*)>
<!ATTLIST codeRegion
  id ID #REQUIRED
  type CDATA #REQUIRED
  criticalSectionName CDATA #IMPLIED
  noWait (true|false) #IMPLIED
>
<!-- The recommended code region type include
(block|assign|loop|if|switch|where|jump|call|io|try|catch|finally|
 vector|forAll|
 parallelRegion|parallelLoop|parallelSections|parallelSingle|
 parallelWorkshare|parallelMaster|parallelCriticalSection|
 parallelAtomic|parallelBarrier|parallelFlush|parallelOrdered)
-->

<!ELEMENT callee EMPTY>
<!ATTLIST callee
  id IDREF #IMPLIED
  name CDATA #IMPLIED
>

<!ELEMENT expression ((codeRegion | group)+)>
<!ELEMENT loopControl (lower?, upper?, stride?)>
<!ELEMENT lower (codeRegion+)>
<!ELEMENT upper (codeRegion+)>
<!ELEMENT stride (codeRegion+)>
<!ELEMENT scheduling EMPTY>
<!ATTLIST scheduling
  type (static | dynamic | guided | runtime) #REQUIRED
  chunk CDATA #IMPLIED
>

<!ELEMENT location EMPTY>
<!ATTLIST location
  startLine CDATA #IMPLIED
  startColumn CDATA #IMPLIED
  endLine CDATA #IMPLIED
  endColumn CDATA #IMPLIED
  uri CDATA #IMPLIED
>

<!ELEMENT variable (location?, dimension*)>
<!ATTLIST variable
  id ID #REQUIRED
  name CDATA #IMPLIED
  type CDATA #IMPLIED
  dimensions CDATA #IMPLIED
>
<!ELEMENT dimension EMPTY>
<!ATTLIST dimension
  index CDATA #REQUIRED
  upperBound CDATA #REQUIRED
  lowerBound CDATA #REQUIRED
>
<!ELEMENT variableRef EMPTY>
<!ATTLIST variableRef
  targetId IDREF #REQUIRED
  accessType (read|write|readwrite) #IMPLIED
>
```

MIR.dtd

```
<!-- COMMON -->
<!ELEMENT site (node*|(process*, thread*))>
<!ATTLIST site
  id CDATA #REQUIRED
  name CDATA #IMPLIED>

<!ELEMENT node (process*, thread*)>
<!ATTLIST node
  id CDATA #REQUIRED
  name CDATA #IMPLIED>

<!ELEMENT process (thread*|stack*)>
<!ATTLIST process
  id CDATA #REQUIRED
  name CDATA #IMPLIED>

<!ELEMENT thread (stack*)>
<!ATTLIST thread
  id CDATA #REQUIRED
  name CDATA #IMPLIED
  ompMaster (true|false) #IMPLIED>

<!ELEMENT stack (#PCDATA)>

<!-- SIR Request -->

<!ELEMENT sirreq (resource+)>

<!ELEMENT resource EMPTY>
<!ATTLIST resource
  in CDATA #REQUIRED
  out CDATA #IMPLIED>

<!-- Snapshot Request -->

<!ELEMENT snapshotreq EMPTY>
<!ATTLIST snapshotreq
    site (true|false) #IMPLIED
    node (true|false) #IMPLIED
    process (true|false) #IMPLIED
    thread (true|false) #IMPLIED
    named (true|false) #IMPLIED
    stack CDATA #IMPLIED
    freeze (true|false) #IMPLIED>

<!-- Snapshot -->

<!ELEMENT snapshot (site*, node*, process*, thread*)>

<!-- Instrumentation Request -->

<!ELEMENT instrreq (
    codeRegion*,
    metric*,
    event*,
    measuring?,
    site*, node*, process*, thread*)>
<!ATTLIST instrreq
    defaults (true|false) #IMPLIED
    activated (true|false) #IMPLIED
    flush (true|false) #IMPLIED>

<!ELEMENT codeRegion EMPTY>
<!ATTLIST codeRegion
    from CDATA #REQUIRED
    to CDATA #IMPLIED>

<!ELEMENT metric EMPTY>
<!ATTLIST metric
    name CDATA #REQUIRED>
```

```
<!ELEMENT event EMPTY>

<!ELEMENT measuring (aggregate*)>
<!ATTLIST measuring
    delivery    CDATA #IMPLIED
    destination CDATA #IMPLIED
    interval    CDATA #IMPLIED
    duration    CDATA #IMPLIED>

<!ELEMENT aggregate EMPTY>
<!ATTLIST aggregate
    function (AVERAGE|MAXIMUM|MINIMUM|SUM|VARIANCE) #IMPLIED
    group CDATA #IMPLIED>
<!-- group contains SITE NODE PROCESS THREAD METRIC -->

<!-- Instrumentation Probe -->

<!ELEMENT probes (probe+)>

<!ELEMENT probe EMPTY>
<!ATTLIST probe
    id CDATA #REQUIRED>

<!-- Value of a Measurement -->

<!ELEMENT measurement (measurement)*>
<!ATTLIST measurement
  probe          CDATA #IMPLIED
  siteId         CDATA #IMPLIED
  nodeId         CDATA #IMPLIED
  processId      CDATA #IMPLIED
  threadId       CDATA #IMPLIED
  value          CDATA #IMPLIED>

<!-- Control Request -->

<!ELEMENT ctrlreq (
  probe+,
  metric*,
  measuring?,
  site*, node*, process*, thread*)>
<!ATTLIST ctrlreq
  flush (true|false) #IMPLIED
  action (VALUE|ACTIVATE|DEACTIVATE|RESET|REMOVE) #REQUIRED>

<!-- Error -->

<!ELEMENT errors (error)+>
<!ELEMENT error (#PCDATA)>

<!-- Standard answer -->
<!ELEMENT ok EMPTY>
```

C

SIR Example

Figure 52 shows a Java program with a loop, a synchronized statement, and method invocations. These are currently the constructs that Twilight can detect when generating the SIR from a compiled Java program.

```
1  import java.util.List;
2
3  class Example {
4      private float addMoreTaxes(float f) {
5          return f * 1.05f;
6      }
7
8      final float sum(List<Float> vlist, TaxCalculator txc) {
9          float sum = 0;
10         synchronized (vlist) {
11             for(int i = vlist.size(); i >= 0; i--) {
12                 float f = vlist.get(i);
13                 sum += addMoreTaxes(txc.addTaxes(f));
14             }
15         }
16         return sum;
17     }
18 }
```

Figure 52. Source code

```
1   <?xml version="1.0"?>
2   <sir language="java">
3     <group type="class" name="Example" id="_1">
4       <inheritance id="_4"/>
5       <location uri="Example.java"/>
6       <unit type="method" name="addMoreTaxes" id="_20"
              arguments="par0" virtual="false">
7         <variable id="par0" type="float"/>
8       </unit>
9       <unit type="method" name="Example" id="_21">
10         <codeRegion type="call" id="_22">
11           <callee id="_5"/>
12           <location startLine="3"/>
13         </codeRegion>
14      </unit>
```

Figure 53. Beginning of class *Example*, methods *addMoreTaxes* and default constructor

Figure 53 shows the first lines of the SIR generated by Twilight, with line 3 starting the definition of class *Example*. This class extends (line 4) the class with id _4 (class *java.lang.Object*, shown in Figure 56, line 54) and it was compiled from the file *Example.java* (line 5). Lines 6 to 8 contains the definition of method *addMoreTaxes*; because this method has only one multiplication and one assignment, the SIR does show contain any code region. Note that, because the method is declared *private*, its definition in the SIR has the attribute *virtual* set to *false* (see Section 4.1.3). Lines 9 to 14 show the definition of the default constructor of class Example. A Java compiler must generate a default constructor if the source does not define any. The default constructor just invokes the constructor of the superclass, whose id is _5. This constructor is shown in Figure 56, line 55.

```
15  <unit type="method" name="sum" id="_23"
           arguments="par1 par2">
16     <variable id="par1" type="java.util.List"/>
17     <variable id="par2" type="TaxCalculator"/>
18     <codeRegion type="parallelCriticalSection" id="_24">
19       <location startLine="10"/>
20       <codeRegion type="call" id="_25">
21         <callee id="_10"/>
22         <location startLine="11"/>
23       </codeRegion>
24       <codeRegion type="loop" id="_26">
25         <location startLine="11"/>
26         <codeRegion type="call" id="_27">
27           <callee id="_11"/>
28           <location startLine="12"/>
29         </codeRegion>
30         <codeRegion type="call" id="_28">
31           <callee id="_17"/>
32           <location startLine="12"/>
33         </codeRegion>
34         <codeRegion type="call" id="_29">
35           <callee id="_19"/>
36           <location startLine="13"/>
37         </codeRegion>
38         <codeRegion type="call" id="_30">
39           <callee id="_20"/>
40           <location startLine="13"/>
41         </codeRegion>
42       </codeRegion>
43     </codeRegion>
44   </unit>
45 </group>
```

Figure 54. Method *add*

Figure 54 shows the definition of method *sum* (lines 15 to 44). Final methods are virtual in Java, so the attribute *virtual* does not appear (thedefault value for this attribute is *true*). The synchronized block that encloses most of the method's code starts at line 18 and ends at line 43. Lines 20 to 23 show the invocation to method with id _10. This is the method *size* of class *java.util.List*, shown in Figure 56, line 72. Note that, although this invocation belongs to the loop in the source code, the algorithm we use for loop detection places the invocation outside the loop (because the method is invoked only once). Lines 24 to 42 shows the definition of the loop corresponding to the lines 11 to 14 in the source code. Lines 26 to 29 show the invocation to method with id _11. This is the method *get* in class *java.util.List* (see Figure 56, line 73). The value returned by the method *get* belongs to the class *java.lang.Float*, and it must be converted to a value of primitive type *float* before it can be assigned to the variable *f* (see line 12 in the source code). The code for the conversion is automatically generated by the compiler, in a process called automatic unboxing: an invocation to the method *floatValue* of class *java.lang.Float*. The invocation is shown from line 30 to line 33, and the definition of method *floatValue* is shown at line 61 of Figure 56. Lines 34 to 37 contain the invocation to method *addTaxes* of class *TaxCalculator* (shown in Figure 55). Finally, lines 38 to 41 contain the invocation to method *addMoreTaxes*.

```
46    <group type="class" name="TaxCalculator" id="_18">
47      <inheritance id="_4"/>
48      <unit type="method" name="addTaxes" id="_19"
              arguments="par4" instrumentable="false">
49        <variable id="par4" type="float"/>
50      </unit>
51    </group>
```

Figure 55. Class *TaxCalculator*

```
52    <group type="package" name="java" id="_2">
53      <group type="package" name="lang" id="_3">
54        <group type="class" name="Object" id="_4">
55          <unit type="method" name="Object" id="_5"
                  instrumentable="false"/>
56        </group>
57        <group type="interface" name="Iterable" id="_9"/>
58        <group type="class" name="Float" id="_12">
59          <inheritance id="_13"/>
60          <inheritance id="_16"/>
61          <unit type="method" name="floatValue" id="_17"
                  instrumentable="false"/>
62        </group>
63        <group type="class" name="Number" id="_13">
64          <inheritance id="_4"/>
65          <inheritance id="_15"/>
66        </group>
67        <group type="interface" name="Comparable" id="_16"/>
68      </group>
69      <group type="package" name="util" id="_6">
70        <group type="interface" name="List" id="_7">
71          <inheritance id="_8"/>
72          <unit type="method" name="size" id="_10"
                  instrumentable="false"/>
73          <unit type="method" name="get" id="_11"
                  arguments="par3" instrumentable="false">
74            <variable id="par3" type="int"/>
75          </unit>
76        </group>
77        <group type="interface" name="Collection" id="_8">
78          <inheritance id="_9"/>
79        </group>
80      </group>
81      <group type="package" name="io" id="_14">
82        <group type="interface" name="Serializable" id="_15"/>
83      </group>
84    </group>
85  </sir>
```

Figure 56. Classes of the Java API directly or indirectly used in the class *Example*

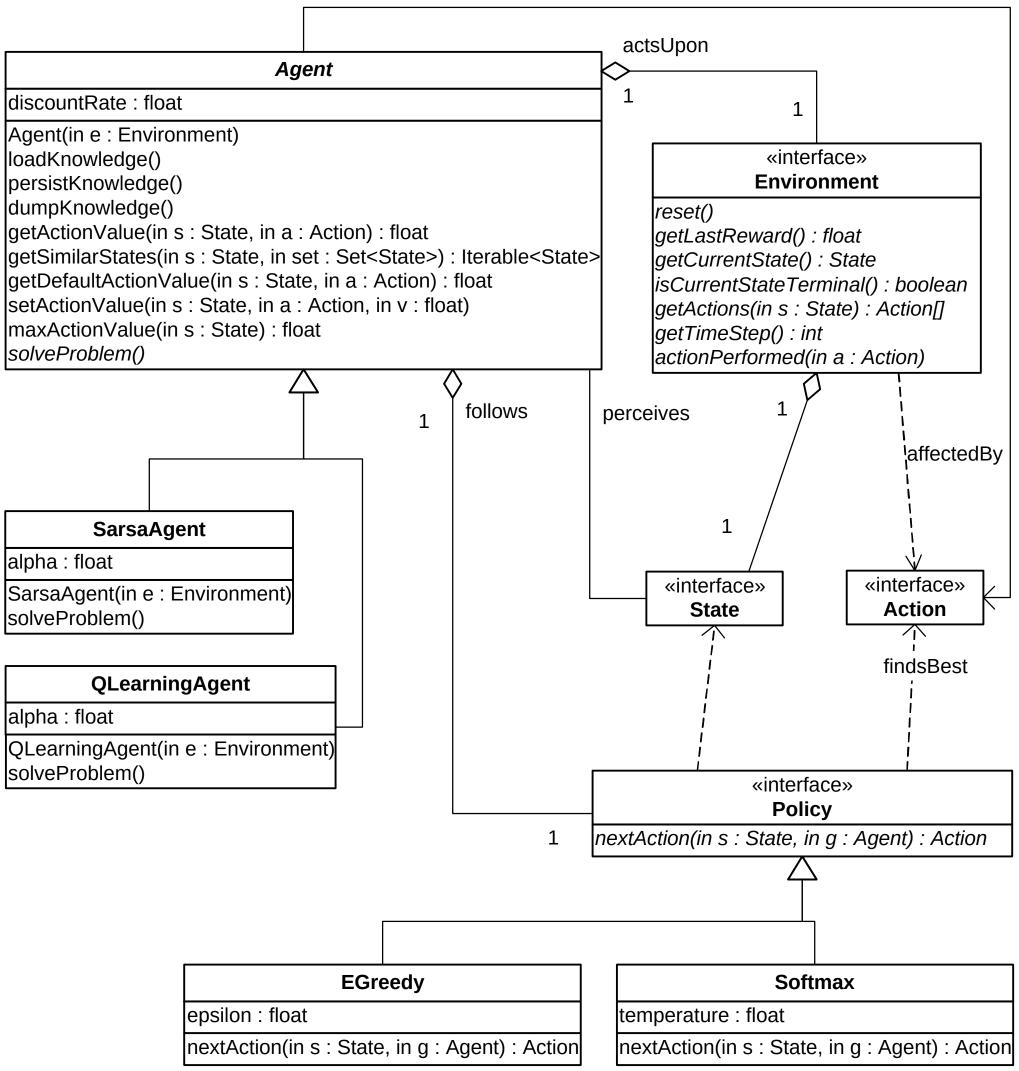
Figure 55 contains the definition of class *TaxCalculator* and one of its methods, *addTaxes*. The class was inserted in the SIR just to resolve the reference to the method *addTaxes* in the method *add*, shown in the previous figure. For this reason, other methods that the class might have, as well as the code of method *addTaxes*, are not inserted. Note also that the method *addTaxes* is marked as non-instrumentable.

Finally, Figure 56 shows the classes and methods of the Java API that were used in the class *Example*. Some of these definitions–class *java.lang.Number* (line 63) and -interfaces *java.lang.Iterable* (line 57), *java.util.Comparable* (line 67), *java.util.Collection* (line 77), and *java.io.Serializable* (line 81)–were not used directly in class Example, but for the classes in the Java API themselves. For example, the class *java.lang.Float* extends the class *java.lang.Number*, which implements the interface *java.io.Serializable*.

D

A Framework for Solving Reinforcement Learning Problems

We developed a generic framework in Java for solving reinforcement learning problems. Figure 57 shows the classes that compose this framework with their respective methods.



Agent
discountRate : float
Agent(in e : Environment)
loadKnowledge()
persistKnowledge()
dumpKnowledge()
getActionValue(in s : State, in a : Action) : float
getSimilarStates(in s : State, in set : Set<State>) : Iterable<State>
getDefaultActionValue(in s : State, in a : Action) : float
setActionValue(in s : State, in a : Action, in v : float)
maxActionValue(in s : State) : float
solveProblem()
actsUpon
1
1
«interface»
Environment
reset()
getLastReward() : float
getCurrentState() : State
isCurrentStateTerminal() : boolean
getActions(in s : State) : Action[]
getTimeStep() : int
actionPerformed(in a : Action)
follows
1
perceives
1
affectedBy
1
SarsaAgent
alpha : float
SarsaAgent(in e : Environment)
solveProblem()
QLearningAgent
alpha : float
QLearningAgent(in e : Environment)
solveProblem()
«interface»
State
«interface»
Action
findsBest
«interface»
Policy
nextAction(in s : State, in g : Agent) : Action
1
EGreedy
epsilon : float
nextAction(in s : State, in g : Agent) : Action
Softmax
temperature : float
nextAction(in s : State, in g : Agent) : Action

Figure 57. Framework for solving reinforcement problems

The interface *Environment* characterizes the reinforcement learning problem. It has the following methods:

- *void reset( )*
  Puts the environment in an initial state.

- *float getLastReward( )*
  Returns the reward obtained from the last execution of an action on the environment.
- *State getCurrentState( )*
  Returns the current state of the environment.
- *boolean isCurrentStateTerminal( )*
  Verifies if the current state of the environment is a terminal state.
- *Action[ ] getActions(State s)*
  Returns all possible actions for the given state.
- *int getTimeStep( )*
  Returns the number of actions executed in the environment.
- *void actionPerformed(Action a)*
  Informs the environment that the agent decided for the given action.

Agent is an abstract class that acts upon an Environment, which is given as argument to the Agent's constructor. One method in the Agent is abstract and must be defined in the subclasses: *solveProblem( ),* which solves the reinforcement learning problem using a particular algorithm. Besides *solveProblem( )* and the *get* and *set* methods used to obtain and define the policy and the discount rate, the other methods of an Agent are:

- *Agent(Environment e) throws java.io.IOException*
  Creates an agent, initializing the environment where the agent operates and loading a knowledge base, if there is one. Throws an IOException if there is knowledge base but it cannot be loaded.
- *void loadKnowledge( ) throws java.io.IOException*
  Reads the file containing the knowledge base and loads the knowledge found there. The value of the property *learning.agent.kb* is used as the name of the file; if the property is undefined, the default name for the knowledge base is used: *learning.kb*. Throws an *IOException* if an I/O error occurs when loading the knowledge base.
- *void persistKnowledge( ) throws java.io.IOException*
  Writes the knowledge of the agent to a file. The value of the property *learning.agent.kb* is used as the name of the file; if the property is undefined, the default name is used: *learning.kb*. Throws an *IOException* if an I/O error occurs when writing the knowledge base.
- *void dumpKnowledge( )*
  Dumps the knowledge of the agent to the standard output.
- *float getActionValue(State s, Action a)*
  Returns the action-value of s and a for policy π, which is the expected return starting from s, taking the action a, and then following policy π, where π is the policy used in the agent. There are two special cases to consider: 1) the state s has already been seen before, but the action a has never been performed when at state s; and 2) the state s has never been seen before, but the agent has some knowledge about similar states. For the first case, the value returned will be the same value returned by the method *getDefaultActionValue(s, a)*. For the second case, the agent examines sequentially the similar states $\{s_1, s_2, \ldots, sn\}$ returned by *getSimilarStates(s, T),* where T is the set of all states the agent has some knowledge about, and returns the first expected reward found. If no reward is

found (because there is no state similar to *s*, or because the agent has never performed the action *a* for any of the similar states), the value returned is also be the same value returned by the method *getDefaultActionValue(s, a)*.

- *Iterable<State> getSimilarStates(State s, Set<State> set)*
  Returns states from a set that are similar to a given state *s*. The default implementation of this method simply returns an empty *Iterable*. Subclasses should override this method if possible.
- *float getDefaultActionValue(State s, Action a)*
  Returns the initial value for an action if executed when the environment is in the given state. The default implementation of this method returns 0.
- *float maxActionValue(State s)*
  Returns the value for the action with the best value among all of the possible actions for the given state *s*.
- *void setActionValue(State s, Action a, float v)*
  Defines as *v* the value of action *a* if executed when the environment is at the given state *s*.

The classes *SarsaAgent* and *QLearningAgent* extend the basic Agent class by implementing the method *solveProblem* using respectively the algorithms Sarsa [120, 143] and Q-learning [155]. Both algorithms depend on *alpha*, a constant step-size parameter.

The interface *Policy* has only one method, *nextAction*, which determines the best action the Agent *g* should execute for state *s*. There are two concrete implementations for this interface: *EGreedy* and *Softmax*. Given the possible actions a_1, a_2, …, a_n for state *s*:

- EGreedy chooses most of the time the action with the best action value, that is, EGreedy selects an action a_k such that $g.getActionValue(s,a_k) \geq g.getActionValue(s,a_i)$, $\forall$i, $1 \leq i \leq n$, but with probability ε chooses randomly some other possible action.
- *Softmax* chooses an action *a* for state *s* with probability $\frac{e^{g.getActionValue(s,a)/t}}{\sum_{i=1}^{n} e^{g.getActionValue(s,a_i)/t}}$,

  where τ is a positive parameter called temperature.

E

The Format of Class Files

In this appendix, we describe the binary representation accepted by the Java virtual machine. This representation is hardware and operating system independent, and is called *class file* because it is normally (but not necessarily) stored in a file with the extension *.class*. It is fundamental to understand this representation in order to manipulate it, as Twilight (described in Chapter 6) does.

E.1. Definitions

The Java virtual machine specification [75, 84] defines areas whose function is common to all virtual machines, although their format depends on specific virtual machine implementations. These areas are listed below.

- *heap*: area from which memory for objects is allocated. The heap is shared among all threads, being created when the virtual machine is initialized.
- *method area*: area where data about loaded classes and interfaces are stored. Like the heap, it is shared among all threads.
- *runtime constant pool*: stores symbols from the constant pool of a class (see section E.3); there is one runtime constant pool for each class or interface loaded.
- *virtual machine stack*: area where frames (defined below) are stored. Each thread has a stack, created when the thread is started.
- *native stacks*: area for the stacks of native methods
- *frame*: area created when a method is invoked, and destroyed when the invocation finishes, whether normally or because an exception is thrown. The memory for the frame is allocated from the virtual machine stack of the thread that invoked the method. A frame is composed of a local variable table, an operand stack (both described below) and a reference to the runtime constant pool of the class where the invoked method was declared.
- *local variable table*: stores local variables, including the arguments received in the method invocation.
- *operand stacks*: store partial results computed by a method, as well as parameters for methods to be invoked and the return values of these methods.
- *pc register*: program counter register, which stores the address of the instruction being executed. There is one *pc register* per thread.

The virtual machine has support for several primitive types (like *char*, *boolean*, and *float*), but only the following types are allowed for variables in the local variable table and for values in the operand stack: *integer*, *float*, *long*, *double*, *object reference* (a reference to an object), and *return address* (the address of an instruction, see section E.7). The primitive types *boolean*, *byte*, *char*, *short*, and *int* are always represented as integer values in the operand stack and in the local variable table. Variables of type *long* and *double* occupies two positions in the local variable table, while other variable types occupies only one position. Similarly, values of type *long* and *double* occupies two positions in the operand stack, while other value types occupy only one position.

E.2. Class File Format

A class file is a sequence of bytes that defines the characteristics and the behavior of a class. Single values with 2, 4, and 8 bytes are built by concatenating consecutive bytes, always in big-endian order (higher-order bytes first). The description of the class file format below will use a notation similar to the C language. Moreover, it will use matrix notation to represent the several tables a class file may have, but because the elements in these tables have variable size, they cannot be seen exactly as matrices in the C language. The notation **u2**, **u4**, and **u8** will be used to represent unsigned values of, respectively, 2, 4, and 8 bytes.

```
class_file {
  u4 magic_number;
  u2 minor_version;
  u2 major_version;
  u2 constant_count_plus_1;
  constant constant_table[constant_count_plus_1 – 1];
  u2 access_flags;
  u2 this_class_index;
  u2 super_class_index;
  u2 interface_count;
  u2 interface_table[interfaces_count];
  u2 field_count;
  field field_table[field_count]
  u2 method_count;
  method method_table[method_count]
  u2 attribute_count;
  attribute attribute_table[attribute_count];
}
```

- `magic_number`:
  The first four bytes of class file represent a "magic number," which must always be 0xCAFEBABE.
- `minor_version, major_version`:
  Represent the version of the class file format used. A Java virtual machine may accept or refuse a given version.
- `constant_count_plus_1`:
  Contains the number of constants in the constant pool plus 1.
- `constant_pool`:
  A table of structures indexed from 1 to constant_count_1 – 1, which contains constants and symbols the class file refers to, like methods and fields. The format of this table is described in section E.3.
- `access_flags`:
  A mask of flags containing properties of the class or interface the file refers to. Figure 58 describes the meaning of the bits in this mask when they are on.
- this_class_index:
  Index, in the constant pool, of the constant that represents the class or interface defined by the class file.
- superclass_index:
  If the class file represents the class `java.lang.Object`, `superclass_index` is 0. If the class file represents any other class, `superclass_index` must be the index,

in the constant pool, of the constant representing the superclass of the class defined by the class file; otherwise, it must be the index of the constant representing the class `java.lang.Object` in the constant pool.

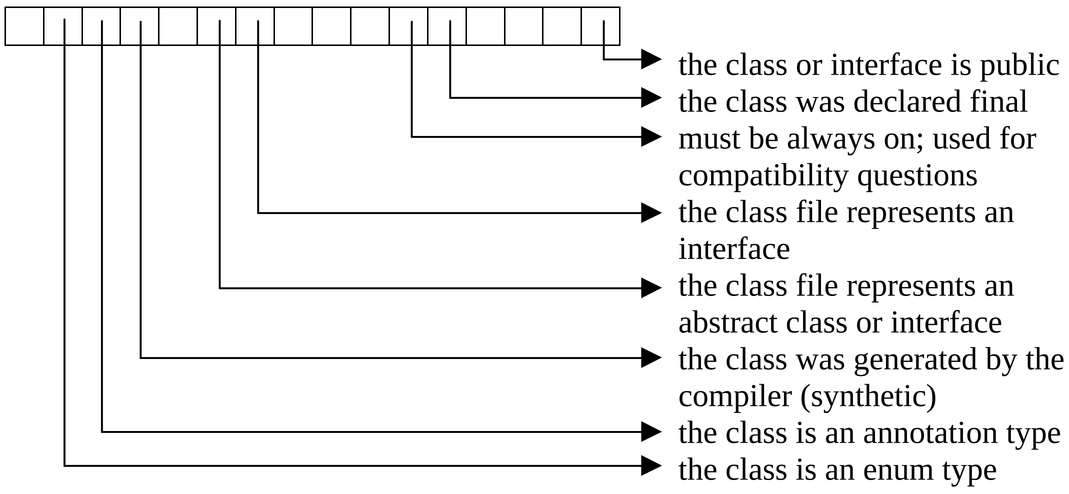

the class or interface is public
the class was declared final
must be always on; used for compatibility questions
the class file represents an interface
the class file represents an abstract class or interface
the class was generated by the compiler (synthetic)
the class is an annotation type
the class is an enum type

Figure 58. Mask of flags for a class

- `interface_count`:
  Contains the number of entries in the interface table.
- `interface_table`:
  Table containing indices, in the constant pool, of entries representing the interfaces implemented or extended by the class or interface represented by the class file. It is indexed from 0 to `interface_count` – 1.
- `field_count`:
  Contains the number of entries in the field table.
- `field_table`:
  Table, indexed from 0 to `field_count` – 1, containing information about the fields declared in the class or interface represented by the class file (inherited fields are not represented). Section 0 describes this table in details.
- `method_count`:
  Contains the number of entries in the method table.
- `method_table`:
  Table, indexed from 0 to `method_count` – 1, containing information about the methods declared in the class or interface represented by the class file. Inherited methods are not represented in this table, described in details in Section 0.
- `attribute_count`:
  Contains the number of entries in the attribute table.
- `attribute_table`:
  Table, indexed from 0 to `attribute_count` – 1, containing extra information about the class or interface the class file represents. Attribute tables are described in Section E.5.

E.3. The Constant Pool

The constant pool is a table where each entry represents some symbol or constant the class file refers to at some moment. For instance, if a code of a method contains an invocation like:

```
System.out.println(3.9);
```

a compiler must generate, in the constant pool of the class file, entries representing the class `System`, the static field `out` (name and type), the method `println` (name, parameter type, and return type) and the constant `3.9`.

Class names are represented in the constant pool using its fully qualified form, that is, containing the name of the package the class belongs to. Furthermore, for historical reasons, slashes, not dots, are used to separate the names of packages and subpackages. Consequently, the name of the class [em]java.lang.Object, for example, is represented as [em]java/lang/Object.

A field type is represented by a string of characters called *field descriptor*, encoded according to the rules shown in Table 9.

String	*Type represented*
B	byte
C	char
D	double
F	float
I	int
J	long
L⟨class name⟩;	reference
S	short
Z	boolean
[	reference to a matrix dimension

Table 9. Coding for types; áclass nameñ is a completely qualified class name

A field of type `double`, for example has its type represented by the letter `D`, while a field of type `double[][]` has its type represented by `[[D`, and a field of type *java.util.List* has its type represented by `Ljava/lang/List;`.

Similarly, a *method descriptor* encodes the type of arguments and return value of a method. It is composed of zero or more characters, between parenthesis, representing the argument types, followed by a string of characters representing the type of the return value. The codes used for arguments and the return value are the same as the codes used for fields, with only one difference: `V` is used to represent the return value of a *void* method. For example, the descriptor of the method

```
void foo(long,java.awt.Component[],int)
```

is `(J[Ljava/awt/Component;I)V`.

The structure of each entry in the constant pool is different, being determined by the first byte. There are currently eleven different structures:

-
  ```
  Constant_Utf8 {
     u1 tag;
     u2 length;
     u1[length] string;
  }
  ```

`tag`, the first byte in the structure, must be 1. The following two bytes contain the size of the following byte array, and the remaining bytes encode a string of characters using the format known as Utf8 [148], but slightly modified (the null byte is represented using 2 bytes, and any character is encoded using at most 3 bytes).

-
  ```
  Constant_Integer {
     u1 tag;
     u4 value;
  }
  ```
  tag must be 3. The following four bytes represent a 32-bit integer constant in big-endian format.

-
  ```
  Constant_Float {
     u1 tag;
     u4 value;
  }
  ```
  tag must be 4. The following four bytes represent a float constant encoded using the IEEE 754 floating point single format [64].

-
  ```
  Constant_Long {
     u1 tag;
     u4 high_order_bytes;
     u4 low_order_bytes;
  }
  ```
  tag must be 5. The remaining eight bytes represent a 64-bit long constant in big-endian format. A constant of type long is considered as occupying two positions in the constant pool, that is, the existence of a constant of type long at position *n* implies the existence of position *n*+1, even though there may be nowhere in the class file a reference to this extra entry. This is regarded as a design mistake of the Java virtual machine.

-
  ```
  Constant_Double {
     u1 tag;
     u4 high_order_bytes;
     u4 low_order_bytes;
  }
  ```
  tag must be 6. The remaining eight bytes represent a double constant encoded using the IEEE 754 floating-point double format [64]. Similar to long constants, double constants also “fill” two entries in the constant pool.

The remaining structures contain only indices to other entries in the constant pool:

-
  ```
  Constant_Class {
     u1 tag;
     u2 class_name_index;
  }
  ```
  Represents a class or interface. tag must be 7, and the index refers to a Constant_UTF8 with the fully qualified name of the class or interface represented.

-
  ```
  Constant_String {
     u1 tag;
     u2 value_index;
  }
  ```
  Represents a string constant. tag must be 8, and the index refers to a Constant_UTF8 containing the value of the string.

-
  ```
  Constant_Name_and_type {
     u1 tag;
     u2 name_index;
     u2 descriptor_index;
  }
  ```

Represents a field or method without specifying its class. tag must be 12; `name_index` refers to the entry in the constant pool with a `Constante_Utf8` containing the name of the field or method, and `descriptor_index` refers to the entry in the constant pool with a `Constant_Utf8` containing the field type or method descriptor.

-
  ```
  Constant_Field {
    u1 tag;
    u2 class_or_interface_index;
    u4 name_and_type_index;
  }
  ```
  Represents a field in some class or interface. `tag` must be 9; `class_or_interface_index` refers to the entry in the table with the `Constant_Class` representing the class or interface where the field was declared; `name_and_type_index` refers to the entry with the `Constant_Name_and_type` that represents the name and type of the field.

-
  ```
  Constant_Method {
    u1 tag;
    u2 class_index;
    u4 name_and_type_index;
  }
  ```
  Represents a class method, being similar to a `Constant_Field` (except for the fact that `tag` must be 10).

-
  ```
  Constant_Interface_method {
    u1 tag;
    u2 interface_index;
    u4 name_and_type_index;
  }
  ```
  Represents an interface method, being similar to a `Constant_Field` and to a `Constant_Method` (except for the fact that `tag` must be 11).

E.4. Fields and Methods

A class file contains a table describing the fields declared in the class or interface represented, and another table describing the declared methods. Any of these tables may have size 0. The structures:

```
field {
  u1 access flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attribute_count;
  attribute attribute_table[attribute_count];
}
```
and

```
method {
  u1 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attribute_count;
  attribute attribute_table[attribute_count];
}
```
make up the field table and the method table, respectively. The meaning of each field is similar for both structures.

- `access_flags`:

A mask of flags containing properties of the class or interface the file refers to. The meaning of the bits in this mask (when they are on) is shown in Figure 59 (for fields) and Figure 60 (for methods).

- `name_index`:
Index, in the constant pool, of the `Constant_Utf8` with the name of the field or method.

- `descriptor_index:`
Index, in the constant pool, of the `Constant_Name_and_type` with the field type or method descriptor.

- `attribute_count`:
Contains the number of entries in the attribute table.

- `attribute_table`:
Table, indexed from 0 to `attribute_count` – 1, containing extra information about the method or field represented. Attribute tables are described in Section E.5.

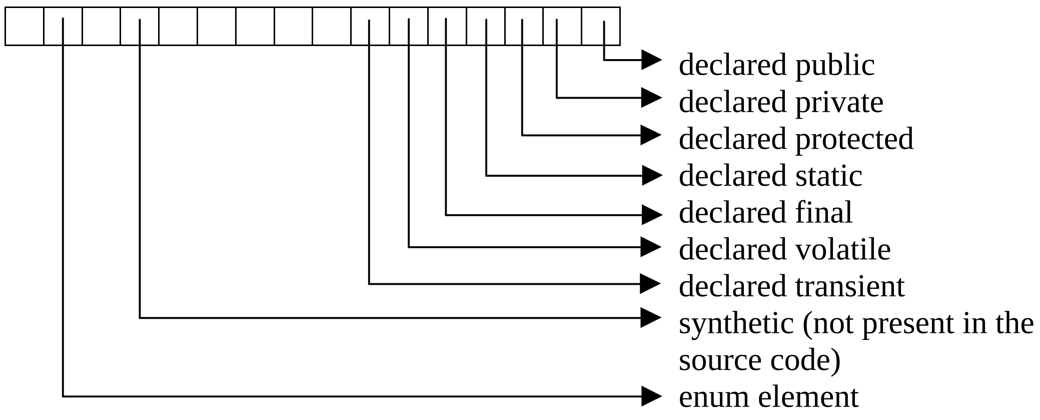


Figure 59. Mask of flags for fields

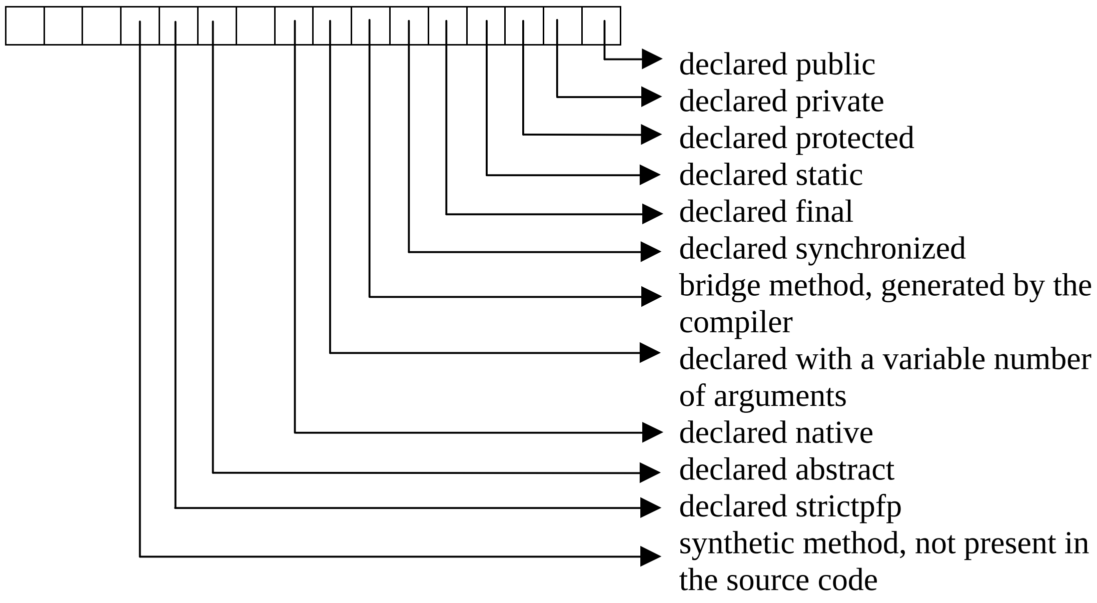


Figure 60. Mask of flags for methods

E.5. Attributes

Attributes provide extra information about a class, field, method or another attribute. The generic structure of each attribute is:

```
attribute {
  u2 name_index;
  u4 attribute_length;
  u1 info[attribute_length];
}
```

where

- `name_index` is an index, in the constant pool, of a Constant_Utf8 containing the name of the attribute;
- `attribute_length` is the length of the attribute, excluding the first 6 bytes; and
- `info`: contains more information about the attribute.

Currently, eleven attributes are defined as part of the class file specification: *Code*, *ConstantValue*, *Deprecated*, *EnclosingMethod*, *Exceptions*, *InnerClasses*, *LineNumberTable*, *LocalVariableTable*, *Signature*, *Synthetic*, and *SourceFile*, of which three must be recognized by any virtual machine (*Code*, *ConstantValue*, and *Exceptions*), three must be recognized in order to implement the Java libraries (InnerClasses, EnclosingMethod, and Synthetic) and one must be recognized by virtual machines that accept class files whose major version is 49 or above (*Signature*). However, nothing prevents a compiler from emitting an attribute that adds functionality to a class, and nothing prevents a particular Java virtual machine from utilizing this attribute. On the other hand, a class or interface must not have its semantics changed when used in other Java virtual machine that does not recognize the attribute, nor can a Java virtual machine reject a class file that does not contain some attribute not defined in the class file specification.

-
  ```
  Code {
    u2 name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 handler_table_length;
    {
      u2 start_pc;
      u2 end_pc;
      u2 handler_pc;
      u2 catch_type;
    } handler_table[handler_table_length];
    u2 attribute_count;
    attribute attribute_table[attribute_count];
  }
  ```
  Contains the code of a method that is neither native nor abstract.
  - `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word “Code.”
  - `attribute_length` is the size of the attribute, excluding the initial six bytes.
  - `max_stack` is the maximum size the operand stack (see Section E.1) may reach during the execution of the code.
  - `max_locals` provides the size of the local variable table (see Section E.1).

- `code_length` is the size of the `code` array.
- `code` contains bytes representing Java virtual machine instructions. The position of an instruction in this array is called instruction's offset but, since many instructions accept operands (and consequently do not occupy exactly one byte), the offset of the *n*-th instruction will seldom be *n*. The execution of a method always starts at offset 0 (the offset of the first instruction).
- `handler_table_length` contains the number of entries in `handler_table`.
- `handler_table` is a table where each entry describes how the Java virtual machine must handle exceptions thrown during the execution of instructions in the code array. If, during the execution of instruction with offset *off*, an exception is thrown, then the `handler_table` of the code is searched for the first entry (`start_pc`, `end_pc`, `handler_pc`, `catch_type`) such that `start_pc` ≤ *off* < `end_pc` and such that `catch_type` is either 0 or the index, in the constant pool, of a `Constant_Class` representing the class of the exception thrown or one of its superclasses. If such entry is found, then the execution branches to `handler_pc`.
- `attribute_count` contains the number of entries in the `attribute_table`.
- `attribute_table` contains attributes providing extra information about the code, like the attributes *LineNumberTable* and *LocalVariableTable* defined below.

-
  ```
  ConstantValue {
    u2 name_index;
    u4 attribute_length;
    u2 constant_value_index;
  }
  ```
  Represents the value of a constant field, that is, a field declared as *static* and *final*.
  - `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word "ConstantValue."
  - `attribute_length` is 2.
  - `constant_value_index` is the entry of a `Constant_Double`, `Constant_Integer`, `Constant_Float`, `Constant_Long`, or `Constant_String` in the constant pool representing the constant value.

-
  ```
  Deprecated {
    u2 name_index;
    u4 attribute_length;
  }
  ```
  Marks a class, method or field as deprecated.
  - `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word "Deprecated."
  - `attribute_length` is 0.

-
  ```
  EnclosingMethod {
    u2 name_index;
    u4 attribute_length;
    u2 class_index;
    u2 method_index;
  }
  ```

Provides nesting information about the class represented by the class file if the class is either local (that is, it has a name and is immediately enclosed by a method) or anonymous. Call this class *C*.

- `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word “EnclosingMethod.”
- `attribute_length` is 4.
- `class_index` is the entry of a `Constant_Class` in the constant pool representing the innermost class that encloses the declaration of *C*.
- `method_index` is the entry of a `Constant_Name_and_type` in the constant pool representing the method where *C* is immediately enclosed. The method must belong to the class referenced by `class_index`. If *C* is not immediately enclosed in any method, then `method_index` is 0.

•
```
Exceptions {
  u2 name_index;
  u4 attribute_length;
  u2 exception_count;
  u2 exception_table[exception_count];
}
```
Describes the exceptions a method may throw.

- `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word “Exceptions.”
- `attribute_length` is 2 + 2 × `exception_count`.
- `exception_count` contains the number of entries in `exception_table`.
- `exception_table` contains indices of `Constant_Class` entries in the constant pool representing the types of classes the method is declared to throw.

•
```
InnerClasses {
  u2 name_index;
  u4 attribute_length;
  u2 class_count;
  {
    u2 inner_class_info_index;
    u2 outer_class_info_index;
    u2 inner_name_index;
    u2 inner_class_access_flags;
  } class_table[class_count];
}
```
Provides information about nested classes or interfaces. This attribute must be present in the attribute table of the nested class (or interface) as well as in the attribute table of the enclosing class (or interface). Moreover, there must be an entry in the `class_table` of this attribute for each nested class represented by a `Constant_Class` entry in the constant pool

- `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word “InnerClasses.”
- `attribute_length` is 2 + 8 × `class_count`.
- `class_count` contains the number of entries in `class_table`.
- `class_table` provides detailed information about enclosing or nested classes. Each entry in the table has the following information:
  – `inner_class_info_index` is the entry of a `Constant_Class` in the constant pool representing some nested class *C*.

– `outer_class_info_index` is the entry of a `Constant_Class` in the constant pool representing the class immediately enclosing the declaration of *C*. If C is a local class, then `outer_class_info_index` is 0
– `inner_name_index` is the entry of a `Constant_Utf8` in the constant pool containing the name with which *C* was declared. If *C* is an anonymous class, then `inner_name_index` is 0.
– `inner_name_access_flags` is a mask of flags describing properties of *C*. Figure 61 describes the meaning of the bits in this mask when they are on.

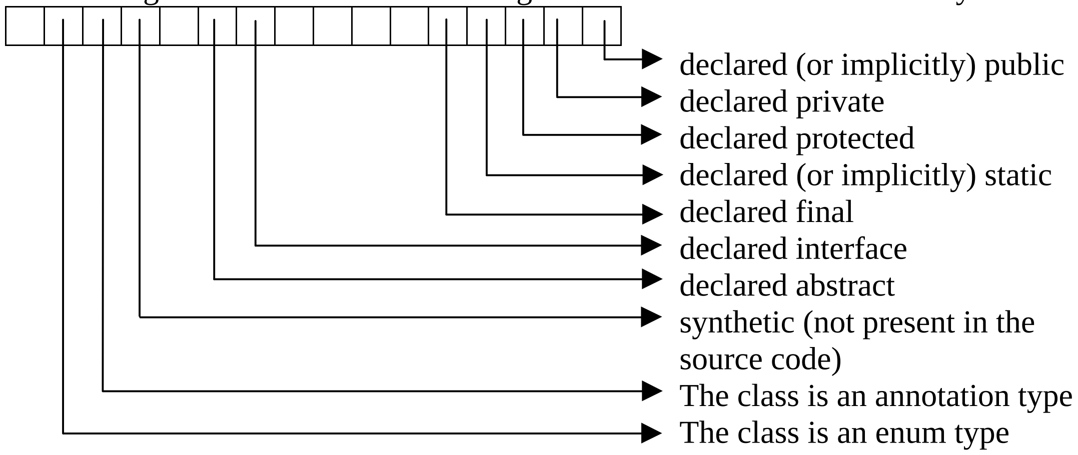

declared (or implicitly) public
declared private
declared protected
declared (or implicitly) static
declared final
declared interface
declared abstract
synthetic (not present in the source code)
The class is an annotation type
The class is an enum type

Figure 61. Mask of flags for nested classes

•
```
LineNumberTable {
  u2 name_index;
  u4 attribute_length;
  u2 line_number_table_length;
  {
    u2 start_pc;
    u2 line_number;
  } line_number_table[line_number_table_length];
}
```
Contains the correspondence between offsets of instructions in the compiled code (see the attribute *Code* above) and the line numbers of the source code from which the class represented by the class file was compiled.

- `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word “LineNumberTable.”
- `attribute_length` is $2 + 4 \times$ `line_number_table_length`.
- `line_number_table_length` contains the number of entries in `line_number_table`.
- `line_number_table` is a table where each entry indicates the offset of the first instruction (`start_pc`) associated to a line number (`line_number`) in the source code.

•
```
LocalVariableTable {
  u2 name_index;
  u4 attribute_length;
  u2 local_variable_table_length;
  {
    u2 start_pc;
    u2 length;
    u2 name_index;
    u2 descriptor_index;
```

```
      u2 index;
    } local_variable_table[local_variable_table_length];
  }
```

Contains information about local variables in the code of a method (see the attribute *Code* above)

- name_index is the entry of a Constant_Utf8 in the constant pool representing the word “LocalVariableTable.”
- attribute_length is $2 + 10 \times$ local_variable_table_length.
- local_variable_table_length contains the number of entries in local_variable_table.
- local_variable_table is a table where each entry provides information about one local variable. The name and type of the local variable are given by the Constant_Utf8 entries in the constant pool at name_index and descriptor_index respectively. The variable is active in the offset range [start_pc, start_pc + length), and its index in the local variable table (see Section E.1) is given by index.

•
```
Signature {
    u2 name_index;
    u4 attribute_length;
    u2 signature_index;
  }
```

Contains the *signature* of a class, method, or field. A signature encodes type information which is not part of the Java virtual machine type system but is used by compilers, debuggers, and the reflection API, like generics and parameterized methods.

- name_index is the entry of a Constant_Utf8 in the constant pool representing the word “Signature.”
- attribute_length is 2.
- signature_index is the entry of a Constant_Utf8 in the constant pool representing a method, field or class signature.

•
```
Synthetic {
    u2 name_index;
    u4 attribute_length;
  }
```

Marks a class, method or field as synthetic, that is, as not present in the source code. Alternatively, the class, method or field may have a bit in its access_flags (see Sections E.2 and E.4).

- name_index is the entry of a Constant_Utf8 in the constant pool representing the word “Synthetic.”
- attribute_length is 0.

•
```
SourceFile {
    u2 name_index;
    u4 attribute_length;
    u2 source_file_index;
  }
```

Provides the name of the source from which a class file was compiled.

- `name_index` is the entry of a `Constant_Utf8` in the constant pool representing the word “SourceFile.”
- `attribute_length` is 2.
- `source_file_index` is the entry of a `Constant_Utf8` in the constant pool representing a name of the source file.

E.6. Valid Class Files

A class file must satisfy several rules in order to be considered valid. These rules are verified by the Java virtual machine:

- when the class or interface is loaded, that is, when its binary representation is found and an object *java.lang.Class* is built, representing the class;
- when the class or interface is linked, that is, when its binary representation, already loaded, is combined with the Java virtual machine so that it can be executed; and
- when a type, method or instruction is referenced for the first time.

The restrictions verified at each moment are distinct and not checked twice. For example, the Java virtual machine verifies only once, when the class is linked, that the name of each field is valid or that no local variable is read before it is initialized; it is also verified only once, the first time method X invokes method Y, if method X does have the permission to invoke method Y.

An error, that is, an object belonging to the class *java.lang.Error*, is thrown if the verification fails.

E.7. Determining Successors and Predecessors

Algorithms for data-flow analysis, among which the one for detecting natural loops mentioned in Section 6.2.2 and described in Section E.8, need to know the possible successors and predecessors of each instruction in the method’s code. Some instructions may have only the following instruction as successor, like the instruction IADD, which pops two integer values off the operand stack, adds the values, and push the result back onto the operand stack. Other instructions have no successor at all, like RETURN, which finishes the method execution. A small set of instructions, called conditional branches, may have two successor instructions: the following instruction and the instruction at the given offset, like IFEQ ⟨offset⟩, which pops an integer value off the operand stack and jumps to the instruction at ⟨offset⟩ only if the popped value is 0. One instruction, GOTO, always cause the execution to branch to the given offset, so the successor of this instructions will be the instruction at the given offset (a GOTO is called an unconditional branch). Finally, the instructions TABLESWITCH and LOOKUPSWITCH, may have several offsets to which the execution may branch; the instructions at these offsets are the possible successors.

One must deal with exception handlers (see the attribute Code in Section E.5) when determining successors. Recall that an exception handler is a quadruple (*start_pc*, *end_pc*, *handler_pc*, *catch_type*). Therefore, each instruction with offset in the range [*start_pc*, *end_pc*) has the instruction at offset *handler_pc* also as a possible successor.

Besides the instructions cited above, there are two instructions that particularly complicate the computation of successors: JSR (jump to subroutine) and RET (return from subroutine). These instructions are used to implement an internal subroutine:

JSR pushes the address of the following instruction (the “return address”) and causes the execution to branch to a given offset in the method’s code. RET reads a return address from the local variable table and causes the execution to branch to the address read (note that everything happens inside the code of a single method). This means that, in the general case, if there is an execution path between an instruction JSR and an instruction RET, the instruction following JSR must be included as a successor of RET. The instructions JSR and RET are traditionally used to implement the `try…finally` construct.

The instructions JSR and RET makes the validation of a class file more difficult and slower. The allowed interactions between these two instructions has never been formally specified by Sun, and the restrictions on correct Java virtual machine code seem to be created “ad-hoc and specific to the particular subroutine labeling algorithm that Sun’s verifier uses” [82]. The instructions JSR and RET will be forbidden as of Java 6 (class files with major version number ≥ 50) [100].

E.8. Detecting Natural Loops and Synchronized Blocks

In order to find natural loops in a class file, one must first determine all pairs of instructions (*m*, *n*) such that *n* is a successor or *m* and such that all execution paths from the first instruction to *m* include the instruction *n*. *n* is said to dominate *m*, and the pair (*m*, *n*) is called a back edge. Next, one must determine all instructions from which *m* can be reached without passing through *n*. The instructions found, *m*, and *n* constitute a natural loop.

The branch caused by exception handlers, as well as the instructions JSR and RET, may generate back edges which, although theoretically belonging to a loop, are not part of any loop written by the programmer; they are an incidental result of the compilation of some other construct, which may have nothing to do with a loop (we detect this, for example, with the compilation of synchronized blocks). Since loops are implemented with conditional and unconditional branches, it makes sense to exclude branches caused by exception handlers, JSR, and RET when determining back edges. Nevertheless, we must considerer conditional and unconditional branches inside exception handlers and inside subroutines generated by pairs JSR/RET.

According to the definition of natural loop, it is not always possible to detect that two loops are nested in the source code. Consider, for instance, the code shown if Figure 62: Loop `a:` is nested in loop `b:`, but because it is impossible, from `b:`, to reach the back edge, the loops are considered disjoint. We had to adopt a pragmatic

```
a:while (true) {
     try {
       doSomething();
     }
     catch (DoingSomethingException e) {
b:     for(User user : loggedUsers) {
         warn(user);
       }
       return;
     }
     done();
     //the compiler generates goto a: here, so creating a back edge
   }
```

Figure 62. Nested loops that cannot be recognized as such using only the algorithm for detection of natural loops

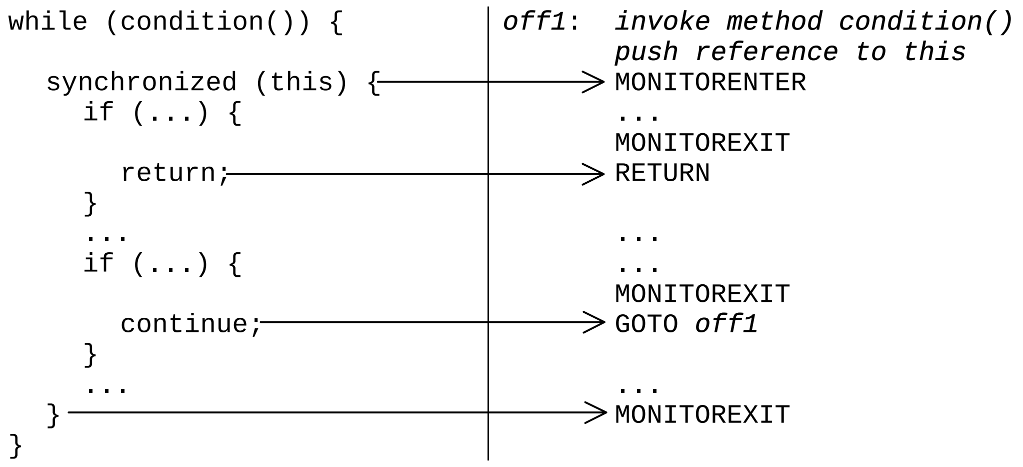
approach and consider that loop X encloses a loop Y if the range of offsets for loop Y contains the range of offsets for loop Y. Although a compiler need not follow this "rule" when generating code, we have not found any compiler which does not.

Detecting synchronized blocks is comparatively easier, not only because of their strict rules in the Java programming language, but also because the Java virtual machine contains only two instructions, MONITORENTER and MONITOREXIT, that deal with entering and leaving monitors. MONITORENTER enters the monitor associated with an object (the one whose reference is at the top of the operand stack), while MONITOREXIT exits the monitor associated with an object.

In the code of a method there may be more than one MONITOREXIT instruction generated for each MONITORENTER (see Figure 63); nevertheless, one, and only one MONITOREXIT instruction is *executed* for each instruction MONITORENTER, that is, there must not be an execution path that allows to leave a method such that a monitor is exited more or less often than the numbered of times it was entered. Note that this rule is enforced by the Java programming language but not by the Java virtual machine.

By determining the successors of each MONITORENTER, one can find the corresponding set of MONITOREXIT instructions. The instruction MONITORENTER, the set of MONITOREXIT instructions found, and the instructions that belong to the successors of MONITORENTER and to the predecessors of the MONITOREXIT instructions found constitute a synchronized block. When determining successors, one can safely ignore branches due to exception handlers: since it is impossible to have a synchronized block that starts outside an exception handler and ends inside of it, there must be at least one normal execution path that reaches a MONITOREXIT instruction from a MONITORENTER; if not, then there is no way of leaving the method normally and all instructions that can be reached from the MONITORENTER are in the synchronized block (which means that there must be an infinite loop inside the synchronized block).

Synchronized blocks may be nested, which means that, when analyzing the execution flow, more than one instruction MONITORENTER may be found before the first MONITOREXIT is reached. We assume that, when traversing the code in



```
while (condition()) {

   synchronized (this) {
      if (...) {

         return;
      }
      ...
      if (...) {

         continue;
      }
      ...
   }
}
```

```
off1:  invoke method condition()
       push reference to this
       MONITORENTER
       ...
       MONITOREXIT
       RETURN

       ...
       ...
       MONITOREXIT
       GOTO off1

       ...
       MONITOREXIT
```

Figure 63. Code generated for a synchronized block

depth-first mode, a MONITOREXIT instruction always refers to the last MONITORENTER seen, which is also how the compilers we tested generate code for nested synchronized blocks.

F

Bugs in the Java API and Sun's Virtual Machine Found During the Development of Aksum and Twilight

The status of these bugs can be monitored at:

`http://bugs.sun.com/bugdatabase/index.jsp`.

Bug ID	**Description**	**Status (Aug 31, 2005)**
4487689	JList.setSelectedValue() throws ArrayIndexOutOfBoundsException on empty list	Fixed
4635001	"." in PATH causes Runtime.exec to execute wrong file	In progress
4655449	Fork in Runtime.exec() hogs processor when interrupted	Fixed when Sun ported the implementation of processes from Solaris to Linux
4662505	llegalArgumentException with empty JTree and key event	Fixed
4681235	JOptionPane.showDialog prints stack trace when thread is interrupted	In progress
4696499	New tree model asked about nodes of previous tree model	Fixed
4704316	API documentation doesn't tell that KeyEvent constructor has been deprecated	Fixed
4772326	copy to clipboard from JList fails	Fixed as a result of fix for bug #4487689
4793099	Keyboard generates concurrent ActionEvents on Solaris/Linux	In progress
4801250	URL.equals inconsistent with RFC1738 and InetAddress	Fixed
4835595	PixelGrabber + setenv DISPLAY slower in Java 1.4	In progress
5003341	class redefined through Instrumentation.redefineClasses can't use native methods	In progress
5053401	SIGSEGV instantiating class redefined through Instrumentation.redefineClasses	Fixed

Bug ID	**Description**	**Status (Aug 31, 2005)**
5053831	IllegalAccessError after Instrumentation. appendToBootstrapClassLoaderSearch	Fixed as result of fix for bug #5055293
5053975	static initializer invoked again after appendToBootstrapClassLoaderSearch	Fixed as result of fix for bug #5055293
5065264	Program needs one minute to finish if JMXConnectorServer.start fails	Fixed
5070671	Arrays.binarySearch can't infer int[]	In progress
5073047	MetalLookAndFeel.setCurrentTheme ignored	In progress
5096167	null class name crashes VM if ClassFileLoadHook is enabled	Fixed
6191049	java.lang.Instrumentation.redefineClass and -Xfuture cause VerifyError	Fixed as result of fix for bug #5092850
6298117	getWaitedTime and getWaitedCount return wrong information	Fixed

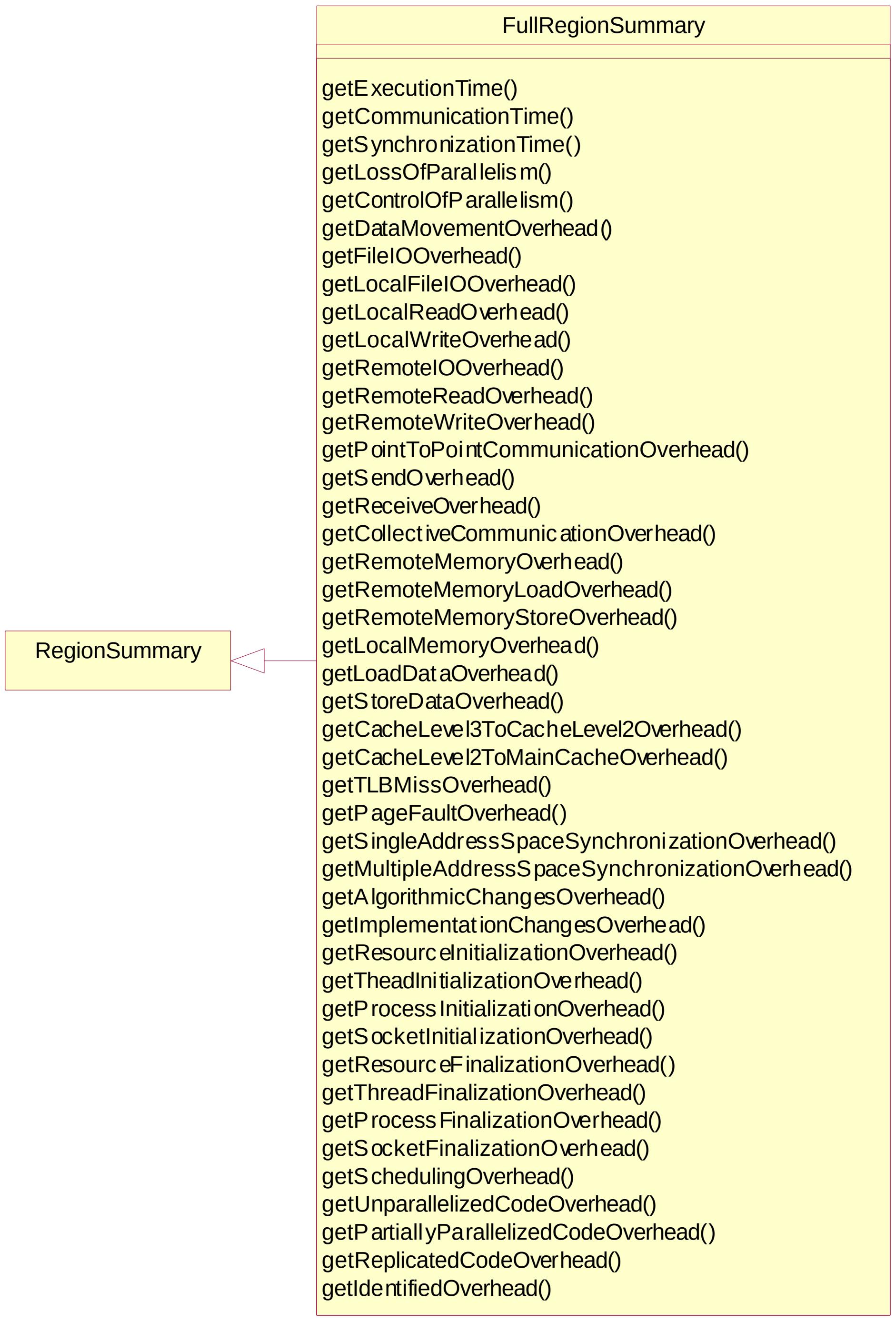
The following bugs have been recently submitted and were not assigned an ID yet.

- Instrumentation.redefineClasses ignores class redefinition (internal review ID: 513666)
- VM spec statement about Sun's compiler is false (internal review ID: 523809)

G

Overhead Properties

Assume that there is a class *FullRegionSummary* that extends *RegionSummary* and provides all overheads defined in Section 2.3.4, as shown in Figure 64.



FullRegionSummary
getExecutionTime()
getCommunicationTime()
getSynchronizationTime()
getLossOfParallelism()
getControlOfParallelism()
getDataMovementOverhead()
getFileIOOverhead()
getLocalFileIOOverhead()
getLocalReadOverhead()
getLocalWriteOverhead()
getRemoteIOOverhead()
getRemoteReadOverhead()
getRemoteWriteOverhead()
getPointToPointCommunicationOverhead()
getSendOverhead()
getReceiveOverhead()
getCollectiveCommunicationOverhead()
getRemoteMemoryOverhead()
getRemoteMemoryLoadOverhead()
getRemoteMemoryStoreOverhead()
getLocalMemoryOverhead()
getLoadDataOverhead()
getStoreDataOverhead()
getCacheLevel3ToCacheLevel2Overhead()
getCacheLevel2ToMainCacheOverhead()
getTLBMissOverhead()
getPageFaultOverhead()
getSingleAddressSpaceSynchronizationOverhead()
getMultipleAddressSpaceSynchronizationOverhead()
getAlgorithmicChangesOverhead()
getImplementationChangesOverhead()
getResourceInitializationOverhead()
getTheadInitializationOverhead()
getProcessInitializationOverhead()
getSocketInitializationOverhead()
getResourceFinalizationOverhead()
getThreadFinalizationOverhead()
getProcessFinalizationOverhead()
getSocketFinalizationOverhead()
getSchedulingOverhead()
getUnparallelizedCodeOverhead()
getPartiallyParallelizedCodeOverhead()
getReplicatedCodeOverhead()
getIdentifiedOverhead()
RegionSummary

Figure 64. A region summary with several methods for retrieving metrics

The following properties can then be defined:

- Communication overhead

The severity of *CommunicationOverhead* is computed as the ratio of the value returned by *getCommunicationTime* and the execution time of a reference code region, for example the main program.

```
public class CommunicationOverhead extends SimpleProperty {
  public CommunicationOverhead(FullRegionSummary rs,
                               CodeRegion basis) {
    severity = rs.getCommunicationTime() / getExecutionTime(basis);
  }
}
```

- Synchronization overhead

The severity of *SynchronizationOverhead* is computed as the ratio of the value returned by *getSynchronizationTime* and the execution time of a reference code region, for example the main program.

```
public class SynchronizationOverhead extends SimpleProperty {
  public CommunicationOverhead(FullRegionSummary rs,
                               CodeRegion basis) {
    severity = rs.getSynchronizationTime()/ getExecutionTime(basis);
  }
}
```

- Loss of parallelism overhead

The severity of *LossOfParallelismOverhead* is computed as the ratio of the value returned by *getLossOfParallelism* and the execution time of a reference code region, for example the main program.

```
public class LossOfParallelismOverhead extends SimpleProperty {
  public LossOfParallelismOverhead( FullRegionSummary rs,
                                    CodeRegion basis) {
    severity = rs.getLossOfParallelism() / getExecutionTime(basis);
  }
}
```

- Control of parallelism overhead

The severity of *ControlOfParallelismOverhead* is computed as the ratio of the value returned by *getControlOfParallelism* and the execution time of a reference code region, for example the main program.

```
public class ControlOfParallelismOverhead extends SimpleProperty {
  public ControlOfParallelismOverhead( FullRegionSummary rs,
                                       CodeRegion basis) {
    severity = rs.getControlOfParallelism()/getExecutionTime(basis);
  }
}
```

- Data movement overhead

The severity of *DataMovementOverhead* is computed as the ratio of the value returned by *getDataMovementOverhead* and the execution time of a reference code region, for example the main program.

```
public class DataMovementOverhead extends SimpleProperty {
  public DataMovementOverhead( FullRegionSummary rs,
                               CodeRegion basis) {
    severity = rs.getDataMovementOverhead()/getExecutionTime(basis);
  }
}
```

- File I/O overhead

The severity of *FileIOOverhead* is computed as the ratio of the value returned by *getFileIOOverhead* and the execution time of a reference code region, for example the main program.

```
public class FileIOOverhead extends SimpleProperty {
  public FileIOOverhead(FullRegionSummary rs, CodeRegion basis) {
    severity = rs.getFileIOOverhead() / getExecutionTime(basis);
  }
}
```

- Local file I/O overhead

The severity of *LocalFileIOOverhead* is computed as the ratio of the value returned by *getLocalFileIOOverhead* and the execution time of a reference code region, for example the main program.

```
public class LocalFileIOOverhead extends SimpleProperty {
  public LocalFileIOOverhead( FullRegionSummary rs,
                              CodeRegion basis) {
    severity = rs.getLocalFileIOOverhead()/getExecutionTime(basis);
  }
}
```

- Local read overhead

The severity of *LocalReadOverhead* is computed as the ratio of the value returned by *getLocalReadOverhead* and the execution time of a reference code region, for example the main program.

```
public class LocalReadOverhead extends SimpleProperty {
  public LocalReadOverhead( FullRegionSummary rs,
                            CodeRegion basis) {
    severity = rs.getLocalReadOverhead() / getExecutionTime(basis);
  }
}
```

- Local write overhead

The severity of *LocalWriteOverhead* is computed as the ratio of the value returned by *getLocalWriteOverhead* and the execution time of a reference code region, for example the main program.

```
public class LocalWriteOverhead extends SimpleProperty {
  public LocalWriteOverhead(FullRegionSummary rs,
                            CodeRegion basis) {
    severity = rs.getLocalWriteOverhead() / getExecutionTime(basis);
  }
}
```

- Remote I/O overhead

The severity of *RemoteIOOverhead* is computed as the ratio of the value returned by *getRemoteIOOverhead* and the execution time of a reference code region, for example the main program.

```
public class RemoteIOOverhead extends SimpleProperty {
   public RemoteIOOverhead( FullRegionSummary rs,
                            CodeRegion basis) {
      severity = rs.getRemoteIOOverhead() / getExecutionTime(basis);
   }
}
```

- Remote read overhead

The severity of *RemoteReadOverhead* is computed as the ratio of the value returned by *getRemoteReadOverhead* and the execution time of a reference code region, for example the main program.

```
public class RemoteReadOverhead extends SimpleProperty {
   public RemoteReadOverhead(FullRegionSummary rs,
                             CodeRegion basis) {
      severity = rs.getRemoteReadOverhead() / getExecutionTime(basis);
   }
}
```

- Remote write overhead

The severity of *RemoteWriteOverhead* is computed as the ratio of the value returned by *getRemoteWriteOverhead* and the execution time of a reference code region, for example the main program.

```
public class RemoteWriteOverhead extends SimpleProperty {
   public RemoteWriteOverhead( FullRegionSummary rs,
                               CodeRegion basis) {
      severity =rs.getRemoteWriteOverhead() / getExecutionTime(basis);
   }
}
```

- Point to point communication overhead

The severity of *PointToPointCommunicationOverhead* is computed as the ratio of the value returned by *getPointToPointCommunicationOverhead* and the execution time of a reference code region, for example the main program.

```
public class PointToPointCommunicationOverhead
      extends SimpleProperty {
   public PointToPointCommunicationOverhead(FullRegionSummary rs,
                                            CodeRegion basis) {
      severity =rs.getPointToPointCommunicationOverhead() /
               getExecutionTime(basis);
   }
}
```

- Send overhead

The severity of *SendOverhead* is computed as the ratio of the value returned by *getSendOverhead* and the execution time of a reference code region, for example the main program.

```
public class SendOverhead extends SimpleProperty {
  public SendOverhead(FullRegionSummary rs, CodeRegion basis){
    severity = rs.getSendOverhead() / getExecutionTime(basis);
  }
}
```

- Receive overhead

The severity of *ReceiveOverhead* is computed as the ratio of the value returned by *getReceiveOverhead* and the execution time of a reference code region, for example the main program.

```
public class ReceiveOverhead extends SimpleProperty {
  public ReceiveOverhead(FullRegionSummary rs, CodeRegion basis){
    severity = rs.getReceiveOverhead() / getExecutionTime(basis);
  }
}
```

- Collective communication overhead

The severity of *CollectiveCommunicationOverhead* is computed as the ratio of the value returned by *getCollectiveCommunicationOverhead* and the execution time of a reference code region, for example the main program.

```
public class CollectiveCommunicationOverhead extends SimpleProperty {
  public CollectiveCommunicationOverhead( FullRegionSummary rs,
                                          CodeRegion basis) {
    severity = rs.getCollectiveCommunicationOverhead () /
               getExecutionTime(basis);
  }
}
```

- Remote memory overhead

The severity of *RemoteMemoryOverhead* is computed as the ratio of the value returned by *getRemoteMemoryOverhead* and the execution time of a reference code region, for example the main program.

```
public class RemoteMemoryOverhead extends SimpleProperty {
  public RemoteMemoryOverhead(FullRegionSummary rs,
                              CodeRegion basis) {
    severity = rs.getRemoteMemoryOverhead()/getExecutionTime(basis);
  }
}
```

- Remote memory load overhead

The severity of *RemoteMemoryLoadOverhead* is computed as the ratio of the value returned by *getRemoteMemoryLoadOverhead* and the execution time of a reference code region, for example the main program.

```
public class RemoteMemoryLoadOverhead extends SimpleProperty {
  public RemoteMemoryLoadOverhead( FullRegionSummary rs,
                                   CodeRegion basis) {
    severity = rs.getRemoteMemoryLoadOverhead() /
               getExecutionTime(basis);
  }
}
```

- Remote memory store overhead

The severity of *RemoteMemoryStoreOverhead* is computed as the ratio of the value returned by *getRemoteMemoryStoreOverhead* and the execution time of a reference code region, for example the main program.

```
public class RemoteMemoryStoreOverhead extends SimpleProperty {
   public RemoteMemoryStoreOverhead( FullRegionSummary rs,
                                     CodeRegion basis) {
      severity = rs.getRemoteMemoryStoreOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Local memory overhead

The severity of *LocalMemoryOverhead* is computed as the ratio of the value returned by *getLocalMemoryOverhead* and the execution time of a reference code region, for example the main program.

```
public class LocalMemoryOverhead extends SimpleProperty {
   public LocalMemoryOverhead( FullRegionSummary rs,
                               CodeRegion basis) {
      severity = rs.getLocalMemoryOverhead()/getExecutionTime(basis);
   }
}
```

- Load data overhead

The severity of *LoadDataOverhead* is computed as the ratio of the value returned by *getLoadDataOverhead* and the execution time of a reference code region, for example the main program.

```
public class LoadDataOverhead extends SimpleProperty {
   public LoadDataOverhead( FullRegionSummary rs,
                            CodeRegion basis) {
      severity = rs.getLoadDataOverhead() / getExecutionTime(basis);
   }
}
```

- Store data overhead

The severity of *StoreDataOverhead* is computed as the ratio of the value returned by *getStoreDataOverhead* and the execution time of a reference code region, for example the main program.

```
public class StoreDataOverhead extends SimpleProperty {
   public StoreDataOverhead( FullRegionSummary rs,
                             CodeRegion basis) {
      severity = rs.getStoreDataOverhead() / getExecutionTime(basis);
   }
}
```

- Cache level 3 to cache level 2 overhead

The severity of *CacheLevel3ToCacheLevel2Overhead* is computed as the ratio of the value returned by *getCacheLevel3ToCacheLevel2Overhead* and the execution time of a reference code region, for example the main program.

```
public class CacheLevel3ToCacheLevel2Overhead extends SimpleProperty{
  public CacheLevel3ToCacheLevel2Overhead( FullRegionSummary rs,
                                           CodeRegion basis) {
    severity = rs.getCacheLevel3ToCacheLevel2Overhead() /
               getExecutionTime(basis);
  }
}
```

- Cache level 2 to main cache overhead

The severity of *CacheLevel2ToMainCacheOverhead* is computed as the ratio of the value returned by *get CacheLevel2ToMainCacheOverhead* and the execution time of a reference code region, for example the main program.

```
public class CacheLevel2ToMainCacheOverhead extends SimpleProperty {
  public CacheLevel2ToMainCacheOverhead(FullRegionSummary rs,
                                        CodeRegion basis) {
    severity = rs.getCacheLevel2ToMainCacheOverhead() /
               getExecutionTime(basis);
  }
}
```

- TLB miss overhead

The severity of *TLBMissOverhead* is computed as the ratio of the value returned by *getTLBMissOverhead* and the execution time of a reference code region, for example the main program.

```
public class TLBMissOverhead extends SimpleProperty {
  public TLBMissOverhead(FullRegionSummary rs, CodeRegion basis){
    severity = rs.getTLBMissOverhead() / getExecutionTime(basis);
  }
}
```

- Page fault overhead

The severity of *PageFaultOverhead* is computed as the ratio of the value returned by *getPageFaultOverhead* and the execution time of a reference code region, for example the main program.

```
public class PageFaultOverhead extends SimpleProperty {
  public PageFaultOverhead( FullRegionSummary rs,
                            CodeRegion basis) {
    severity = rs.getPageFaultOverhead() / getExecutionTime(basis);
  }
}
```

- Single address space synchronization overhead

The severity of *SingleAddressSpaceSynchronizationOverhead* is computed as the ratio of the value returned by *getSingleAddressSpaceSynchronizationOverhead* and the execution time of a reference code region, for example the main program.

```
public class SingleAddressSpaceSynchronizationOverhead
       extends SimpleProperty {
  public SingleAddressSpaceSynchronizationOverhead(
         FullRegionSummary rs, CodeRegion basis) {
    severity = rs.getSingleAddressSpaceSynchronizationOverhead() /
               getExecutionTime(basis);
  }
}
```

- Multiple address space synchronization overhead

The severity of *MultipleAddressSpaceSynchronizationOverhead* is computed as the ratio of the value returned by *getMultipleAddressSpaceSynchronizationOverhead* and the execution time of a reference code region, for example the main program.

```
public class MultipleAddressSpaceSynchronizationOverhead
       extends SimpleProperty {
  public MultipleAddressSpaceSynchronizationOverhead(
         FullRegionSummary rs, CodeRegion basis) {
    severity = rs.getMultipleAddressSpaceSynchronizationOverhead() /
               getExecutionTime(basis);
  }
}
```

- Algorithmic changes overhead

The severity of *AlgorithmicChagesOverhead* is computed as the ratio of the value returned by *getAlgorithmicChagesOverhead* and the execution time of a reference code region, for example the main program.

```
public class AlgorithmicChagesOverhead extends SimpleProperty {
  public Algorithmic ChagesOverhead( FullRegionSummary rs,
                                     CodeRegion basis) {
    severity = rs.getAlgorithmicChagesOverhead() /
               getExecutionTime(basis);
  }
}
```

- Implementation changes overhead

The severity of *ImplementationChagesOverhead* is computed as the ratio of the value returned by *getImplementationChagesOverhead* and the execution time of a reference code region, for example the main program.

```
public class ImplementationChagesOverhead extends SimpleProperty {
  public ImplementationChagesOverhead( FullRegionSummary rs,
                                       CodeRegion basis) {
    severity = rs.getImplementationChagesOverhead() /
               getExecutionTime(basis);
  }
}
```

- Resource initialization overhead

The severity of *ResourceInitializationOverhead* is computed as the ratio of the value returned by *getResourceInitializationOverhead* and the execution time of a reference code region, for example the main program.

```
public class ResourceInitializationOverhead extends SimpleProperty {
  public ResourceInitializationOverhead(FullRegionSummary rs,
                                        CodeRegion basis) {
    severity = rs.getResourceInitializationOverhead() /
               getExecutionTime(basis);
  }
}
```

- Thread initialization overhead

The severity of *ThreadsInitializationOverhead* is computed as the ratio of the value returned by *getThreadInitializationOverhead* and the execution time of a reference code region, for example the main program.

```
public class ThreadInitializationOverhead extends SimpleProperty {
   public ThreadInitializationOverhead( FullRegionSummary rs,
                                         CodeRegion basis) {
      severity = rs.getThreadInitializationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Process initialization overhead

The severity of *ProcessInitializationnOverhead* is computed as the ratio of the value returned by *getProcessInitializationOverhead* and the execution time of a reference code region, for example the main program.

```
public class ProcessInitializationOverhead extends SimpleProperty {
   public ProcessInitializationOverhead( FullRegionSummary rs,
                                          CodeRegion basis) {
      severity = rs.getProcessInitializationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Socket initialization overhead

The severity of *SocketInitializationOverhead* is computed as the ratio of the value returned by *getSocketInitializationOverhead* and the execution time of a reference code region, for example the main program.

```
public class SocketInitializationOverhead extends SimpleProperty {
   public SocketInitializationOverhead( FullRegionSummary rs,
                                         CodeRegion basis) {
      severity = rs.getSocketInitializationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Resource finalization overhead

The severity of *ResourceFinalizationOverhead* is computed as the ratio of the value returned by *getResourceFinalizationOverhead* and the execution time of a reference code region, for example the main program.

```
public class ResourceFinalizationOverhead extends SimpleProperty {
   public ResourceFinalizationOverhead( FullRegionSummary rs,
                                         CodeRegion basis) {
      severity = rs.getResourceFinalizationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Thread finalization overhead

The severity of *ThreadsFinalizationOverhead* is computed as the ratio of the value returned by *getThreadFinalizationOverhead* and the execution time of a reference code region, for example the main program.

```
public class ThreadFinalizationOverhead extends SimpleProperty {
   public ThreadFinalizationOverhead( FullRegionSummary rs,
                                      CodeRegion basis) {
      severity = rs.getThreadFinalizationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Process finalization overhead

The severity of *ProcessFinalizationOverhead* is computed as the ratio of the value returned by *getProcessFinalizationOverhead* and the execution time of a reference code region, for example the main program.

```
public class ProcessFinalizationOverhead extends SimpleProperty {
   public ProcessFinalizationOverhead(FullRegionSummary rs,
                                      CodeRegion basis) {
      severity = rs.getProcessFinalizationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Socket finalization overhead

The severity of *SocketFinalizationOverhead* is computed as the ratio of the value returned by *getSocketFinalizationOverhead* and the execution time of a reference code region, for example the main program.

```
public class SocketFinalizationOverhead extends SimpleProperty {
   public SocketFinalizationOverhead( FullRegionSummary rs,
                                      CodeRegion basis) {
      severity = rs.getSocketFinalizationOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Scheduling overhead

The severity of *SchedulingOverhead* is computed as the ratio of the value returned by *getSchedulingOverhead* and the execution time of a reference code region, for example the main program.

```
public class SchedulingOverhead extends SimpleProperty {
   public SchedulingOverhead(FullRegionSummary rs,
                             CodeRegion basis) {
      severity = rs.getSchedulingOverhead() / getExecutionTime(basis);
   }
}
```

- Unparallelized code overhead

The severity of *UnparallelizedCodeOverhead* is computed as the ratio of the value returned by *getUnparallelizedCodeOverhead* and the execution time of a reference code region, for example the main program.

```
public class UnparallelizedCodeOverhead extends SimpleProperty {
   public UnparallelizedCodeOverhead( FullRegionSummary rs,
                                      CodeRegion basis) {
      severity =  rs.getUnparallelizedCodeOverhead () /
                  getExecutionTime(basis);
   }
}
```

- Partially parallelized code overhead

The severity of *PartiallyParallelizedCodeOverhead* is computed as the ratio of the value returned by *getPartiallyParallelizedCodeOverhead* and the execution time of a reference code region, for example the main program.

```
public class PartiallyParallelizedCodeOverhead
         extends SimpleProperty {
   public PartiallyParallelizedCodeOverhead(FullRegionSummary rs,
                                            CodeRegion basis) {
      severity =  rs.getPartiallyParallelizedCodeOverhead() /
                  getExecutionTime(basis);
   }
}
```

- Replicated code overhead

The severity of *ReplicatedCodeOverhead* is computed as the ratio of the value returned by *getReplicatedCodeOverhead* and the execution time of a reference code region, for example the main program.

```
public class ReplicatedCodeOverhead extends SimpleProperty {
   public ReplicatedCodeOverhead( FullRegionSummary rs,
                                  CodeRegion basis) {
      severity = rs.getReplicatedCodeOverhead() /
                 getExecutionTime(basis);
   }
}
```

- Unidentified overhead

Under the extra assumption that all processors used in a parallel experiment are identical, unidentified overhead can be computed as defined in Section 2.3.3: Using the sequential execution time of a code region, obtained from the summary *seqSummary*, the parallel execution time, obtained from the summary *parSummary*, and the number of processors used in the parallel experiment, one can compute the absolute unidentified overhead and then the severity of the property as the ration between the absolute unidentified overhead and the execution time of a reference code region, for example the main program.

```
public class UnidentifiedOverhead extends SimpleProperty {
   public UnidentifiedOverhead( FullRegionSummary seqSummary,
                                FullRegionSummary parSummary,
                                CodeRegion basis) {
      float sequentialTime = seqSummary.getExecutionTime();
      float parallelTime = parSummary.getCodeRegionTime();
      int n = parSummary.getExperiment().getNumberOfProcessors();
      float unidentifiedOverhead =
         parallelTime – sequentialTime/n –
         parSummary.getIdentifiedOverhead();

      severity = unidentifiedOverhead / getExecutionTime(basis);
   }
}
```

References

1. A. Acharya, G. Edjlali, and J. Saltz. “The Utility of Exploiting Idle Workstations for Parallel Computation.” In *Proceedings of 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*. Seattle, USA. June 1997.
2. E. Allen, D. Chase, V. Luchangco, J-W. Maessen, S. Ryu, G. L. Steele, S. T.-Hochstadt. *The Fortress Language Specification Version 0.618*. http://research.sun.com/projects/plrg/fortress0618.pdf, retrieved on July 14, 2005.
3. Apache Ant. http://ant.apache.org/, retrieved on March 31, 2005.
4. R. A. Aydt. “SDDF: The Pablo Self-Describing Data Format.” Technical Report, Department of Computer Science, University of Illinois. April 1994.
5. G. J. Badros. “JavaML: A Markup Language for Java Source Code.” In *9th International World Wide Web Conference*. Amsterdam, The Netherlands. May 2000.
6. M. K. Bane and G. D. Riley. “Automatic Overheads Profiler for OpenMP Codes.” In *Second European Workshop on OpenMP*. Edinburgh, Scotland, UK. September, 2000.
7. G. Bell and J. Gray. “High Performance Computing: Crays, Clusters, and Centers. What Next?” Technical Report MSR-TR-2001-76, Microsoft Research, 2001.
8. S. Benkner. VFC: The Vienna Fortran Compiler. Scientific Programming, IOS Press, The Netherlands, 7(1):67-81, 1999.
9. Beowulf Project Overview. http://www.beowulf.org/overview, retrieved on June 26, 2005.
10. P. Blaha, K. Schwarz, and J. Luitz. *WIEN97, Full-potential, Linearized Augmented Plane Wave Package for Calculating Crystal Properties*. Institute of Technical Electrochemistry, Vienna University of Technology. Vienna, Austria. 1999.
11. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. “A scalable cross-platform infrastructure for application performance tuning using hardware counters.” In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Dallas, Texas, USA, 2000. IEEE Computer Society.
12. P. A. Buhr and A. S. Harji. “Concurrent Urban Legends.” *Concurrency and Computation: Practice and Experience* 17(9):1133-1172, John Wiley and Sons. June/July 2005.
13. J. M. Bull. “A Hierarchical Classification of Overheads in Parallel Programs.” In *Proceedings of the First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*. London, UK, 2000. Chapman & Hall, Ltd.
14. H. W. Cain, B. P. Miller, and B. J. N. Wylie, “A Callgraph-Based Search Strategy for Automated Performance Diagnosis”, in *Proc. Of the European Conference on Parallel Computing*. Germany, August 2000.

15. D. Callahan, B. L. Chamberlain, H. P. Zima. “The Cascade High Productivity Language.” In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*. IEEE Computer Society, April 2004.
16. D. Carrera, J. Guitart, J. Torres, E. Ayguadé and J. Labarta. “An Instrumentation Tool for Threaded Java Application Servers.” In *XIII Jornadas de Paralelismo*. Lleida, Spain. September 2002.
17. Cleanscape FortranLint Fortran source code analysis tool. http://www.cleanscape.net/products/fortranlint/index.html, retrieved on July 1.
18. M. J. Clement, M .J. Quinn. “Multivariate Statistical Techniques for Parallel Performance Prediction.” In *Proceedings of the 28th Hawaii International Conference on system Sciences*. Hawaii, USA. January 1995.
19. E. D. Collins. Loop-Based Automated Performance Analysis. Presentation at the Paradyn/Condor Week, March 2005.
20. A Conversion Tool from DTD to XML Schema. http://www.w3.org/2000/04/schema_hack, retrieved on March 4, 2005.
21. R. E. Crew. "ASTLOG: A Language for Examining Abstract Syntax Trees." In *Proceedings of the Conference on Domain-Specific Languages*, October 1997.
22. M. E. Crovella And J. J. LeBlanc. “Parallel Performance Prediction Using Lost Cycles Analysis.” In *Proceedings of Supercomputing ’94*. IEEE Computer Society. Washington D.C., USA. 1994.
23. D. E. Culler, J. P. Singh, A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc. 1999.
24. N. Darcy. Tempest 1000. http://www.boomahtrix.btinternet.co.uk/mainsite/Temp1000/tempest.htm, retrieved on September 13, 2005.
25. S. J. Deitz. High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations. PhD Thesis, University of Washington. 2005.
26. P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. http://www.isi.edu/in-notes/rfc1951.txt, retrieved on March 9, 2005.
27. P. T. Devanbu. “Genoa: A Customizable, Language-and-front-end Independent Code Analyzer.” In *Proceedings of the Fourteenth International ACM Conference on Software Engineering*. ACM Press, 1992.
28. Distributed Parallel Programming Environment for Java. http://www.alphaworks.ibm.com/tech/dppej, retrieved on July 28, 2005.
29. E. Dockner and H. Moritsch. *Pricing Constant Maturity Floaters with Embedded Options Using Monte Carlo Simulation*. Aurora Technical Report Aur99_04, University of Vienna. Vienna, Austria. January 1999.
30. J. Dongarra, T. Sterling, H. Simon, E. Strohmaier. “High Performance Computing: Clusters, Constellations, MPPs, and Future Directions.” *Communications of the ACM*, 7(2), 51-59, March/April 2005.
31. J. Donnel. *Java Performance Profiling using the VTune™ Performance Analyzer*. http://www.intel.com/cd/software/products/asmo-na/eng/vtune/219355.htm, retrieved on July 9, 2005.
32. T. Durkin. SETI Researchers Sift Interstellar Static for Signs of Life. Xcell Journal. 2004.
33. Edison Design Group. Compiler Front Ends for the OEM Market. http://www.edg.com, retrieved on July 1, 2005.

34. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. “Dynamically discovering likely program invariants to support program evolution.” In *IEEE TSE*, 27(2):1-25, Feb. 2001.
35. A. Espinosa, T. Margalef, and E. Luque. “Automatic Performance Evaluation of Parallel Programs.” In *IEEE Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing*. IEEE Computer Society Press, January 1998.
36. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, retrieved on March 4, 2005.
37. T. Fahringer. “Estimating and Optimizing Performance for Parallel Programs.” In *IEEE Computer* 28(11):47-56. November 1995.
38. T. Fahringer, M. Gerndt, Tianchao Li, B. Mohr, C. Seragiotto, H.-L. Truong. *Monitoring and Instrumentation Requests for Fortran, Java, C and C++ Programs*. Aurora Technical Report AuR_04-17, University of Vienna. Vienna, Austria. 2004.
39. T. Fahringer, M. Gerndt, Tianchao Li, B. Mohr, C. Seragiotto, H.-L. Truong. *Standardized Intermediate Representation for Fortran, Java, C and C++ Programs*. Aurora Technical Report AuR_04-18, University of Vienna. Vienna, Austria. 2004.
40. T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, J. L. Träff. “Knowledge Specification for Automatic Performance Analysis.” APART Technical Report. Zentralinstitut für Angewandte Mathematik, Interner Bericht. August 2001.
41. T. Fahringer, A. Jugravu, B. Di Martino, S. Venticinque, H. Moritsch. “On the Evaluation of JavaSymphony for Cluster Applications.” In *Proceedings of the IEEE International Conference on Cluster Computing CLUSTER 2002*. Chicago, Illinois. September 2002.
42. M. J. Flynn. “Some Computer Organizations and Their Effectiveness.” *IEEE Transactions on Computing* C-21(September): 948-960.
43. I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Publishing Company. 1995.
44. Free On-Line Dictionary of Computing. http://www.foldoc.org, retrieved on June 5, 2005.
45. Frequently Asked Questions about Garbage Collection in the Hotspot^{TM} Java^{TM} Virtual Machine. http://java.sun.com/docs/hotspot/gc1.4.2/faq.html, retrieved on September 26, 2005.
46. J. E. F. Friedl. *Mastering Regular Expressions, 2nd edition*. O'Reilly & Associates. July 2002.
47. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc. 1995.
48. M. Geissler. *Interaction of High Intensity Ultrashort Laser Pulses with Plasmas*. Vienna University of Technology, 2001.
49. W. Gropp, E. Lusk, N. Doss, A. Skjellum. “A high-performance, portable implementation of the MPI message passing interface standard.” *Parallel Computing*, 22(6), 789-828. September 1996.
50. W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable parallel Programming with the Message-passing Interface, Second Edition*. The MIT Press. 1999.
51. M. Gerndt, A. Krumme. “A Rule-based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory System.” In *Proceedings of 2nd International Workshop on High-Level Programming Models and Supportive Environments*. Geneva, Switzerland, 1997.

52. J. Guitart, J. Torres, E. Ayguadé, J. Oliver, and J. Labarta. "Java Instrumentation Suite: Accurate Analylsis of Java Threaded Applications." In *2nd Workshop on Java for High Performance Computing (part of the 14th ACM International Conference on Supercomputing ICS'00)*. USA, 2000.
53. M. Harkema, D. Quartel, R. van der Mei, and B. Gijsen. "JPMT: A Java performance monitoring tool." In *Proceedings TOOLS-2003*, USA, 2003.
54. P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.
55. D. G. Heap. *Taurus: A Taxonomy of the Actual Utilization of Real UNIX and Windows Servers*. January 2003. http://www.ibm.com/servers/library/pdf/taurus.pdf, retrieved on July 27, 2005.
56. B. R. Helm, A. D. Malony, and Stephen F. Fickas. "Capturing and Automating Performance Diagnosis: The Poirot Approach." In *Proceedings of the 9th International Symposium on Parallel Processing*, 606–613. April 1995.
57. R. Hempel and D. W. Walker. "The Emergence of the MPI Message Passing Standard for Parallel Computing." In *Computer Standards and Interfaces* 21:51-62. 1999.
58. V. Herrarte and W. Lusk. "Studying parallel program behavior with upshot." Technical Report ANL-91/15, Argonne National Laboratory. 1991.
59. High Performance Fortran Forum. High Performance Fortran Language Specification Version 2.0. http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/index.html, retrieved on June 29, 2005.
60. J. K. Hollingsworth, B. P. Miller, J. Cargille. "Dynamic Program Instrumentation for Scalable Performance Tools." In *Scalable High Performance Computing Conference (SHPCC) 1994*, May 1994.
61. J. K. Hollingsworth and B. P. Miller. "An Adaptive Cost Model for Parallel Program Instrumentation." In *EuroPar 1996*. Lyon, France. August 1996.
62. HP 9000 Superdome technical white paper, May 2005. http://h71028.www7.hp.com/ERC/downloads/5982-4000EN.pdf, retrieved on June 24, 2005.
63. IBM developer kit for Linux. http://www-106.ibm.com/developerworks/java/jdk/linux140/, retrieved on May 18, 2005.
64. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Standard. 754-1985.
65. InfiniBand® Trade Association. http://www.infinibandta.org, retrieved on June 27, 2005.
66. Java Technology. http://java.sun.com, retrieved on March 11, 2005.
67. Java Compiler Compiler™ (JavaCC™). https://javacc.dev.java.net, retrieved on March 7, 2005.
68. Java™ Debug Interface. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi, retrieved on July 5, 2005.
69. Java Networking and Proxies. http://java.sun.com/j2se/1.5.0/docs/guide/net/proxies.html, retrieved on September 27, 2005.
70. Java™ Virtual Machine Debug Interface Reference. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html, retrieved on July 9, 2005.

71. Java™ Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html, retrieved on July 9, 2005.
72. Jikes. http://jikes.sourceforge.net/, retrieved on July 11, 2005.
73. JVM™ Tool Interface Version 1.0. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, retrieved on July 9, 2005.
74. B. Joy, G. Stelle, J. Gosling, G. Bracha. *Java™ Language Specification (2nd Edition)*. Addison-Wesley-Professional. 2000.
75. JSR 924: Java™ Virtual Machine Specification. Maintenance review of changes to the Java™ Virtual Machine Specification, Second Edition for J2SE 1.5. http://jcp.org/en/jsr/detail?id=924, retrieved on March 7, 2005.
76. A. Jugravu and T. Fahringer. "JavaSymphony, a Programming Model for the Grid." *Journal for Future Generation Computer Systems–Grid Computing: Theory, Methods and Applications, Promotional Issue*. January 2005.
77. L. P. Kaebling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". *Journal of Artificial Intelligence Research* 4:237-285. 1996.
78. KOJAK: Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks. http://www.fz-juelich.de/zam/kojak/, retrieved on July 4, 2005.
79. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. Benjamin/Cummings. 1994.
80. J. Labarta, S. Girona, V. Pillet, T. Cortés, L. Gregoris. "DiP: A Parallel Program Development Environment." In *2nd International EuroPar*. Lyon, France, August 1996.
81. C. Lazou. "Fortran 2000 Evolves to Meet Challenge of Large Scale Numeric Applications." In *News on HPCN Europe and the European Commission*. http://www.hoise.com/primeur/01/articles/monthly/CL-PR-09-01-1.html, retrieved on July 8, 2005.
82. X. Leroy. "Java bytecode verification: algorithms and formalizations." In *Journal of Automated Reasoning*, 30(3-4):235-269. 2003.
83. J. P. Lewis and U. Neumann. *Performance of Java versus C++*. http://www.idiom.com/~zilla/Computer/javaCbenchmark.html, retrieved on July 14, 2005.
84. T. Lindholm, F. Yellin. *The Java™ Virtual Machine Specification (2nd Edition)*. Addison-Wesley-Professional. 1999.
85. K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen. "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates." In *Proceedings of SC2000*. Maryland, USA. November 2000.
86. T. Ludwig and R. Wismueller. "OMIS 2.0; a Universal Interface for Monitoring Systems." Lecture Notes on Computer Science, 1332:267–276. 1997.
87. J. Marner. *Evaluating Java for Game Development*. http://www.rolemaker.dk/articles/evaljava, retrieved on July 14, 2005.
88. J. M. Malard. "A Role for Pareto Optimality in Mining Performance Data." *Computation: Practice and Experience* 17(1):27-48, John Wiley and Sons. June/July 2005.
89. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn Parallel

Performance Measurement Tool.” In *IEEE Computer* 28, 11, 37-46. November 1995.

90. T. M. Mitchell. *Machine Learning*. WCB/McGraw-Hill. 1997.
91. B. Mohr, D. Brown, and A. Malony. “TAU: A Portable Parallel Program Analysis Environment for pC.” In *Proceedings of CONPAR 94 - VAPP VI*. University of Linz, Austria, LNCS 854, 29-40. September 1994.
92. A. Morajko, O. Morajko, J. Jorba, T. Margalef, and E. Luque. “Automatic Performance Analysis and Dynamic Tuning of Distributed Applications.” In *Parallel Processing Letters* 13(2):169-187. World Scientific, June 2003.
93. mpC Workshop: Integrated Parallel Programming System for Heterogeneous Networks of Personal Computers. http://www.atssoft.com/downloads/mpc_workshop_white.pdf, retrieved on July 13, 2005.
94. MPICH: A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich, retrieved on July 7, 2005.
95. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers. 1997.
96. N. Mukherjee, G.D. Riley, and J.R. Gurd. “Finesse: A Prototype Feedback-guided Performance Enhancement System.” In *Euromicro Workshop on Parallel and Distributed Processing PDP’2000*. January 2000.
97. Myrinet. http://www.myri.com/myrinet, retrieved on June 1, 2005.
98. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer* 12(1):69-80. January 1996.
99. NAS Computing Resources - Columbia Supercomputer. http://www.nas.nasa.gov/Resources/Systems/columbia.html, retrieved on June 26, 2005.
100. New Java SE Mustang Feature: Type Checking Verifier. https://jdk.dev.java.net/verifier.html, retrieved on August 31, 2005.
101. T. Newhall and B. P. Miller. “Performance Measurement of Dynamically Compiled Java Executions.” In *Proceedings of the ACM 1999 Conference on Java Grande*. USA, 1999.
102. K. O'Hair. *The JVMPI Transition to JVMTI.* http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition, retrieved on July 9, 2005.
103. One$DB. http://www.daffodildb.com/one-dollar-db.html, retrieved on September 14, 2005.
104. OPARI: OpenMP Pragma And Region Instrumentor. http://www.fz-juelich.de/zam/kojak, retrieved on July 4, 2005.
105. Open64 Compiler Tools. http://open64.sourceforge.net, retrieved on March 31, 2005.
106. Open64 Project at Rice University. http://www.hipersoft.rice.edu/open64/, retrieved on July 1, 2005.
107. OpenMP Application Program Interface Version 2.5. http://www.openmp.org/drupal/mp-documents/spec25.pdf, retrieved on June 3, 2005.
108. C. M. Pancake. “Exploiting Visualization and Direct Manipulation to Make Parallel Tools More Communicative.” In *Applied Parallel Computing,* edited by B. Kagstrom et al., Springer Verlag. Berlin. 1998.
109. PAPI; Performance Application Programming Interface. http://icl.cs.utk.edu/papi, retrieved on March 7, 2005.

110. P. Pazandak and D. Wells. “ProbeMeister: Distributed Runtime Software Instrumentation.” In *First International Workshop on Unanticipated Software Evolution*. Spain, June 2002.
111. Performance Visualization for Parallel Programs. http://www.lcrc.anl.gov/~dvorak/teragrid/experiments, retrieved on June 19, 2005.
112. S. Pllana, T. Fahringer, and F. Breitenecker. “Performance Modeling and Predictions of Parallel and Distributed Programs with PerformanceProphet.” In *The 2005 International Conference on Parallel Processing. Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems*. Oslo, Norway. 2005.
113. The Portland Group™: PGHPF Compiler User's Guide. http://www.pgroup.com/doc/pghpf_ug/hpfug.htm, retrieved on July 26, 2005.
114. Quadrics homepage. http://www.quadrics.com/quadrics/QuadricsHome.nsf/DisplayPages/Homepage, retrieved on June 27, 2005.
115. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, California.
116. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt, retrieved on March 4, 2005.
117. R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. “Autopilot: Adaptive Control of Distributed Applications.” In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*. July 1998.
118. P. C. Roth and B. P. Miller, "Deep Start: A Hybrid Strategy for Automated Performance Problem Searches." In *Euro-Par 2002*. August 2002.
119. J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc. 1999.
120. G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*. Techincal Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University. 1994.
121. S. J. Russel and P. Norvig. *Artificial Inteligence: A Modern Approach*. Prentice Hall, Inc. 1995.
122. A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers.” *IBM Journal of Research and Development*, 3(3):210-229. 1959
123. N. G. Santiago, D. T. Rover, and D. Rodriguez. “A Statistical Approach for the Analysis of the Relation Between Low-level Performance Information, the Code, and the Environment.” In *International Conference on Parallel Processing Workshops*. 2002.
124. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. “Eraser: A Dynamic Data Race Detector for Multithreaded Programs.” In *ACM Transactions on Computer Systems* 15(4):391-411. 1997.
125. J. Schaeffer, J. C. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. “A World Championship Caliber Checkers Program.” In *Artificial Intelligence* 53(2-3):273-289.1992
126. C. Seragiotto and T. Fahringer. “Aksum: A Performance Analysis Tool for Parallel and Distributed Applications.” In *Performance Analysis and Grid Computing*, edited by V. Getov et al. Kluwer Academic Publishers, ISBN 1-4020-7693-2. Boston, USA. October 2003.

127. C. Seragiotto and T. Fahringer. "Analysis of Distributed Java Applications Using Dynamic Instrumentation." In *Proceedings of Cluster 2005*. Massachusetts, USA. September 2005.
128. C. Seragiotto and T. Fahringer. "Automatic Search for Performance Problems in Parallel and Distributed Programs by Using Multi-Experiment Analysis." In *Proceedings of HiPC 2002*. Bangalore, India. December 2002.
129. C. Seragiotto and T. Fahringer. "Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL." In *Proceedings of Supercomputing 2001*. Colorado, USA. 2001.
130. C. Seragiotto, T. Fahringer, M. Geissler, G. Madsen, H. Moritsch. "On Using Aksum for Semi-Automatically Searching of Performance Problems in Parallel and Distributed Programs." In *Proceedings of 11th Euromicro Conference on Parallel Distributed and Network-based Processing (PDP 2003)*. Genoa, Italy. 2003.
131. C. Seragiotto and T. Fahringer. Performance Analysis for Distributed and Parallel Java Programs." In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. Cardiff, UK. 2005.
132. M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database, And Multiprocessor Operating Systems*. McGraw-Hill, Inc. 1994.
133. S. Shende and A. D. Malony. "Integration and application of TAU in parallel Java environments." *Concurrency and Computation: Practice and Experience*. 15:501-519. John Wiley & Sons, Ltd. 2003.
134. Y. Solihin, V. Lam, and J. Torrellas. "Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors." In *Proceedings of Supercomputing'99*. Portland, USA. November 1999.
135. SQL Server Query Execution Plan Analysis. http://www.sql-server-performance.com/query_execution_plan_analysis.asp, retrieved on June 17, 2005.
136. Sun Fire E25K/E20K Systems Overview Manual. http://www.sun.com/products-n-solutions/hardware/docs/pdf/817-4136-11.pdf, retrieved on June 23, 2005.
137. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press. Cambridge, USA. 1998.
138. Standard Performance Evaluation Corporation (SPEC). http://www.spec.org, retrieved on March 3, 2005.
139. A. J. van der Steen and J. J. Dongarra. "Overview of Recent Supercomputers - 2004." http://www.top500.org/ORSC/2004, retrieved on June 27, 2005.
140. T. Sterling. *Beowulf Cluster Computing with Windows (Scientific and Engineering Computation)*. The MIT Press, 2001.
141. V. S. Sunderam. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, 2, 4, 315-339, December, 1990.
142. Supercomputador MareNostrum, Barcelona Supercomputer Center. http://www.bsc.org.es/resources/marenostrum,es.htm, retrieved on June 27, 2005.
143. R. S. Sutton. "Generalization in reinforcement learning: Successful examples using sparse coarse coding." In *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference*. MIT Press. Massachusetts, USA.
144. A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Inc. 1992.
145. A. S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1999.

146. Top500 Supercomputer sites. http://www.top500.org, retrieved on June 1, 2005.
147. H.-L. Truong and T. Fahringer. "SCALEA: A Performance Analysis Tool for Parallel Programs.” *Concurrency and Computation: Practice and Experience* 15(11-12):1001-1025, John Wiley and Sons. September 2003.
148. The Unicode Consortium. *The Unicode Standard, Version 4.0.0, defined by: The Unicode Standard, Version 4.0*. Addison-Wesley, 2003.
149. J. S. Vetter. “Dynamic Statistical Profiling of Communication Activity in Distributed Applications.” In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. California, USA. 2002.
150. J. S. Vetter. “Performance Analysis of Distributed Applications Using Automatic Classification of Communication Inefficiencies.” In Proceedings of the 14th international conference on Supercomputing. Santa Fe, New Mexico, USA. 2000.
151. J. S. Vetter and M. O. McCracken. “Statistical Scalability Analysis of Communication Operations in Distributed Applications.” In *Proceedings of ACM SIGPLAN Symposium, Principles and Practice of Parallel Programming*. 2001.
152. J. S. Vetter and D. A. Reed. “Managing Performance Analysis with Dynamic Statistical Projection Pursuit.” In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. Oregon, USA. 1999.
153. J. S. Vetter and P. H. Worley. Asserting Performance Expectations. In *Proceedings of SC2000*. Maryland, USA. November 2000.
154. Visual Studio .NET Code Analysis On Demand. http://www.fmsinc.com/dotnet/Analyzer, retrieved on June 17, 2005.
155. C. J. C. H. Watkins. *Learning from Delayed Rewards*. Ph.D. thesis, Cambridge University. 1989.
156. WHIRL Intermediate Language Specification. http://prdownloads.sourceforge.net/open64/whirl.pdf, retrieved on July 1, 2005.
157. F. Wolf and B. Mohr. “Automatic Performance Analysis of MPI Applications Based on Event Traces.” In *Proc. Of the European Conference on Parallel Computing*. Germany, August 2000.
158. F. Wolf and B. Mohr. “EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs.” In *Proceedings of 7th International Conference, HPCN Europe 1999*, 503-512. April 1999.
159. M. Woodacre, D. Robb, D. Roe, and Karl Feind. The SGI® AltixTM 3000 Global Shared-Memory Architecture. http://www.sgi.com/pdfs/3474.pdf, retrieved on June 25, 2005.
160. XML Schema Part 1: Structures. http://www.w3.org/TR/xmlschema-1, retrieved on March 4, 2005.
161. K. Yeung, P. H. J. Kelly, and S. Bennett. “Dynamic Instrumentation for Java Using a Virtual JVM.” *Performance analysis and grid computing*. Kluwer Academic Publishers, ISBN 1-4020-7693-2, 2004.

Lebenslauf

Angaben zur Person	
Name	Clóvis Seragiotto Júnior
Staatsangehörigkeit	Brasilien
Geburtsdatum	08.01.1975
Schul- und Berufsbildung	
Jänner 1999 – November 2000	**Magister der Naturwissenschaften -** Institut für Mathematik und Statistik, Universität São Paulo (Brasilien) und gefördert von FAPESP (Stiftung für die Förderung der Forschung des Bundeslandes São Paulo)
Fachgebiet	Informatik, Schwerpunkt „Concurrent Programming
Diplomarbeit	Automatische Erkennung von „Race conditions“ in Java-Programmen
März 1995 – November 1998	**Bakkalaureus der Computerwissenschaften -** Institut für Mathematik und Statistik, Universität São Paulo (Brasilien)
März 1990 – November 1993	Elektronische Datenverarbeitung, Technische Bundeslehranstalt São Paulo (Brasilien)
1982-1989	Grund- und Hauptschule