Jül-4335

Jülich Supercomputing Centre (JSC)

A Generic Binary Instrumenter and Heuristics to Select Relevant Instrumentation Points

Jan Mußler

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

A Generic Binary Instrumenter and Heuristics to Select Relevant Instrumentation Points

Jan Mußler

Berichte des Forschungszentrums Jülich; 4335
ISSN 0944-2952
Jülich Supercomputing Centre (JSC)
Jül-4335

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL)
unter http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
Telefon: 02461 61-6123 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Contents

Abstract ix

Acknowledgments xi

1 Introduction 1

2 Basics 3

2.1 Parallel Computing 3

2.1.1 Message Passing Interface 4

2.1.2 Open Multi Processing 5

2.1.3 HPC Architecture 5

2.2 Performance Analysis 6

2.2.1 Performance Bottlenecks 7

2.2.2 Direct Measurement and Sampling 7

2.2.3 Tracing 8

2.2.4 Profiling 8

2.2.5 Call Graph and Call Tree 8

2.3 Instrumentation 9
2.3.1 Manual Source Code Instrumentation 10
2.3.2 Automated Source Code Transformation 10
2.3.3 Automated Compiler Supported Instrumentation 10
2.3.4 Library Interposition 11
2.3.5 Static Binary Instrumentation 12
2.3.6 Dynamic Binary Instrumentation 12
2.3.7 Effects of Instrumentation 13
2.4 Dyninst 13
2.4.1 Binary Access and Program Abstraction 14
2.4.2 Supported Instrumentation Points 14
2.4.3 Instrumenting Binary Code 15
2.4.4 Code Representation 16
2.5 Scalasca 16
2.5.1 Workflow 17
2.5.2 Instrumentation 18
2.5.3 Filtering 19

3 Related Work **21**

3.1 Tuning and Analysis Utilities 22
3.1.1 The TAU Source Code Instrumenter 22
3.1.2 The TAU Binary Instrumenter 23
3.2 The OPARI Source Code Instrumenter 23

3.3 Generic Source Code Instrumentation 24
3.4 Paradyn 25
3.4.1 Metric Description Language 26
3.4.2 Dyner 26
3.5 Conclusion 27

4 The Generic Binary Instrumenter **29**
4.1 Overview 29
4.2 Selective Instrumentation 30
4.2.1 Implemented Rules 31
4.2.2 Filter Specification 38
4.3 Generic Instrumentation 40
4.3.1 Adapter Specification 41
4.3.2 Example Adapter Specifications 43
4.4 Implementation 45
4.4.1 Program Flow 45
4.4.2 Execution 46
4.4.3 Filter Generation 46
4.4.4 Code Generation 51
4.5 Dynamic Library Workaround 52

5 Evaluation **55**
5.1 JuRoPa 55

5.2 Early Observations 56
5.3 Overhead Measurements 56
5.4 Measurements 58
5.4.1 Restrictions to Measurement Runs 58
5.4.2 Used Filters 61
5.4.3 DROPS 62
5.4.4 Cactus 67
5.4.5 Gadget 73
5.5 Conclusion 76

6 Summary **81**

7 Future Work **83**

A Appendix **85**

Bibliography **87**

List of Figures

2.1 The Performance Analysis Workflow 6
2.2 MPI Library Interposition 12
2.3 Dyninst Instrumentation 16
2.4 EPIK Run-time Library Architecture 19

4.1 Generic Binary Instrumenter Overview 30
4.2 Properties Related to the Call Graph 35
4.3 Selected Rule Related Classes 47

5.1 Total Introduced Overhead by Different Filters 77
5.2 Achieved Overhead Reduction by Different Filters 78

List of Tables

5.1 Overhead observations: compiler vs. Dyninst instrumentation 57
5.2 DROPS: run times with erroneous paths 59
5.3 Number of overlapping functions, before changes 60
5.4 Number of overlapping functions, after changes 61
5.5 DROPS: run times un- and compiler-instrumented 63
5.6 DROPS: functions instrumented by filter 63
5.7 DROPS: run times by filter 64
5.8 DROPS: run times by filters with combined properties 66
5.9 Cactus benchmarks: run times un- and compiler-instrumented 67
5.10 PUGH: functions instrumented by filter 68
5.11 PUGH: run times by filter 69
5.12 Carpet: functions instrumented by filter 70
5.13 Carpet: run times by filter 71
5.14 Carpet: run times by number of processes 72
5.15 Carpet: time spent in MPI calls 73
5.16 Gadget: run times by filter 74

5.17 Gadget: run times compiler instrumentation 75

A.1 DROPS: run time with I/O 86

A.2 DROPS: run time by filter, I/O enabled 86

Abstract

Various tools targeting High Performance Computing applications, e.g., Scalasca, have been developed to aid software engineers in their analysis and subsequently in the improvement of their application's performance.

This thesis focuses on the very first step of application analysis, introducing a generic binary instrumenter, developed to support different measurement systems and fulfill their demands imposed on the instrumentation. The definition of code fragments uses a subset of C to expose Dyninst's code generation and provide compatibility with different tools. In order to improve the selection process, the instrumenter also features rule-based filtering, permitting a targeted analysis and a reduction of instrumentation overhead that otherwise perturbs the measurement.

The rule-based filter utilizes patterns and properties to identify functions to instrument. Patterns focus on function identifiers and properties expose attributes gathered through the binary analysis using Dyninst. Properties include for example: call graph access, number of instructions, and Lines of Code.

In the evaluation the performance results gathered with instrumented applications are presented. Results obtained by different heuristics that try to improve the modified binary's performance by removing less relevant instrumentation points are compared to the performance results that are obtained selecting only points relevant to Scalasca's communication analysis. Two Cactus benchmarks (C++), DROPS (C++), and Gadget (C) are used as example codes to evaluate possible benefits of selective binary instrumentation.

The results show that binary instrumentation is expensive, but can prove especially beneficial to the analysis of optimized C++ applications. The filters used were able to remove up to 90 percent of the introduced overhead by excluding small functions, which contribute little to overall run time but proved to be responsible for most of the overhead.

Acknowledgments

I would like to thank Prof. Dr. Felix Wolf for giving me the opportunity to join his team and allowing me to peek into the world of High Performance Computing. Additionally, I would like to thank him for encouraging me to present my early work at the Paradyn Week in Madison and the RWTH Aachen Undergraduate Funds for funding my travel.
I am grateful for Prof. Dr. Barton Miller's invitation to Madison and would like to thank him and his group for the hospitality during my visit abroad. I would also like to express my gratitude to Madhavi Krishnan and Andrew Bernat for assisting me with my questions and difficulties regarding Dyninst.
I thank Dr. Daniel Lorenz for guiding me during my thesis, and I would also like to thank Dominic Eschweiler, Dr. Markus Geimer, Marc-André Hermanns, Michael Knobloch, and Dr. Brian Wylie for listening to my questions and helping me out. My gratitude also goes to Zoltan Szebenyi for his advice and insight regarding experiments on JuRoPa.
Last but not least, I would like to thank Lena Walter, Christian Remy, and my parents for their continuous support and encouragement.

Chapter 1

Introduction

The super computers developed today continue to push for new limits in performance through the continuously growing number of integrated processors. To harness all of the theoretical performance delivered current and future High Performance Computing (HPC) applications need to use the computing power efficiently, something that requires both single core and parallel performance to be optimal. Achieving optimal parallel performance, and thereby not wasting expensive compute resources, has become especially difficult with the exponentially growing numbers of cores.

The process of optimizing HPC applications requires performance analysis tools to assist developers and deliver insight into aspects like work load distribution, communication behavior and other relevant performance metrics. In order to allow for an evaluation, it is essential that every optimization process measures the changes in the application's performance. There are many tools available that focus on different aspects of the performance and utilize different measurement techniques. Common to some of them is the need to instrument the application before performing an experiment.

Instrumentation modifies the target application, to enable the triggering of events, observable by the measurement system for its analysis. With the increasing size and complexity of current applications, the process of instrumentation has become more important for achieving constructive results. To execute the instrumentation compute time is consumed, to store the gathered data memory is occupied, and if measurement data exceeds memory limits, slow Input/Output (I/O) operations may become necessary. Therefore, to obtain constructive results instrumentation should not simply instrument every location possible but instead avoid less relevant ones and avoid those where the overhead excessively perturbs the experiment.

Static binary instrumentation is one of the multiple possibilities of instrumentation. It can provide some benefits over compiler based instrumentation, as it is executed after compilation and the binary code generated by the compiler is fully optimized. Instrumenting template generated code now equals instrumenting any other function, as they have been instantiated by the compiler. The process of binary instrumentation does require less modifications of the build process and thereby avoids otherwise necessary recompilations. When filtering is done before instrumentation, it reduces performance penalties because there are no changes to uninstrumented areas, as there would be if runtime filtering was used by the measurement system. Binary instrumentation is also language independent and allows instrumentation of executables and libraries where no source code is available, thereby enabling analysis of proprietary executables.

The generic instrumenter presented in this thesis is designed with two things in mind: The first one is to provide an instrumenter not tailored to one single performance analysis tool, but instead allowing for it, through appropriate configuration, to be used together with different measurement systems and their specific measurement Application Programming Interface (API). This will be made possible through exposing Dyninst's rich code generation capabilities to the tool developers. Secondly, focusing on rule-based filtering, delivering the necessary filter properties to be used in limiting the scope of the instrumentation, either to cover only parts of the application the user is interested in, or to remove those instrumentation points where the expected overhead will dilate the measurement severely, and the instrumented region itself contributes only little relevant information.

Properties used for eliminating small functions use the number of instructions, the Lines of Code metric, and their cyclomatic complexity. To select relevant areas, call graph information is exploited, allowing to gather more context information for relevant function calls.

During the course of the evaluation it is analyzed how the proposed filter properties are able to limit the scope of the instrumentation to achieve a measurement runtime with instrumented optimized binaries similar to the performance of the uninstrumented application. Using three C++ codes, which are two Cactus benchmarks plus DROPS, and one C code example, Gadget, the impact on different types of applications will be observed.

Chapter 2

Basics

This chapter first starts with introducing concepts of parallel programming, software solutions for parallel programming and a brief overview over current HPC hardware. It will then present the process of performance analysis and how Scalasca is used for performance analysis. The chapter closes with a description of how Dyninst enables binary instrumentation.

2.1 Parallel Computing

Following Moore's predictions the development of processors continues to double the number of transistors per processor about every two years. In comparison the clock speed with its current range of 3 GHz to 4 GHz has reached a level where power consumption becomes the demanding factor, limiting further improvements with today's technologies. Therefore, current developments favor increasing the number of cores per single processor. To benefit from multiple cores, applications need to do computations in parallel, distributing work onto more than one core.

Although there is bit- and instruction-level parallelism, the focus here is on applications using data and task parallelism. Data parallelism executes the same computations on the parts of the same data. In task parallelism, different threads or processes are allowed to carry out different tasks on distributed data.

When developing a parallel application or improving an already existing serial application, there are two important metrics to evaluate the benefits of parallelism. The first one describes the speedup achieved through the use of multiple processes, while the other one determines the parallel efficiency, comparing the speedup

gained to the additional resources consumed. The speedup is defined as followed, where n is the number of processes:

$$speedup(n) = \frac{T_{serial}}{T_{parallel(n)}} \tag{2.1}$$

The parallel efficiency is defined as, where again n is the number of processes:

$$efficiency(n) = \frac{speedup(n)}{n} \tag{2.2}$$

The efficiency of the parallel solution is limited by the amount of serial computation still present in the parallel application. This behavior was first described by Gene Amdahl in [1967] and became known as *Amdahl's Law*:

$$speedup(n) = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.3}$$

In which case P is the fraction of the application executable in parallel, and N is the number of processes. Although not feasible in reality due to constraints on the system, the maximum speedup for any application with N reaching $infinity$ becomes

$$\frac{1}{1-P} \tag{2.4}$$

When investigation the scaling behavior of an application, there are two types of scaling that are important to distinguish: An application can show weak scaling or strong scaling, whereas the latter one implies the first one. Weak scaling describes the time required to compute the solution for constant problem size per process. Contrariwise strong scaling describes the time to solution for a fixed problem size over all processes.

2.1.1 Message Passing Interface

The Message Passing Interface (MPI) [MPI Forum, 1994] defines a standardized language independent interface for parallel programming. Different MPI library

implementations exist, often tailored by the hardware vendors to meet their hardware requirements and to provide the best possible performance. The MPI realizes the concept of message passing between independent processes, whether they run on the same processor, compute node, cluster system, or possibly in a distributed environment.

The MPI defines functions to use for peer-to-peer communication and collective communication. Since version 2.0 it also defines an interface for one-sided communication using remote memory access. For synchronization purposes different barrier functions are available in the MPI.

2.1.2 Open Multi Processing

Open Multi Processing (OpenMP) [2008] is an API, which enables parallel programming on a shared memory multiprocessor system. The API itself is platform independent. OpenMP uses compiler directives, `#pragma` directives for C/C++, and directives in commented lines for Fortran, in combination with run-time libraries. It is today implemented by multiple compilers for different platforms including Linux, Windows, and MacOS. The focus of OpenMP in earlier versions was on loop level parallelism, dividing the number of iterations and distributing them over multiple cores. This results in an application where parallel regions and serial regions take turns.

Combining more than one paradigm of parallel programming in a single application, such as using MPI for communication between compute nodes and OpenMP within a single node, leads to an hybrid application model.

2.1.3 HPC Architecture

Nowadays super computers are for the most part build from smaller computers, so called compute nodes, each having their own processors and memory. These nodes are connected using interconnection networks, which also connect them to a shared storage. Inside each compute node are multiple processors, in some systems two, some feature four or more. Likewise, each of these processors consists of multiple cores, among which the compute node's memory is shared.

On a regular basis, the Top500 [2010] list presents the worlds fastest super computers, ranked by their Linpack [Dongarra et al., 2003] score. In June 2010, 81 percent of the systems used Intel Central Processing Units (CPU). 85 percent of the clusters

used quad-core processors and 5 percent featured six or more cores. The Jülich Research on Petaflop Architectures (JuRoPa) cluster, used during evaluation later on in this thesis, scored 14th place, whereas Juelicher BlueGene/P (JUGENE) ranked number five.

2.2 Performance Analysis

Software engineers follow the performance analysis workflow, depicted in Figure 2.1, to validate or improve their application's performance. The necessary steps include: modify the application to enable the measurement (instrumentation), execute an experiment and record the observations. Before the results are presented to the developer, they are sometimes analyzed automatically to identify known problems. After evaluating the results the developer decides whether there are performance inefficiencies and changes the application to improve the behavior. Finally the developers should verify that the changes indeed result in a better performance.

In some cases the results gathered are insufficient or present problems that originate from the instrumentation process. In these cases it is necessary to refine the instrumentation to obtain better results.

Figure 2.1: The performance analysis workflow illustrating how to evaluate, improve, and verify the performance of an application.

2.2.1 Performance Bottlenecks

There are different kinds of performance bottlenecks possible, on different levels within a single application. With respect to peak performance on a parallel system many factors add up to why an application does not achieve the system's theoretical peak performance. Reasons may be:

- too little optimization considering single threaded performance
- using unbalanced workloads where some processes are busy while others idle
- using a communication pattern that does not efficiently utilize the system's network topology
- algorithms and models that do not scale well with the growing size of the problem and the increasing number of processes

Uneven distribution of workload generally leads to one or more processes doing work while other processes do nothing, i.e., spend their time waiting for a synchronization event. Time spent by processes waiting for others is lost, reducing the application's overall performance.

Performance in parallel programming is affected by single thread performance, too. To achieve optimal single threaded performance is difficult, and mainly the focus of compilers or hardware vendor specific math library implementations, e.g., the Intel Math Kernel library. An example problem commonly used to elaborate single thread optimizations refers to matrix multiplication operations not being implemented efficiently in respect to cache layout of the CPU being used, as in Wolf and Lam [1991] or Lam et al. [1991], or not using available vector operations to speed up computations.

2.2.2 Direct Measurement and Sampling

One way of obtaining raw performance data is direct measurement. The application itself records events at specific locations in the application's execution. To record these events the application has to be modified, a process called instrumentation (see Section 2.3).

Sampling is another method of measurement when doing performance analysis. Sampling is triggered at defined time intervals, it then halts the program, analyzes

the current state and records an event describing the application's state of execution. Which information is recorded or how the interrupts are triggered depends on the measurement systems. Systems may measure at fixed time intervals or wait for hardware counters to trigger an overflow interrupt.

Some researchers consider sampling to produce a coarser picture, as it may miss events in between two triggered events, others argue its advantage of introducing a constant overhead compared to direct instrumentation, in reference to Shende et al. [2006], Tallent et al. [2008], and Adhianto et al. [2009]. In this thesis, the focus is on direct measurement and the thereby necessary application modifications.

2.2.3 Tracing

The word tracing is used when all observed events are recorded and stored for later analysis. This requires tool developers to consider efficient storage techniques for those events, to reduce the storage needed and avoid I/O intensive buffer flushes, if the number of recorded events exceeds the buffer's size. The final result of an experiment where tracing is used is an ordered stream of events that is available for deeper analysis.

2.2.4 Profiling

Profiling an application reduces the amount of stored data by aggregating equal events during a so called summarization run. Equality depends on the model used by the measurement system. For example, all calls done to a function `foo()` may be accumulated in a single record, where all performance metric values associated with the state are aggregated, if applicable. While aggregating all events of equal type, e.g., the same function, creates a flat profile, the aggregation is often done in respect to the current position in a call tree (see Section 2.2.5) to increase the granularity of the resulting profile. The accumulation into a single record loses information, e.g., when aggregating the time spent within a particular single function the original distribution over how much time is spent in it on each execution is lost.

2.2.5 Call Graph and Call Tree

The call graph of a particular program is constructed by creating nodes for every function in a program. Edges are then added for each call done from one function, the caller function, to another function the callee. In general the creation of a correct

and complete call graph depends on the available information and the point in time the graph is generated. One distinguishes the static and the dynamic call graph, where the static call graph is constructed utilizing only the binary executable or the source code whereas the dynamic call graph is generated at run time.

Depending on the information available the call graphs differ. Using the source code and information on inheritance one may be able to produce a more complete call graph than by looking only at the binary program. The binary may contain calls through function pointer constructs, originating from inheritance using virtual function tables or through function pointers used in languages like C and C++. These types of calls are not resolvable, since the addresses of the called functions are not available.

Recording the order of function calls while the application is running yields the call tree. For each function that the flow enters and exits a node is added to the tree. While generally the call graph consists of one node per function the call tree can contain many nodes for a particular function. For each node in the call tree, there is only one incoming edge, showing explicitly that the function was invoked from the preceding one. This may vary with the implementation, especially in regards to how recursion is handled. The call tree is often supplemented by user defined regions, which are marked in the source code.

2.3 Instrumentation

From the users perspective the application normally has no knowledge of any measurement system and does by itself not generate any events that can be observed by analysis tool relying on direct measurement. For these systems it is necessary to modify the application at some level, to invoke at some point the necessary measurement functions to trigger the events it observes. Instrumentation is the process of modifying the target application to create the measurement system specific events.

Some methods of modification are less intrusive and require almost no changes, while some others do require the user to recompile his program after making changes to either source code or the build environment. For example, a measurement system that observes time spent within the application's functions needs to modify the target application to trigger a measurement system event each time a function is entered and each time it is exited.

2.3.1 Manual Source Code Instrumentation

Manual source code instrumentation is a process where the application developers or the users responsible for the analysis themselves modify the source code at specific locations with the intent to trigger observable events. These locations can include: enter and exit sites of a function or enter and exit locations of a particular loop, but in general can be every location accessible through the source code. For the purpose of manual source code instrumentation measurement systems provide header files that define the interface to use to trigger particular events.

Instrumenting the complete source code manually is unfeasible for larger sets of events, e.g., marking all available function entry and exit locations.

2.3.2 Automated Source Code Transformation

To overcome the difficulties of instrumenting large portions of a program's source code manually, automated source code instrumenters have been developed. The source code is transformed according the requirements of the used measurement system, placing the necessary measurement calls at locations it wants to observe.

2.3.3 Automated Compiler Supported Instrumentation

Different compilers support compile time instrumentation of function entry and exit sites. The compilers provide interfaces to functions, which the measurement system needs to implement, and then places calls to those functions at the respective entry and exit sites. At which time during compilation the instrumentation is actually inserted can differ between compilers.

GNU Compiler Instrumentation

The GNU compiler provides two means of instrumenting a program [2010]. Using the compiler flag `-finstrument-functions` it will place calls to the measurement system at the enter and exit site of any function. The measurement system must provide functions according to the following GNU defined interface:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

To restrict which functions to instrument, the GNU compiler allows for files or functions to be excluded from instrumentation using the following compiler flags:

```
-finstrument-functions-exclude-file-list
-finstrument-functions-exclude-function-list
```

As an alternative the GNU compiler in combination with the linker supports the definition of wrapper functions. Calls to a wrapped function in the user code are then redirected to the wrapper function. The wrapper functions itself executes the original function and passes its result to the user code. This enables the measurement system to gather information about functions that themselves cannot easily instrumented, e.g., system library calls.

2.3.4 Library Interposition

Library interposition follows a similar approach to wrapper functions. It too places wrapper functions between the original application making the call and the library containing the called function. Library interposition interferes at the linking step for static binaries, changing the order in which the libraries are linked with the application. By changing the order, the original call is redirected to the measurement system. In case of dynamic libraries loaded at run time, loading a library through the `LD_PRELOAD` directive gives the user the ability to interfere with the order of how symbols are resolved.

Library interposition is often used in HPC performance tools to wrap MPI functions. The MPI libraries, although not all libraries, provide weak symbols in their library allowing tool developers to provide their own library containing MPI wrappers. The MPI libraries provide two symbols for each function, one weak symbol prefixed by MPI and one defined symbol prefixed with PMPI respectively. If no measurement library is present, the MPI symbol resolves to the appropriate PMPI symbol. However, if another library loaded earlier provides the MPI symbols, calls from user code to any MPI function will use those functions instead. Providing the PMPI symbols allows tool developers to execute MPI calls in their measurement code, which are not affected by the interposition. This is illustrated in Figure 2.2.

In case of MPI there are sometimes more symbols to cover when comparing C/C++ code and Fortran code. Different compilers tend to produce different symbols, where some are mangled and some are transformed to use all lower or upper case letters. Prefixing or suffixing with _may occur, e.g., the _suffix is found with all lower case symbols if gfortran is used. To work with all those symbol varieties the wrappers have to either include all of them or provide a library which directs all lower case calls to the C `MPI` symbols, as it is done by the MPICH2 library.

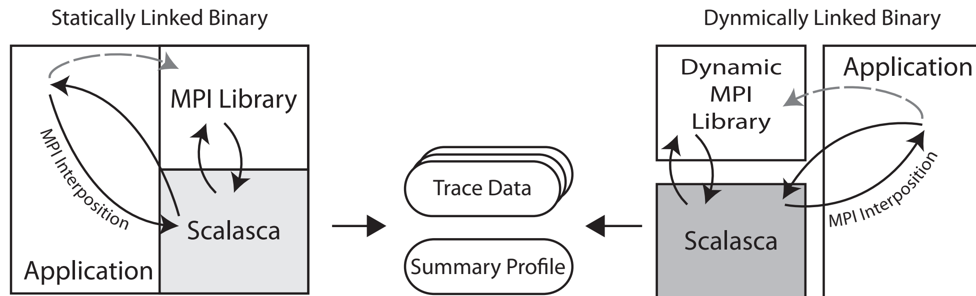

Figure 2.2: MPI library interposition in the static case, currently employed by Scalasca, and in case of a dynamic MPI and Scalasca library.

2.3.5 Static Binary Instrumentation

Static binary instrumentation means modifying an existing executable or library and rewrite it to include the instrumentation instructions. This approach is possible even if no source code is available, for example to analyze proprietary vendor libraries. To be able to use static binary instrumentation, it is necessary to support different executable file formats and to support the platform specific instruction sets.

2.3.6 Dynamic Binary Instrumentation

In contrast to static instrumentation, dynamic binary instrumentation is done at run time. A mutator, a piece of software - not necessarily an independent application - is responsible to execute the modifications. While the instrumented or modified process is called the mutatee. Therefore, dynamic binary instrumentation may require a mutator present on all compute nodes in a cluster environment to do the modifications and communicate with other mutators for decision making.

Binary instrumentation is architecture dependent and safeguarding has to take place to make sure the programs correctness is not affected by the modifications. This makes it more difficult than e.g. source code instrumentation where one can expect the compiler to produce code unaffected, with respect to produce the correct solution, by the instrumentation.

2.3.7 Effects of Instrumentation

Common to all types of instrumentation is the introduced overhead into the application. After instrumentation there is additional computing time required to execute the instrumentation. Additionally, more memory necessary to store measured data, especially in those cases where tracing is enabled. The measurement system itself my need some kind of communicate, increasing the strain on the communication system.

Instrumentation, on source code or compiler level may also change the way the application is optimized. Therefore, the observed performance may differ from the one expressed by an unmodified application. For compiler instrumentation it varies with the type of compiler used, whether optimization is affected or not.

When using binary instrumentation or in other cases to provide more detailed information at run time, it is required to have debug information available in the binary. Debug information stores among other things stores information regarding which part of the binary originates from which source file. This increases the binary's size, but contrary to common believe does generally not affect the programs performance. Nevertheless, some compilers reduce the default level of optimization if the debug flag is enabled, probably producing a different result, if the user does not explicitly specify the level of optimization.

2.4 Dyninst

Dyninst[2010] is a library originally developed for dynamic instrumentation at the University of Wisconsin, Madison and the University of Maryland. Static binary instrumentation has been added for the `x86` and `x86_64` platform to modify dynamic binaries. It is planed to add static rewriting of statically linked binaries in the near future and to add support for the PowerPC platform, too. With Intel Xeon CPUs dominating the Top500, and AMD Opterons present in 3 of the top 10 systems, both using the `x86_64` instruction set, binary rewriting is possible on most of the systems, excluding however the JUGENE system running IBM Power CPUs.

Dyninst provides the developer with the C++ Dyninst API, which allows to interact with a running processes, to modify a processes and also to open, to modify and to save an existing binary. It provides a high level API to the developer to specify both what to execute and where to place it, thus no deeper knowledge of assembler of the underlying executable binary file formats is necessary.

The Dyninst developers are building new libraries by extracting existing and in itself independent functionality from the original Dyninst library [Ravipati et al., 2007]. These provide interested users for example with the Symtab API, for symbol table access, the Instruction API, or the Stackwalker interface, for stack analysis of running applications.

Other tools and libraries that are able to either interact with a running process or to modify an existing binary for other platforms include Etch for Windows/x86 binaries, EEL for the Sparc platform or PIN to modify running processes on most Intel platforms.

2.4.1 Binary Access and Program Abstraction

Loading an existing binary executable with the Dyninst API provides access to modules, functions and the internal structure of the functions. Modules try to group functions by either the files they were defined in, if debug information can be found inside the binary, or by the executable or library they are contained in.

Looking at a single function, one gets access to a control flow graph consisting of basic blocks and edges representing jumps from one basic block to its successors. Additionally a tree representing all loops within the function is available. Nodes in the loop tree make it possible to query what functions are called inside a loop and which basic blocks are executed within the loop body. Inspecting a single basic block, Dyninst can provide additional information about instructions executed, e.g., whether they read from or write to memory.

Looking back at the function object, it provides access to the function identifier and a module within which it is contained. If debug information is found, one can access file name and line number information, through use of the function address or using the address of a specific basic block. The function name is available in its mangled and demangled form.

2.4.2 Supported Instrumentation Points

Dyninst can insert code into the binary at different types of instrumentation points. It can insert code at function entry and exit locations. Within functions it can instrument loops, which allows to place code before and after the loop and at enter and exit points of the iteration. In addition Dyninst allows code to be inserted at function call sites, surrounding the call site. Dyninst provides access to the control

flow graph, represented by basic blocks and edges between blocks, and allows the instrumentation of its edges.

Depending on the site of the instrumentation, access to, e.g., call parameters or return values is possible.

2.4.3 Instrumenting Binary Code

To insert the instrumentation at a particular point in the binary, Dyninst in general does the following:

- Create the Base Trampoline
- Copy instructions at target location to Base Trampoline
- Place jump instruction to Base Trampoline at location
- Create the proper Mini Trampoline
- Generate instrumentation code and place jump instruction in the Mini Trampoline

Frequently some instructions of the binary code at the designated location need to be copied. This is necessary as the x86 instruction set does not have a fixed size, thus there may not be enough space to place the jump instruction at the target location.

The Base Trampoline is made out of three sections. The first section, targeted by the jump at the original code location, makes a jump to the Mini Trampoline. The second section is made out of the original code, which will be executed after the instrumentation has executed. The last section jumps back to the original code location, or to the location immediately after the copied code, to continue with the execution (see Figure 2.3).

The Mini Trampoline contains what is necessary to execute the instrumentation. If needed, there is a section to store registers that are modified by the instrumentation. In addition, there is code to setup arguments for the instrumentation code. This is followed by code to execute the instrumentation and the code itself. The inserted code fragment is enclosed by instructions restoring previously saved registers, reverting to the state of execution before entering the instrumentation, and a jump returning to the base trampoline. Due to the copying of parts of the executable and newly added coded variables the size of the binary increases with more functions or locations in general being instrumented.

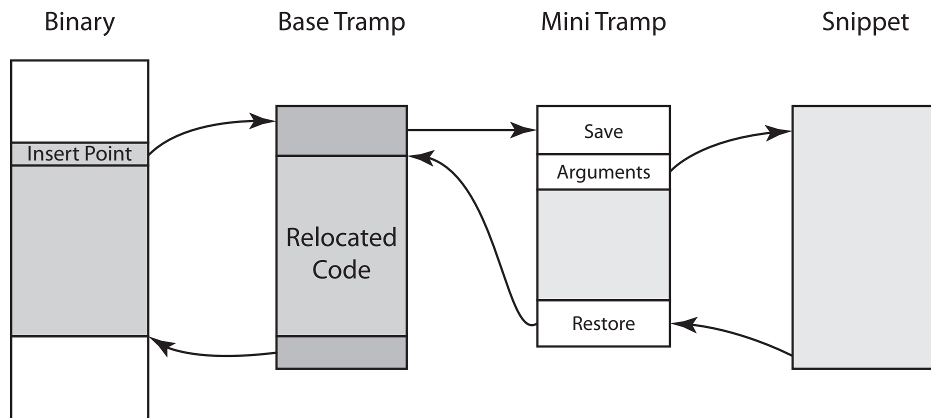

Figure 2.3: Dyninst instrumentation with Trampolines, showing relocated code, the Mini Trampoline for saving and restoring registers and the inserted snippet.

2.4.4 Code Representation

As the Dyninst API itself is platform independent, the ability to modify processes at run time extends to platforms other than the `x86` platform, it provides abstractions for the code to be executed. Therefore, the Dyninst API provides various classes for different code constructs. The developer builds an abstract syntax tree (AST) with objects of theses classes, that is at last compiled into the platform specific machine instructions.

The base class for these constructs is the `BPatch_snippet` class, whereby Dyninst uses the term snippet to describe any code generated. Derived classes include the `BPatch_funcCall` class to call functions in the binary, `BPatch_arithExpr` to execute an binary arithmetic expression, or `BPatch_sequence` to create a sequence of other statements.

2.5 Scalasca

Scalasca is a set of tools to analyze the performance of HPC applications. Its analysis focuses on the run-time behavior of large scale parallel applications, running on many thousands of cores, with the goal to uncover inefficiencies in communication patterns [Wolf et al., 2010]. The Scalasca tools permit the user to perform run-time summarization experiments or to enable the tracing of all events.

The recorded trace files enable Scalasca to do a parallel replay of the application's communication events to detect known patterns of communication inefficiencies [Geimer et al., 2009b], e.g., expressed in wait states or remote MPI-2 memory access [Hermanns et al., 2009]. Patterns of extended wait periods, often a result of an unbalanced workload across the system, are challenging when scaling applications to large parallel systems. The parallel replay, provided by Scalasca's SCOUT utility, improves the analysis performance of large data sets of process-local traces, which are generated by applications running on thousands of cores. Scalasca exploits the same processing power, using the same number of cores that was available to the application, whereas the predecessor KOJAK executed a sequential trace analysis.

Scalasca is currently developed at the Jülich Supercomputing Centre (JSC) and the German Research School for Simulation Sciences (GRS) and is freely available under the BSD open source license. Scalasca has been used in experiments at the JSC, running on up to 292,914 cores on the IBM Blue Gene system JUGENE.

The Scalasca system is at different levels compatible with other performance analysis tools. It can benefit from the TAU source code instrumenter capabilities or allows result presentation in tools like Paraver and Vampir by converting its measurement results. Scalasca is not limited to MPI analysis, but features the ability to work with OpenMP and hybrid applications employing both paradigms for parallel programming. It supports Fortran, C, and C++ source code, provides wrappers for the most important MPI libraries, and supports compiler instrumentation with different compilers like Intel and the GNU Compiler Collection (GCC).

2.5.1 Workflow

To analyze an application with the Scalasca tool set it is necessary to rebuild the application to add the instrumentation and MPI measurement capabilities. Executing the application is handled by the workflow managers, the `scan` application, which sets up the environment by reading the input parameters for, e.g., filter file, a directory where to save the result,s and whether or not tracing should be enabled. After a successful measurement run, the user finds within the measurement directory files containing the results. One file stores the program and measurement system output, including setup parameters. Another file stores a plain text representation of the observed call tree. When tracing is enabled, Scalasca archives the trace files for all involved processes separately. The profile results are saved within a cube archive file, which the user can later on analyze using the various *cube* tools delivered alongside Scalasca. For graphical representation and browsing of the results the user can resort to the cube3 viewer, which will in detail present the different metrics, a call tree or flat view of the call tree nodes and a topology showing the distribution of metric values across the processes.

2.5.2 Instrumentation

The Scalasca instrumentation process integrates into the build process of the target application. The user has to prepend his compile and link commands with the `skin` command. With command line parameters, the user is able to select the type of instrumentation, e.g., enabling compiler instrumentation or manual user instrumentation. Additional options are available to enable OpenMP instrumentation.

The function entry and exit instrumentation relies on the compiler's capabilities, e.g. as described in case of the GNU compilers in Section 2.3.3. Not all compilers do support this kind of instrumentation, and not all compilers express the same behavior with respect to instrumentation and optimization. In some cases optimization is not as effective or even disabled when instrumentation is requested.

To measure MPI communication events, Scalasca interferes at the linker level in the build chain. By reordering of the included object files, the Scalasca object files, containing the MPI wrapper functions, are placed between the user code objects and the MPI library. The Scalasca MPI wrappers themselves use the PMPI interface to make their calls to the MPI library.

```
EPIK_USER_REG(r_name,"region");
EPIK_USER_START(r_name);
EPIK_USER_END(r_name);
```

Code 1: Scalasca's macros for manual source code instrumentation.

Besides the compiler supported instrumentation, the user can mark regions of interest with predefined macros in the source code (see Code 1). This creates regions that will appear in the call graph as new nodes alongside those created by function instrumentation or the MPI wrappers. The start and end macros expand to function calls invoking `EPIK_User_start()` and `EPIK_User_end()`. Both of them take the following parameters: file name, function name, and line number. This allows to map any recorded data to the source code region. These two functions will be the ones used in conjunction with the generic binary instrumenter to mark function enter and exit locations for the Scalasca measurement system.

Scalasca's measurement system architecture, presented in Figure 2.4 and described in more detail in Wylie et al. [2006], uses different event adapters, each one responsible for either user instrumented regions, compiler generated function instrumentation, OpenMP instrumentation, MPI Library interposition, or Global Arrays (GAS). The event handlers are responsible for either summary creation (EPITOME), or trace recording (EPILOG). Support for creating Open Trace Format (OTF) files is built into EPI-OTF.

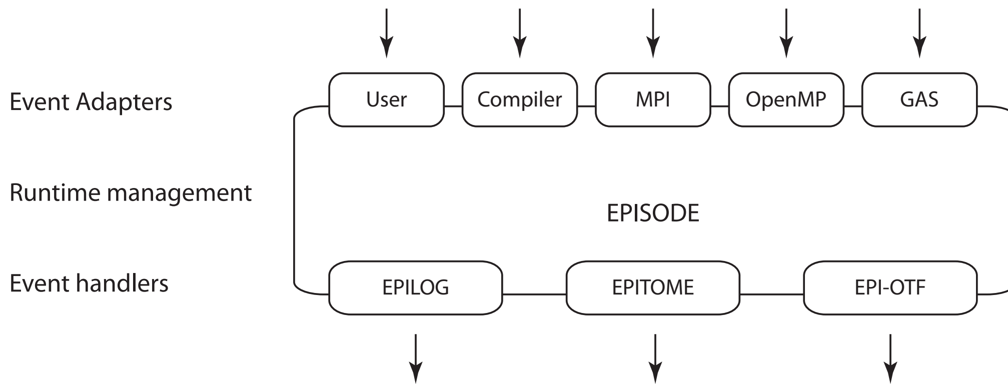

Figure 2.4: Scalasca's EPIK run-time library architecture, showing the different event adapters.

2.5.3 Filtering

To influence the impact of instrumentation, the user may provide a configuration file, specifying which functions he wants to exclude from being measured. However, this filter is invoked at run time, where the measurement system decides whether or not to record the event. The compiler instrumentation is unaffected by the filter.

To assist the user with the decision which functions to filter Pfeifer [2008] analyzed how functions could be classified according to their call paths with respect to MPI functions and how to exclude functions by the frequency at which they are invoked. The `cube3_score` tool analyses the results of a profiling run, do determine which functions lead to MPI calls. Those are classified as COM functions, functions immediately preceding MPI calls are WRP functions, and the remaining functions are classified as USR functions. It also simulates the amount of memory necessary to trace all events, by multiplying the size needed to store a particular event with the number of occurences. The information about memory requirements, frequency of execution and whether or not the functions are classified as COM functions aid the user in deciding which functions not to measure.

The classification of COM functions opposing USR functions is influenced by the primary goal of Scalasca, which is to analyze communication behavior in large-scale parallel applications. Hence, it is necessary to track MPI communication events, including their calling context. Adding the call path information assists the developers in pinpointing under which conditions performance problems occur.

Because Scalasca invokes compiler instrumentation, it is possible to modify the build process to disable particular files or functions using compiler provided mechanisms, this however increases the user's effort required to limit the instrumentation. For selective instrumentation it is also possible to use the TAU system (see Section 3.1), redirecting its calls to the Scalasca measurement API (mentioned in Geimer et al. [2010]).

Using Scalasca to analyze several codes, like Sweep3D by Wylie et al. [2010], the SPEC MPI Benchmark suite by Szebenyi et al. [2008], the XNS Solver by Wylie et al. [2007], and PEPC by Szebenyi et al. [2009] has shown that filtering may not be necessary for all investigated applications, because the dilation is within reasonable limits (mentioning a 5 percent mark in Wylie et al. [2010]). In other applications it may however become necessary to reduce the overhead and the amount of generated trace data.

Chapter 3

Related Work

Deciding whether or not to instrument a particular location, such as a particular function's entry or exit location, is difficult without knowing the focus of the user's investigation, not knowing the resulting perturbation in relation to uninstrumented performance and not knowing how often an event will be triggered. Hernandez et al. [2007] use compile-time information provided by the OpenUH compiler[Liao et al., 2006] for that decision. The decision is based on the number of instructions, the number of call sites, and where a particular function is invoked. Functions invoked in inner loops are less likely to be instrumented.

For example, TAU (see Section 3.1) and Scalasca approach the decision by first doing a fully instrumented run. The user then has to use the gained profile data to generate a filter to exclude particular functions. In case of TAU, the filter can be created using an extra tool that analyzes the profile data. Whether the filtering is done at run time or at compile time depends on the measurement system.

On the other hand the call-graph-based profiler CATCH by DeRose and Wolf [2002] uses Dyninst and the Dynamic Probe Class Library (DPCL) [DeRose et al., 2001], which is built upon Dyninst, to insert its instrumentation probes into the running application. By default it supports a restriction to functions deemed interesting. These are functions that are part of call paths leading to MPI calls or to OpenMP parallel regions. CATCH also allows the user to instrument additional subtrees or remove them from the instrumentation. CATCH generates a profile where the observed call tree nodes are associated with hardware performance counter metrics. This paints a more detailed picture than tools limited to record the number of executions per function and the time spent within a particular function.

DynaProf and Paraver are two examples of tools making use of the DPCL to dy-

namically instrument and profile applications. Both tools support the analysis of MPI, OpenMP, and hybrid applications. For OpenMP applications, Paraver uses DPCL for instrumenting parallel regions, and the enclosing functions. If requested, user defined functions whose names are provided in a separate file, are instrumented, too.

Two approaches to reduce performance perturbation are: to reduce the amount of instrumentation inserted and to use run-time filtering to further reduce the number of measured events and the generated data. Another different approach is to try to lower the perturbation's influence on the measurement result by overhead compensation. Models for compensating overhead on a per-process-basis are evaluated in Malony and Shende [2004] and further improvements for parallel applications are discussed in Malony et al. [2007].

Adhianto et al. [2009] argue that direct measurement using instrumentation introduces significant overhead and the employed filter techniques introduce blind spots in the observed behavior. They conclude that statistical sampling combined with binary analysis yields a more accurate picture and allows for better bottleneck detection and its localization. The common approach of removing small functions is especially criticized, because they may, e.g., include thread synchronization.

The ROSE compiler framework, introduced by Schordan and Quinlan [2003], provides many of the features necessary to do source-to-source transformations to insert instrumentation hooks into an application's code. ROSE supports C, C++, and Fortran code.

3.1 Tuning and Analysis Utilities

The TAU measurement system [Shende, 2006] is developed at the University of Oregon, Portland. It supports multiple languages including, among others, C++ and Java. TAU features an instrumentation layer that can observe a broad range of events. These events include routine or code region enter and exit events, system generated events, library events, and user-defined events. TAU is also able to analyze applications using MPI or OpenMP.

3.1.1 The TAU Source Code Instrumenter

The TAU source-code instrumenter supports automatic instrumentation of C, C++, and Fortran code. TAU includes the Program Database Toolkit (PDT) [Lindlan

et al., 2000], which provides a parser front-end for C, C++ and Fortran. The parser generates an intermediate language (IL) representation of the parsed source code. This IL data is then analyzed and stored in a PDB database file. The source-code instrumenter reads the PDB file using DUCTAPE, a library exporting a C++ API to access the PDB database, and creates list of target locations to be instrumented. At which point the filter configuration is evaluated, which includes a list of files and a list of functions not to instrument. The filter configuration also contains the settings regarding whether to instrument particular code regions, such as loops and functions. The application's source files are read line by line in the final step and the TAU API calls are inserted at the selected locations.

3.1.2 The TAU Binary Instrumenter

The TAU binary instrumenter is specifically tailored to be used in conjunction with the TAU measurement system [Shende et al., 2001]. It first adds the TAU measurement library, which is available as a dynamic library to the binary. Secondly, it instruments all functions, for which the source files are of either C, C++ or Fortran type determined by their file extension. It removes functions inside the TAU shared library from instrumentation. The instrumentation places calls to the TAU API at function entry and exit points.

The TAU measurement system needs to create unique identifiers, because its API requires these to record region entry and exit events. Thus the application's `main` function is instrumented with a special function call that informs TAU about which region is associated with which unique id. The instrumenter supports user defined filtering by black listing functions and file names in a separate configuration file.

3.2 The OPARI Source Code Instrumenter

OPARI is a source-to-source translation tool used to instrument OpenMP applications [Mohr et al., 2001]. Both Scalasca and TAU use OPARI for their instrumentation. During instrumentation, OPARI transforms OpenMP constructs according to defined rules, which in most cases adds function calls to the POMP profiler interface immediately before or after an OpenMP directive. The POMP interface is designed to observe OpenMP applications performance. The POMP interface has to be implemented by the tool used, thus Scalasca features its own POMP event adapter (see Figure 2.4).

3.3 Generic Source Code Instrumentation

Geimer et al. [2009a] introduce a generic source-code instrumenter. The generic source-code instrumenter's purpose is to extend the usability of the TAU source-code instrumenter to a broader range of measurement systems. Thus it removes the fixed code fragments calling the TAU API and replaces them with user specified code. To be more general and applicable basic constructs are identified, which would be needed to instrument an application. They specify six *building-blocks* necessary to transform existing source code:

- `entry`
- `exit`
- `decl`
- `init`
- `file`
- `abort`

The `entry` and `exit` constructs define code fragments to be inserted at entry and exit locations for routines. To limit the instrumentation's scope, not only they but also the other elements support two attributes: `file` and `routine`. These define exactly which files and functions will be instrumented. The default behavior is to instrument all files and functions. To cope with the possible presence of different languages, one can create separate language dependent specifications by exploiting the `lang` attribute.

The `decl` construct enables the user to place variable declarations in the instrumented file. With the help of the `file` element additional source files can be included. For example files belonging to the measurement system, such as header files containing the API definitions, leaving the `init` element to execute one time code for necessary initializations. If clean-up work is needed, the `abort` element places the specified code fragments before calls made to `exit` or `abort` functions.

For the executed instrumentation to be useful it needs to transfer context information about where it is called to the measurement system. For example, an enter event is of more use if it tells the measurement system which function is entered. The generic source code instrumenter supports special identifiers (enclosed in `@`), which the user can reference in his code fragments. These are then replaced with the requested information during the insertion into the source code. In case of the

information available at compile time being insufficient, `@RTTI@` enables access to dynamic routine names.

Putting it all together Geimer et al. [2009a] provide the following example for the Scalasca measurement system:

```
file="*" line=1 code="#include <epik_user.h>"
entry code="EPIK_User_start(\"@ROUTINE@\", \"@FILE@\", @BEGIN_LINE@);"
exit code="EPIK_User_end(\"@ROUTINE@\", \"@FILE@\", @END_LINE@);"
```

3.4 Paradyn

Paradyn is a performance analysis tool, which uses dynamic instrumentation to locate performance problems at run time [Callaghan et al., 1994]. To identify a probable performance problem, Paradyn features the W^3 Search Model that was developed by Hollingsworth and Miller [1993].

The W^3 Model is influenced by three questions regarding performance bottlenecks: *Why*, *where*, and *when*. *Why* focuses on why there is a problem, *where* tries to look at the location in the application, and *when* observes when the problem occurs during execution. For example, the *where* axis tries to locate in which procedure the problem appears, on which processor, or which type of synchronization is involved, e.g., whether messages, semaphores, or barriers are the cause.

At run time the application is instrumented by the performance consultant. The performance consultant is responsible for exploring the search space defined by the W^3 Search Model and communicating necessary instrumentation commands to daemon processes that insert the instrumentation.

Additionally, Hollingsworth and Miller [1996] introduced an adaptive cost system that is designed to limit the measurement perturbation by not introducing expensive instrumentation and stay below a user defined threshold. To compute the expected costs, the frequency of execution is at first guessed but refined by the available data during the experiment.

The automated search for performance problems is improved by Collins and Miller [2005]. They add loop level instrumentation improve the granularity of the measurement and show that precision and search time for particular bottlenecks improves.

3.4.1 Metric Description Language

The Metric Description Language (MDL) [Hollingsworth et al., 1997] is designed to issue instrumentation requests and is part of the Paradyn tool. A parsed MDL file yields both what to instrument and how to instrument it. Inserted instrumentation is limited to primitives and predicates. A primitive is a simple operation that changes a counter value or a timer. A predicate is a boolean expression deciding whether or not to execute a primitive.

MDL limits the number of instrumentation points using constraints to select different program components. Components include procedure, file, or process and are identified using unique names. The components are grouped together in a resource hierarchy, thus procedures are collected in modules, which are themselves collected in the `Code` resource. Hence, `foo()` in `foo.c` becomes: `/Code/foo.c/foo`. MDL then uses a `foreach` statement to iterate over items in a resource and inserts the instrumentation.

In general MDL is used to define performance metrics, based on the defined primitives that are the inserted according to the issued constraints.

3.4.2 Dyner

Dyner, introduced by Williams and Hollingsworth [2004], is a Tool Command Language (Tcl) based command line client for the use of Dyninst API features. It contains its own parser for code statements, which translates them to the Dyninst representation and inserts them into the target application. Dyner enables the user to connect to running processes for their modification; static modification of executables is also possible.

Its ability to load batch scripts containing Dyner commands makes executing common or repeating tasks more manageable. Dyner additionally supports exploratory features. This enables the user to inspect a target binary or process in more detail. The user gains access to modules, procedures or variables present using commands like 'show', 'whatis', or 'find function'.

Similar to a debugger, Dyner can take control of a process and halt it, if required by the user, to add or remove instrumentation, show current variable values or make a new library available in the mutatee's address space.

3.5 Conclusion

Each of the presented tools and filter techniques cover some of the possibilities for instrumentation and filtering. The generic source code instrumenter delivers a good insight into what would be necessary to support with a generic binary instrumenter, but its filter capabilities are limited. Also missing is the ability to specify different code fragments for different types of regions for instrumentation.

MDL code-generation features are not sufficient to make function calls to the measurement system API. It also merges filter and code into a single file, exposing more complexity to the user than possibly necessary. With its goal to describe performance metrics, not only instrumentation, it does also include more features that go beyond the instrumentation process. Dyner allows to execute specific batch scripts containing user defined code but does not feature a richer set of filters nor does it provide a flexible way to include context information in the user code.

The tools that allow users to define filters depend on explicitly naming functions, although some support patterns using wild cards. Such filtering is, with exceptions, unable to describe filters that are applicable to more than one application.

The CATCH profiler illustrates possible benefits of providing call graph access, especially to query call paths to particular target functions, which allows a targeted analysis and suggests two possible targets for call path filtering: MPI calls and OpenMP regions.

The generic binary instrumenter therefore has to improve the filtering by providing rules that permit more than to exclude functions based on their names. Additionally, it must provide a more flexible code generation interface to be compatible with different measurement APIs. However, proposed properties are limited to knowledge gained by static binary analysis.

The OPARI instrumenter illustrates, with regard to possibilities of instrumentation, the necessity to select between different API calls for the same type of instrumentation points. For example, to call a region adapter at all function call sites, but call the OpenMP POMP adapter if the called function represents a parallel region.

Chapter 4

The Generic Binary Instrumenter

In this chapter the generic binary instrumenter will be introduced which was developed as part of this thesis. The generic in generic binary instrumenter focuses on not being tied to a single measurement system, but rather allowing tool developers to specify the code fragments they want to be executed at the different supported locations. The supported locations for this instrumenter cover: function entry and exit points, points surrounding loops and their body, supplemented by the ability instrument call sites, before the function call and immediately after the return. Adding to the configurability of the inserted code, it must be flexible as to how the user later can refine and make changes to which parts of the application will be covered by the instrumentation. Therefore, a filter definition was added which supports necessary filter rules in addition to means for combining these rules to form one single filter.

After giving a short system overview, the first part looks at the requirements for the filter and how filters have to be specified. It is followed by the second part, looking at what is necessary to be supported on the instrumentation side and how the insertable code fragments have to be specified. A closing look at how filter and instrumentation are implemented concludes the generic binary instrumenter's introduction. The last section will cover briefly how a shared library was produced to allow instrumenting binaries without the need for any recompilation.

4.1 Overview

To instrument a target application the developed instrumenter loads two different input files, filter and adapter specification, in addition to the target binary, illus-

trated in Figure 4.1. The adapter specification defines, possibly multiple, named code fragments which can be inserted into the binary. The filter specification contains a set of named filters, comprised of rules. By the user's request one or more of these filters will be evaluated, each resulting in a set of functions into which the instrumenter inserts the requested function, loop, and call site instrumentation.

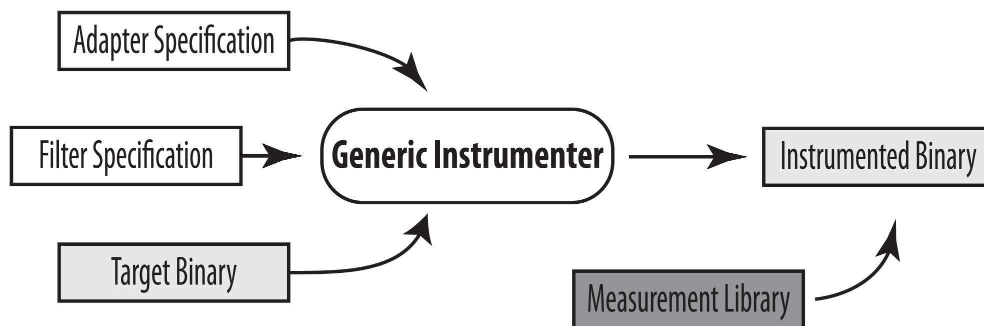

Figure 4.1: Input to and output of the generic binary instrumenter.

Depending on the measurement system and the instrumentation the result will be a binary, instrumented with calls to a measurement system and possibly a newly added shared library dependency, which includes the measurement system.

4.2 Selective Instrumentation

HPC applications today have grown in code size, functionality, and use of existing frameworks. As a result of this, the total number of functions has increased. Many analysis tools use function entry and exit as locations where instrumentation is inserted, thus with the increasing number of functions, the potential overhead increases, too.

With frameworks in mind, the user may have no interest in instrumenting the framework part of the application, whereas she is especially interested in her part of the application. However, the binary instrumenter may provide a more detailed look at things happening inside a closed source library linked with the analyzed program, if required. In both cases the user needs a way to limit the scope of the instrumentation.

Software engineers developing applications in C++ generally use templates to create their own generic code or use templates contained in the Standard Template Library (STL). The use of container classes declared in the STL, such as `vector` or `map`, forces the compiler to instantiate many templates. This creates a lot of small functions for the different types used in the user code. Using object oriented paradigms may in addition introduce small functions, e.g., get and set functions for member variables. The perturbation created by instrumenting small functions is more severe than it is for bigger functions, therefore, reducing the overhead requires a filter that is able to remove these small functions.

Tools today support different filter mechanisms (see Section 3), however, they often rely on an initial summarization run to identify exactly which functions should better be left out. The filters are often based on white and black listing, although wildcards are available, requiring increasing efforts by the user as code size and the number of excluded functions grow.

Therefore, the available filtering mechanisms should provide an easier way of limiting the instrumentation to particular areas of interest and to exclude functions where the expected overhead may dominate the function's execution time. The decision regarding introduced overhead must rely on data available without an initial summarization run.

4.2.1 Implemented Rules

This section will introduce, in more detail, the different types of rules available to the user and tool developers. The available set of rules is separated into two distinct groups. One group, named patterns, uses string matching to compare user specified patterns with function identifiers or the module names (possibly containing the file name). The other group of rules is called properties and it exposes more detailed information about the functions and the binary as a whole.

All patterns share the ability to specify how to compare the given pattern with the identifier in question. One available method is to compare with respect to equality for a complete match of the identifier and the pattern (for example to single out exactly one function). The alternatives include to match only substrings, either prefix or suffix comparing the identifiers first or last characters, respectively, or to check whether the pattern is contained within that particular identifier. For more flexibility the use of regular expressions is possible as well.

Module Name Patterns

The module name pattern corresponds to the module name provided by the Dyninst API. The module name contains the name of the file where the function is defined, when debug information is available. If that is not the case, or the information is not found for that particular function, the module name contains the name of the library where the function resides or it contains `DEFAULT_MODULE`.

If debug information is present, this rule can be used to quickly identify user code. This can be achieved by matching the filename suffix against common file endings, e.g., *.c, .cpp, .f,* and so on. Non-user code, which often includes system or other libraries, does normally not include debug information and thus includes no file name in the module name.

Function Identifier Related Patterns

Patterns to query the function identifier (or parts of it) are required to identify particular functions and are used as a starting point for properties later described. Because most tools today use white or black listing, with some kind of wildcard, the provided patterns can be used to provide an equal set of features.

To provide easier access to sections of the function identifier, which is relevant for C++ code where the function identifier may contain more than just a function name, the instrumenter supports additional rules: namespace, class name, and function name. Because there are multiple possibilities how the namespace and class name can be constructed, there remains some ambiguity. For example a C++ identifier can either provide `namespace::funcionname` or `classname::functionname`.

Any identifier is made out of a set of tokens, where tokens are joined by `::`. When splitting the identifier into tokens care has to be taken, because the separator `::` may be present due to template instantiations, where it is part of the type name. Hence, the namespace rule provides access to all tokens but the last one, which is always a function name. The class name returns the second to last token, which may indeed be a namespace qualifier. The function name rule exposes only the last token.

Selecting Short Functions

The following three properties, number of instructions, Lines of Code, and cyclomatic complexity are implemented to identify small functions. A function is considered small and thereby less relevant if it contributes very little to the overall execution time. These small functions are prone to create a lot of overhead, because the instrumentation consumes more time than the function itself.

These properties are independent of the user's code, because they do not rely on function names or namespace information. They may therefore be useful independent of the used language and may allow the tool developers to define a default filter that excludes a set of functions that show extensive overhead.

Selecting short functions is difficult. Simply from the following three metrics one cannot with certainty determine that a function will indeed require little run time. A loop may still be present, whose number of iterations may be big enough and the included computation expensive. Whether the function really introduces an overhead that perturbs the total measurement, does also not only depend on the relation between its run time and instrumentation's run time, but also on the number of its executions.

Number of Instructions

The first option for identifying short functions is to count the number of instructions in the function and to exclude those where the value is not within the user specified range. This property counts the Dyninst API instructions in basic blocks, and does not take into account any difference in their computational costs.

Lines of Code

The Lines of Code metric enables the user to remove functions from the ones instrumented based on their length in code lines.

The user specifies a minimum and a maximum value for the Lines of Code property and the instrumenter excludes all functions which are not within that range. It is necessary that the binary includes debug information to access the line number information. The Lines of Code values are calculated using the function's entry and the function's exit points' addresses. Using the Address, Dyninst can retrieve the associated file and line number.

The Lines of Code filter may not reflect the real size or complexity of the function in the binary. Due to inlining by the compiler, the original function may be short in source code, but show code from other functions in the binary. It may also be affected by the code style employed by the application developers.

Cyclomatic Complexity

McCabe [1976] introduced the cyclomatic complexity as a graph metric describing the decision complexity of Fortran codes. It calculates the number of paths through a function's control flow graph.

The cyclomatic complexity for a single function is defined as:

$$M = E - N + 1 \tag{4.1}$$

where M equals the cyclomatic complexity, E equals the number of edges in the flow graph, and N equals the number of nodes in the graph. The higher the cyclomatic complexity, the more complex a function is assumed to be, due to its more difficult to understand branch structure.

In cases where the Lines of Code property is not available due to missing debug information or the user believes the Lines of Code metric is not accurate enough, because of the code style, the cyclomatic complexity may yield a more useful value that is not affected by the code style. To limit the functions instrumented, the user specifies a minimum and maximum value, and function exhibiting a cyclomatic complexity within that range are returned by the property.

Implications of Inlining and Code Style Both properties, Lines of Code and cyclomatic complexity, may prove less useful with modern coding practices and compiler optimizations. For example, in software engineering, one is taught to reduce a function's length and its complexity and instead favor shorter and easier to understand functions by following Fowler's [2000] ideas of code refactoring. Therefore, the Lines of Code property may not be restrictive enough or too restrictive.

Lines of Code may also be less useful if optimizers use inlining. The function itself may be defined with a few lines of code, but due to other functions being inlined, the observed execution time is no longer directly related to the value of the Lines of Code metric.

Binaries compiled with enabled optimization may show a reduced number of functions with low complexity and less functions with low Lines of Code, because inlining replaces function calls with the code of the called function.

Call Path

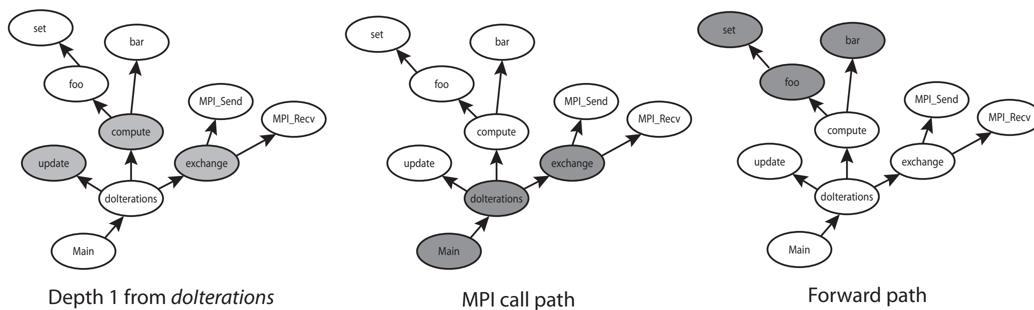

Figure 4.2: The different properties using the static call graph: Depth and call path in forward and backward direction.

The call path property is implemented to select those function necessary to describe the full calling context of functions interesting to the user or the measurement system. For example, Scalasca would use the call paths to MPI function calls to accurately record their calling context.

The user can define a set of target functions, using rules to describe the target set, and the call path property yields all functions that lay on possible call paths leading up to the target functions. Additionally, the call path property is implemented for the forward direction. This enables the selection of all functions that are called after a set of target functions. The intention is to enable the user to observe parts of his application, especially those that are visited during some target function's execution; Figure 4.2 illustrates this.

The instrumenter creates a static call graph by analyzing the binary. This may yield functions without any callee function, thus having no incoming edges, because function pointers and virtual functions are, with a few exceptions, not resolvable by static analysis.

Depth

The depth property is applicable for situations where the close proximity of a function may be of interest. For example, considering the following order of arbitrary function calls `update()`, `compute()`, `verify()`, and `exchangeResults()` inside of the main iteration (see Figure 4.2). With a depth rule, where the value is equal to one, the instrumentation would produce a coarse overview of where the time is spent, but not interfere with functions being called during computation or communication. To properly exploit this property some knowledge of the application is necessary.

The depth property takes a rule defining a set of origin functions. Additionally, it takes a number specifying the maximum depth. It returns true for all functions that lay on the call graph within the maximum depth distance in the forward direction of any function in the origin set.

Loops

The loop property is designed to identify functions that contain loop structures in their control flow. This may hint at the computational time spent inside a particular function. The loop property queries whether the function itself contains any loops. The user has to provide parameters specifying a minimum and a maximum value, to restrict the number of loops necessary for the filter to return true. Optionally the property takes a level parameter, which defines how deeply nested a loop has to be, before it is counted. This enables the user to select functions that have a particular number of deeper nested loops.

Call Sites and Callees

The call site property provides the capability to filter functions with respect to the amount of call sites they contain. This can identify functions that are more or less connected than other functions. It is also applicable to exclude leaf nodes in the call tree.

The filter counts all call sites in the function. Similar to other properties, it takes a rule as parameter and thereby enables the user to limit which call sites are counted. For example, the user could specify a rule checking the called function's name for the MPI prefix to identify any functions that invoke MPI functions.

The Callees property works in the opposite direction of the call site filter. It counts the number of functions making calls to the one currently being checked by the filter. A calling function is only counted if it matches the rule given in the specification.

Called in Loops

The intention behind the called in loop property is to identify functions that are called within a certain loop nesting level. Although the iteration count itself is not known, one may assume: the deeper the loop nesting, the more calls to the function may be executed. Whereas the Lines of Code property or the cyclomatic complexity property work on the overhead relative to the time spent in the function itself, this particular property tries to defer overhead from the number of times the function may be executed.

The called in loops property optionally takes one parameter named `level` and checks for the given function whether there exists any call site targeting it which is inside a loop of a nesting level greater or equal to the `level` parameter. In consideration of compiler optimization, functions may not always be found in the binary due to inlining, which removes the overhead of invoking a short function many times, although they are present in the source code.

Library Function

In cases where not only the binary but also the dependencies are loaded, one may need to identify functions that are located within the binary or within any of the dynamic libraries. In those cases Dyninst's module information may produce a file name, making it difficult to separate these functions from user code, or more generally from code where debug information is available and allows Dyninst to provide a file name.

Therefore, the library property identifies those functions that are not within the binary but rather contained within a dynamic library loaded by the executable.

Overlapping

The overlapping property does not relate to performance related properties. The overlapping filter is used to exclude functions, where the Dyninst API reports over-

lap, meaning that according to Dyninst at least two functions share common code in the binary, including exit and enter locations. This filter is necessary to eliminate these functions, as instrumenting their enter or exit locations might confuse the measurement system by miss-triggering enter or exit events.

4.2.2 Filter Specification

All filters are defined inside a single Extensible Markup Language (XML) document, separated from the adapter specification file. This separation is beneficial, because the filters will be modified according to the application developer, while the adapter specification is expected to be left untouched and delivered with the measurement system.

The filter specification file may contain more than one filter definition. Individual filters will be identified by their `name` attribute. For each specified filter one has to define the `start` attribute, to be either *all* or *none*. This selects the initial set of functions the filter would return, if no further criteria were to be specified. A `description` attribute is provided, whose value will be shown when executing the instrumenter and evaluating that particular filter.

Because there may be more than one type of instrumentation defined in the adapter specification (to instrument different points with different code fragments), a filter not only specifies what to instrument, but also how to instrument its result set. This is achieved by creating a list of key value pairs, where the possible keys are: *functions*, *loops*, and *callsites* according to the supported locations, as it is shown in the example Code 2.

```
<filter name="all"
        instrument="functions=epik,loops=epik_loops"
        start="all"> [...] </filter>
```

Code 2: Filter element named *all* where functions will be instrumented with the code named *epik* and loop instrumentation uses *epik_loops*, and the filter starts with set of all functions.

Possible values to select inserted code are the names given inside the adapter specification to identify code elements. If the value remains unspecified, it will be the key itself. In this case the tool developer is supposed to deliver code named *functions*, *loops*, and *callsites*. This mimics a default behavior and thus specifying the `instrumentation` attribute resembles switching on and off a particular type of instrumentation. The gained flexibility is important, because it allows to specify,

in conjunction with a filter criteria, different instrumentations for the same type of locations.

The instrumenter limits the scope of instrumentation by evaluating a filter in the following order: at first the start set is reduced or extended according to the given rules, secondly all resulting functions are instrumented with the specified code fragment. Afterwards, the loop element and the call site element is evaluated for each of these functions, to determine whether any loops or call sites within it have to be instrumented. Thus, the filter restricts the parts of the application where any instrumentation is inserted. If the user decides to instrument everything, he must set the starting set to *all* and then use the True rule, which always returns a match, for the loop and the call site restriction. This forces the instrumentation of all functions, loops, and call sites.

Each `filter` element contains a start set of functions. To modify the set of functions returned, the user specifies `include` and `exclude` elements inside the filter element. Functions that are added to the set (include) or those that will be removed must comply with the set of rules declared as child elements of the `include` and `exclude` element, respectively. The include element iterates over all possible functions present in the binary and adds those that abide by its rule.

To declare the rule according to which the `include` and `exclude` elements adds or removes functions, the user builds a tree of property and pattern elements. The tree will be evaluated for each checked function. To combine multiple pattern or property rules, three rule elements for logical operations are available. These include: `and`, `or`, and `not`. Additionally, `true` and `false` are supported. The `and` and the `or` element both take multiple child rule elements.

To access rules using the patterns, the user needs to add the desired XML elements: `functionnames`, `classnames`, `names`, and `namespaces`. Limiting the module name given by Dyninst is expressed with the `modulenames` element. The identifier related elements take a `match` attribute to specify how to match the patterns against the requested identifier. The list of patterns is placed within each element. To access function properties the `property` element is available, where selecting a particular property is done using the `name` attribute, possible other attributes depend on the requested property. A brief example to show how to instrument all functions in all `.c` or `.cpp` files without `std` namespace members from `.cpp` files is given in Code 3.

To permit the user to restrict where loop instrumentation is inserted and where call sites are instrumented, the user can provide two more elements to limit instrumentation. To limit loop instrumentation the element `loops` is used to specify one additional filter rule. This rule restricts the functions where loop instrumentation is inserted. The user may additionally limit loop instrumentation to outer loops or

```
<filter name="allcfiles" instrument="functions=epik"
  start="none">
    <include>
      <or>
        <modulenames match="suffix">.c</modulenames>
        <and>
          <modulenames match="suffix">.cpp</modulenames>
          <not>
            <namespaces match="prefix">
              std
            </namespaces>
          </not>
        </and>
      </or>
    </include>
</filter>
```

Code 3: Filter to instrument all functions defined in *.c* files or in *.cpp* files without *std* namespace member functions. The filter starts with no functions and includes the ones matching the combination of rules.

until a particular level of loop nesting. The call site instrumentation is restricted by providing one additional `callsites` element, within which the user specifies one rule, which is evaluated for each called function. This enables the restriction of call site instrumentation to those calls made to a particular set of functions. However, with the True rule one can instrument all call sites within selected functions.

4.3 Generic Instrumentation

Targeting not only the Scalasca measurement system, but also to support other measurement APIs, e.g., the TAU system, it is necessary to enable the tool developer to specify how particular locations should be instrumented. The instrumenter must expose most of Dyninst's code generation capabilities to be as generic as possible. Aiming for flexibility and comfortable usage, the code to be executed at particular instrumentation points is specified in a language featuring a subset of the C syntax. This reduces the tool developer's work to specify the code that shall be executed, by using a language most developers are familiar with. This should also provide the flexibility to support future APIs or different event adapters which may require more than a function call. One example could be the upcoming SILC (Skalierbare Infrastruktur zur Leistungsanalyse paralleler Codes) API, where registering a region yields a handler, which needs to be stored and passed to the enter and exit event functions.

Although more points for insertion of instrumentation are available with Dyninst, the possible instrumentation points are limited to the following three types of locations:

Functions can be instrumented at the following locations:

- function entry points
- function exit points

Call sites are instrumentable at the two locations:

- immediately before the call site
- immediately after the call site

And to instrument loops, these four locations are supported:

- right before a loop
- at the entry of a loop iteration
- at the end of a loop iteration
- right after the end a loop

4.3.1 Adapter Specification

The adapter specification will be located in its own XML document, to separate it from the filter specification, as it is more closely related to the measurement system and thus not expected to change, contrary to the filter specification, where user interaction is more likely. The adapter specification in general contains the following three items, although not all of them have to be present in some cases:

- a list of possible library dependencies
- an adapter filter
- one or more code specifications

The List of possible dependencies is necessary to request additional libraries to be dynamically linked to the binary using the Dyninst API. These libraries would in most cases contain the measurement system, but may also provide other necessary functionalities, e.g., needed wrappers (libfmpich.so to wrap Fortran MPI calls).

The adapter filter defines one rule, similar in its construction to the ones used in the filter specification. It is necessary in cases where the measurement system is included in the binary or where the tool providers know of certain criteria for functions that must not be instrumented at all.

The code specification elements are those that define how the snippets look like that will be inserted into the application at the instrumentation points. Derived from the three types of supported locations, one code specification can contain some of the four following code elements: Before, enter, exit, and after. Additionally, one fifth element may contain initialization code. If variables are necessary, the code element also needs a child element listing those. The variables are then available in all of the four fragments.

Because of the ability to instrument different types of locations, functions, loops, and call sites, it may become necessary to define different code fragments for different locations. Therefore, each code elements features the `name` attribute, to achieve this flexibility. Thereby all code definitions are referencable by their defined name. This name was already referred to in the filter specification (see Section 4.2.2).

Code 4 presents a simple instrumentation, which will call `printf()` at function entry and exit locations, if it is used to instrument functions.

```
<?xml version="1.0" encoding="UTF-8"?>
<instrumentation>
  <code name="doprint">
    <enter>printf("entering function");</enter>
    <exit>printf("exiting function");</exit>
  </code>
</instrumentation>
```

Code 4: Adapter specification containing a code element, named *doprint*, with *printf* calls. If a function were instrumented with it, *printf* would be called at its enter and exit location.

Specified initialization code will be placed at the entry site of the `main()` function, allowing for one time code execution in the beginning of the program like it is necessary for the TAU system to define region identifiers (see Section 3.1). However, this currently requires the `main()` function to be present and suitable for the

measurement initialization. Using a filter selecting one particular function where to place initialization code could improve this in the future.

The code itself is defined according to a syntax closely resembling the C syntax, supporting a subset of its features. One essential feature missing though is variable declaration. Because all variables created with Dyninst are static variables, they do not resemble C variables. This means any variable created, receives one unique address and thereby a fixed location in the binary where its value is stored. There is no scope limiting the variables accessibility. Dyninst cannot create variables behaving like the stack variables as one may be used to.

A special notation for variables is introduced, in analogy to Geimer et al. [2009a] (see Section 3.3), to permit the developer to query for information specific to the location of instrumentation. The following variables are available to the developer:

- @functionname@ for the function surrounding the instrumentation point
- @filename@ defining the file where the function is defined
- @linenumber@ line number where the instrumentation point is placed
- @calledfunctionname@ name of called function
- @loopname@ name of the loop instrumented
- @[1...9]@ access to function parameters at function entry points
- @id@ unique identifier in respect to the querying context

The availability of the above specified variables is however limited, because not all are available for all instrumentation points. The `@id@` variable, is defined by the instrumenter and unique for each context of instrumentation.

4.3.2 Example Adapter Specifications

The following two brief sections demonstrate how the adapter specification would look like for the Scalasca measurement system and for the TAU measurement system. It illustrates how to specify the API calls, how to add the new library, and in the TAU case how their current filter could be transformed to an adapter filter.

TAU

The TAU measurement system requires the registration of an ID with the corresponding function identifier (see Section 3.1). Entering and exiting regions or functions is then reported to the TAU system using that specific ID. The `tau_run` binary instrumenter limits the instrumentation according to file types and module names, which is now placed inside the adapter filter. The complete adapter specification is listed in Code 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<instrumentation>
  <dependencies>
    <library name="libTAU.so" path="" />
  </dependencies>

  <adapterfilter>
    <or>
      <modulenames match="prefix">libTAU</modulenames>
      <not>
        <modulenames match="suffix">
         .cpp .c .f .f90 .cc .CC .C  .CPP
        </modulenames>
      </not>
    </or>
  </adapterfilter>

  <code name="TAU">
    <init>trace_register_func(@functionname@,@id@);</init>
    <enter>tauEnter(@id@);</enter>
    <exit>tauExit(@id@);</exit>
  </code>
</instrumentation>
```

Code 5: TAU adapter specification with initialization code to register the id with a region, and the API calls to *tauEnter* and *tauExit*. It also shows the added library and the adapter filter extracted from *tau_run*.

Scalasca

For the Scalasca measurement system the user code instrumentation adapter is used. To enter and exit a region the identifier plus file name and line number are passed on to the measurement system. This adapter specification requires the Scalasca measurement system inside a library. However, with the current instrumentation process used by Scalasca, one would need to instrument the binary first, which adds the measurement system to the application, and then use the adapter

filter to make sure that no Scalasca code is instrumented. Code 6 shows the adapter specification using the Scalasca library.

```
<?xml version="1.0" encoding="UTF-8"?>
<instrumentation>
  <dependencies>
    <library name="libgccscalasca.so" path="" />
  </dependencies>

  <code name="epik">
    <enter>
      EPIK_User_start(@functionname@,@filename@,@linenumber@);
    </enter>
    <exit>
      EPIK_User_end(@functionname@,@filename@,@linenumber@);
    </exit>
  </code>
</instrumentation>
```

Code 6: Adapter specification for Scalasca, calling the *EPIK* functions that otherwise are used for user code instrumentation. It also shows the Scalasca library to be added as a new dependency to the application.

4.4 Implementation

With the overview presented in Section 4.1 in mind, the following sections will present selected parts of the implementation. These include: how properties and patterns are implemented and how flexibility is achieved to easily add new properties and how rules in general are realized to enable their combination. Thereafter, the code generation will be described, followed by how the Scalasca library was build, which is necessary to work with otherwise unmodified binaries.

4.4.1 Program Flow

In short, the instrumenter passes through the following steps: first parse input parameters, then read the given filter specification, parse the requested adapter specification, and make Dyninst analyze the target binary. The binary instrumenter then generates the static call graph for the executable, including if necessary or requested dynamically linked libraries. The previously parsed filters, more specifically used properties, are then initialized. The next step is one loop iterating over all filters requested. Each filter loops over a set of functions and marks those matching the filter criteria to be instrumented as specified. During a last step the instrumen-

tation is inserted by again looping over all functions and checking whether they are flagged for a particular instrumentation.

4.4.2 Execution

Executing the instrumenter in general is done by specifying the following four parameters:

- `--filter`
- `--adapter`
- `--bin`
- `--out`

Where `--filter` and `--adapter` take the file names of the respective specifications. The Parameter `bin` specifies the binary to be instrumented, and `out` the new name of the instrumented binary. Additional parameters allow to select which filters to use. If none is specified all filters are evaluated, something that is useful to determine exactly how many functions are instrumented by each filter. Further options allow to generate a report about the instrumented functions showing more detailed information, or a preview option which disables the binary rewriting, especially useful if the user is only interested in quickly obtaining the size of the result sets.

For a more detailed list of parameters see the Appendix A.

4.4.3 Filter Generation

The filters are generated by parsing the requested filter specification. For each filter this yields a chain of include and exclude rules that will be evaluated according to their order of appearance in the specification. The loop and call site elements are handled separately.

According to the specification, the include and exclude elements contain a tree of XML elements. The XML parser, in this case the Apache Xerces library, yields a Document Object Model (DOM) tree representation of the specification document. The instrumenter evaluates that tree to create its own representation. The new tree is composed of rule objects, whose classes all must inherit from the `IRule` interface.

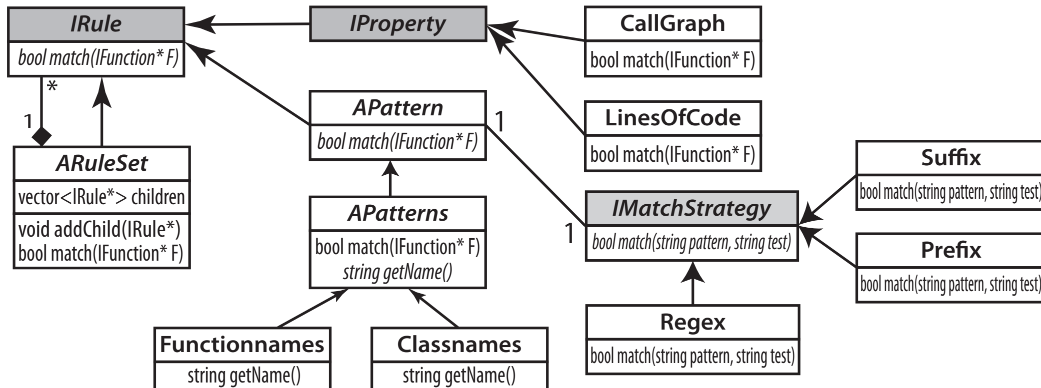

Figure 4.3: Select *IRule* related classes involved in the internal rule representation.

The Rule Interface

The `IRule` interface has to be implemented by all used pattern classes and property classes. It defines the one function `match()` that is used when a particular include or exclude element is evaluated and the instrumenter decides whether or not to add or remove a certain function. Figure 4.3 presents a selected set of classes and interfaces involved in the filter representation. The `IRule` interface is listed in Code 7.

The internal composition of the tree, using logical operators *and*, *or*, and *not* is implemented using the *Composite* pattern [Gamma et al., 1993]. Properties and patterns form leaf nodes in the tree, while and or and not store lists of children. Common to all classes used in the tree is the implementation of the `IRule` interface. Invoking the `match` function on the logical operators, evaluates it for all its children, and returns the value according to its defined operation.

The `ARuleSet` class serves as the base class for the logical operators *and* and *or* and the `match()` function is implemented by the derived classes, where both iterate over their children to evaluate their rules. Parsing the specification, the instrumenter adds a new object to the one object representing the parent logical operator for each encountered child element.

Properties

All classes implementing properties must implement the `IProperty` interface (see Code 8). The interface inherits from `IRule` and therefor enables the use of

```
class IRule {
public:
  virtual ~IRule() {};
  virtual bool match(gim::IFunction* f) = 0;
};

class ARuleSet : public IRule {
protected:
  vector<IRule*> children;
public:
  virtual ~ARuleSet();
  void addRule(IRule* rule);
  unsigned int count();
};
```

Code 7: The *IRule* interface with the *ARuleSet* base class, from which the *and* and *or* operators are derived.

`IProperty` objects within the tree of rules. The interface provides in addition the necessary functions for setup and initialization. Any property object is instantiated through the PropertyFactory.

Property object creation is separated into two steps. The setup itself, executed while the XML element is parsed. During this first step the property's values are set according to the specified values. After proper setup, the PropertyFactory checks whether the object needs the second step, by invoking `needsInit()`. The PropertyFactory will at a later stage go through all property objects that need to be initialized. When executing the initialization all properties get access to the mutatee program, enabling for example access to the application's call graph. Therefore, the call path property is one class which needs initialization to precalculate the set of functions on the requested call path.

Separating setup and initialization of properties makes it possible to parse the adapter and filter specifications before creating the mutatee object, which represents the binary. Its creation, in particular parsing the binary and creating the call graph, takes some time and would otherwise delay quick feedback about errors in the specification files to the user.

Property Management

The specification of properties in the XML document uses the `name` attribute to flexible select different properties. The implementation should reflect this variable property creation. Additionally, supporting new properties should be possible

```
class IProperty : public IRule {
public:
  virtual bool setUp(gix::FilterParser* fp,
                    DOMElement* headElement) = 0;

  virtual bool init(Mutatee* mutatee) = 0;

  virtual bool needsInit() = 0;
};
```

Code 8: The *IProperty* interface showing *setUp*, used during specification parsing, and *init*, which is executed after the target binary is parsed.

without greater efforts or system changes. It was therefore necessary to implement a central component, responsible for both registering properties and instantiating properties of different kinds. This flexibility was realized through the use of the *Abstract Factory* pattern [Gamma et al., 1993]. Registering a new property stores a concrete factory object and associates it with the registered name.

```
class AbstractPropertyFactory {
public:
  virtual IProperty* getProperty() = 0;
};

class PropertyFactory {
private:
  MapNamePropFactory factories;
  IProperty* getPropertyByName(string name);
  PropertyFactory();
public:
  static PropertyFactory& Get();
  IProperty* getProperty(string name);
  void InitProperties(Mutatee* mutatee);
  void registerProperty(string name,
                        AbstractPropertyFactory* af);
};

template <typename P>
  class TemplatePropFactory : public AbstractPropertyFactory {
public:
  IProperty* getProperty() {
    return new P();
  }
};
```

Code 9: The PropertyFactory related classes used to instantiate objects from classes implementing *IProperty* by passing in the property's name

The PropertyFactory is responsible for instantiating objects from classes implementing the `IProperty` interface. It stores objects, whose class implements the AbstractPropertyFactory interface and associates them with a particular name. This name corresponds to the name attribute the user chooses in the filter specification. During parsing of the filter, the PropertyFactory creates the appropriate object through invoking the `getProperty()` method of the stored factory object. A generic factory class is provided to create objects of type IProperty, reducing the necessary effort of writing a class, inheriting from AbstractPropertyFactory, for each new property added (see Code 9).

Pattern Implementation

To allow the user to switch between how a specified pattern is compared against the function's identifiers, different matching strategies were implemented. The simplest one tries to match the whole function identifier, while the most complex allows the user to specify a regular expression. Between these two the user may choose *prefix*, *suffix*, or *find*. Where *find* evaluates to true if any occurrence of the string is found within the identifier and *suffix* or *prefix* check the identifiers suffix or prefix, respectively.

This variability is implemented using the *Strategy* pattern, where the different matching algorithms are hidden in the strategy objects implementing the `IMatchString` interface, shown in Code 10.

```
class IMatchString {
friend class MatchStrategyFactory;
protected:
  IMatchString();
public:
  /**
  * @return true if test matches pattern
  */
  virtual bool match(string pattern,
   string test) const = 0;

  /**
  * @return true if provided pattern is valid
  * ( relevant for regular expression patterns )
  */
  virtual bool validPattern(string pattern) { return true; };
};
```

Code 10: The IMatchString interface, which has to be implemented by the different strategy classes.

The pattern matching may be employed to check a function name, a class name or a namespace identifier as well as the complete identifier. In addition, the file name or module name within which the function is contained may be matched, too. This corresponds to the available string properties exposed by the function object. Using the *Template Method* pattern, the base class `APatterns` implements a match function and a virtual `getName()` function (see Code 11), hiding which name is actually queried in the derived classes, e.g., the `Modulenames` or `Functionnames` classes.

```
class APatterns : public APattern {
protected:
  virtual string getName(gi::mutatee::IFunction* f) = 0;

  /**
  * @param strategy matching strategy
  */
  APatterns(IMatchString* strategy);
public:
  void addPattern(std::string aPattern);
  virtual bool match(gi::mutatee::IFunction* f);
};
```

Code 11: The APatterns base class, with its virtual *getName()* function to be implemented by derived pattern classes.

4.4.4 Code Generation

The tool developers code fragment, which will be inserted at selected instrumentation points, is specified using a subset of the C language syntax, from which it will be parsed into the generic instrumenter's internal representation. After having selected points where the particular code is inserted, the internal representation is translated to the Dyninst Snippet representation. With the help of objects representing the instrumentation point's context, the special variables used to query this context information,e.g., @functionname@ are replaced by Dyninst's constant expression objects.

To integrate the parser into the C++ code, the Boost::Spirit parser, part of the Boost framework classes, was chosen [Boost, 2010]. The Spirit parser itself delivers an intermediate representation, an AST where all tree nodes carry values and IDs referring to the grammar rules applied. All tree node related classes of the instrumenter's internal representation, must implement the `IExpression` interface, where the `getSnippet()` function is responsible for the transformation to the specific DyninstAPI snippet objects (see Code 12).

```
class IExpression {
public:
  virtual BPatch_snippet* getSnippet(IContext* c) = 0;
  virtual ~IExpression();
};
```

Code 12: The *IExpression* interface, which all classes used in the AST implement. *getSnippet* is handed down to possible child objects to generate one large Dyninst *BPatch_snippet*.

During the actual instrumentation, the instrumenter creates a hierarchy of context objects. Each context maps to objects on different levels of the mutatee application. Where the top level context represents the mutatee, followed by one context object for each function and at the lowest level a context object for a particular instrumentation point. Each context stores a reference to its parent context.

Similar to the *Chain of Responsibility* pattern [Gamma et al., 1993], these objects are then responsible for the evaluation of the user requested context variables, like `@functionname@`. If they cannot be resolved by a context object, they are passed on to the parent context. The contexts are also responsible for the creation of variables necessary for a particular instrumentation. Having separate context objects storing user defined variables presents the different scopes of variables, which is necessary for the correct instrumentation.

To insert the code, until now stored in the internal AST representation, the instrumentation point's context is passed to the `getSnippet()` function, and the resulting snippet, actually a root element of the Dyninst AST, is inserted into the mutatee.

As, e.g., Scalasca's user adapter implementation relies on region name identifiers to be of type `const char*` and uses only the address but not its value, the transformation has to make sure that queried identifiers always produce the same constant expression.

4.5 Dynamic Library Workaround

Scalasca in its current release (1.3.1) does not provide the measurement system as a dynamic library. Instrumentation, as mentioned in Section 2.5.2, creates one static binary including the Scalasca measurement system, which is contained in the static Scalasca libraries. Early experiments were conducted by creating a binary including the Scalasca measurement system but without any compiler generated function instrumentation. This defeated part of the purpose of the generic binary instru-

menter, which is being able to instrument binaries without the need to interact with the build environment or where no source is available.

This problem was subsequently solved by executing the following steps: A small dynamic library was created, which in its unmodified form contained functions that call specific MPI functions and make calls to the Scalasca user instrumentation functions. This library was then compiled using the Scalasca instrumenter, resulting in a dynamic library containing the complete measurement system. Using the Fortran MPI symbols, the linker included the Scalasca Fortran wrapper, too, making it possible to use this library with C, C++, and Fortran code.

Having the MPI wrappers for Fortran and C/C++ in the binary made MPI measurements possible, because adding the library to the target binary with help of Dyninst resembles the effect of library preloading, thus enabling library interposition. The target applications dependencies, dynamically linked libraries, are modified by the instrumenter. The Scalasca library is placed second in the list of dependencies, first is the Dyninst run time library. The Scalasca library thereby overloads the MPI symbols defined by the MPI libraries loaded at a later time, thus MPI calls in the application are observed by the measurement system.

With this library it was possible to instrument binary executables, without any modification to the build process followed by a rebuild.

Chapter 5

Evaluation

With the instrumenter available, the filter capabilities and the overhead introduced have to be evaluated. Therefore, different filters with varying threshold values are used to instrument executables of different benchmarks and applications. The aspects of interest are how much overhead is introduced by full instrumentation and how the filters are able to reduce this overhead by excluding functions responsible for it. The filters aim to reduce the instrumentation overhead by excluding functions that are deemed to be less relevant, because their original contribution to the application run time is insignificant and will be dominated by the time spent inside the instrumentation.

The metrics measured are the total run time, in combination with the number of functions instrumented, and the number of different call paths in the recorded call tree. The observed run time allows to determine the overhead introduced, while the number of functions instrumented and the number of call paths provide insight into the information lost due to the reduced instrumentation. The experiments were conducted on JuRoPa.

5.1 JuRoPa

JuRoPa is an Intel Nehalem cluster located at the Jülich Supercomputing Centre, Forschungszentrum Jülich. It consists of 2208 compute nodes, each equipped with two Intel Nehalem quad core CPUs, running at 2.93 GHz having 24 GB of memory per compute node. This provides the cluster with 17664 cores in total. The theoretical peak performance is 207 teraflops. Running the Linpack benchmark yields

a score of 183.5 teraflops. Communication between the nodes is implemented via Quad Data Rate Infiniband network in a non-blocking fat-tree topology.

5.2 Early Observations

The test applications (DROPS and the Cactus benchmarks) were initially compiled with the Intel compilers on JuRoPa, because this is considered the default production environment on the Intel cluster. Running the instrumented versions of the binaries, however, resulted in reproducible segmentation faults and application specific internal errors. After providing small and comprehensible test cases to reproduce the errors and the changes in the application's behavior, the Dyninst developers were able to locate the causes of the problems. One was attributed to missing saves of floating point registers, due to Dyninst's internal optimization not recognizing the use of floating point registers in our sample code. This turned out to be responsible for most of the program internal errors. The second problem being that applications compiled with the Intel compiler and enabled optimization would produce segmentation faults originated from an erroneous stack layout, which was no longer aligned to a 16 byte boundary as it is required by the Intel Application Binary Interface(ABI)[2009].

Although a patch for saving the floating point registers was provided, it did not solve the problem in its entirety, which could only be achieved by forcing Dyninst to save these particular floating point registers independent of the instrumented location and executed code. The Dyninst manual states and measurements done for the initial evaluation show that saving of floating point registers is expensive and introduces additional overhead. In Table 5.1 one can observe that calling the Scalasca API without saving the floating point registers takes 0.0632 seconds compared to the 0.131 seconds consumed when saving those (for 500,000 executions).

5.3 Overhead Measurements

To get an initial impression of how much overhead is introduced when using Dyninst, a small program containing one loop to make a single call to the function `bar` was created and instrumented afterwards. More specific, function entry and exit points of `bar` were instrumented with a function call to `foo()`. For direct comparison with the compiler instrumentation done by Scalasca, the entry and exit points were instrumented with calls to the Scalasca measurement system, too. Note that the called function `foo()` itself is empty, thus its execution time is very short.

The program was compiled using g++ with `-O0`, because optimization had to be deactivated as otherwise the empty function would have been removed and no longer been called. For the Scalasca measurement, all functions were instrumented with `-comp=all` specified. Filtering was done at run time using Scalasca's `scan -f` command. For the binary instrumentation, the binary was compiled with the Scalasca instrumenter, but disabling full compiler instrumentation and instead adding Scalasca's user instrumentation, providing the necessary functions to call through the binary instrumentation.

	Run time [s]	
Number of iterations	500k	10000k
no instrumentation	0.0010	0.0219
compiler instrumentation	0.0316	0.6729
compiler + run-time filter	0.0064	0.1282
Dyninst 6.1		
call to Scalasca	0.0632	1.2631
call `foo()`	0.0338	0.6659
call `foo("")`	0.0338	0.6752
Dyninst with floating point register save		
call to Scalasca	0.1310	2.6114
call `foo()`	0.0961	1.9226
call `foo("")`	0.0960	1.9194

Table 5.1: Showing the run time consumed for a number of iterations to compare the overhead introduced by compiler instrumentation (using run-time filtering) to binary instrumentation, which calls either the Scalasca measurement system or *foo()* at the function's entry and exit site.

Looking at the results presented in Table 5.1, the overhead introduced by Dyninst looks severe in both cases, with and without the forced save of floating point registers. Using Dyninst without saving the floating point registers introduces two times the overhead compared to using the compiler instrumentation. The Scalasca measurement system uses floating point values, e.g., for storing time stamp values, making it necessary to force Dyninst to save the floating point registers before executing the instrumentation. Otherwise current register values would be overwritten. However, this may be changed or optimized in the future. Dyninst already tries to optimize which registers it needs to save. Due to the additional save of floating point registers the overhead introduced by the Dyninst instrumentation increases to four times the overhead introduced by the compiler based instrumentation. The compiler instrumentation results in a run time of 0.0316 seconds compared to the 0.131 seconds.

Using binary instrumentation with a good filter for selective instrumentation may still yield some benefit. The compiler instrumentation by Scalasca does feature a filter, but this filter is applied at run time. Run-time filtering still introduces a small overhead, illustrated in Table 5.1, and interference with enabled compiler optimization may negatively influence the overall performance. One could selectively instrument the code on a per-file basis, or by splitting up code even with compiler instrumentation and avoid the run-time filter overhead, but not without requiring a more serious effort and code changes prior to any experiment.

5.4 Measurements

To evaluate both the filters and the resulting instrumentation overhead, the following example applications were instrumented: at first the DROPS application, written in C++ and developed at the RWTH Aachen was analyzed. Second two different benchmarks, PUGH and Carpet from the set of the Cactus code benchmarks, both written in C++ with parts written in Fortran, were used to analyze code built on frameworks publicly available. The impact of the binary instrumenter on C applications is evaluated by examining the Gadget C code developed at the Max Planck Institute, which during the course of this thesis ran on two thirds of JuRoPa as part of the Millennium XXL simulation.

Due to the described difficulties with Intel compiled binaries all applications, Scalasca, and the dynamic library were built using GNU compilers g++, gcc, and gfortran version *4.3.5*. The Parastation MPI implementation for GCC was used. All measurements were conducted on the Intel cluster JuRoPa and all of the analyzed applications used MPI for their interprocess communication.

5.4.1 Restrictions to Measurement Runs

When evaluating the filters on real applications, some filters resulted in numbers that could not be explained at first and the results deviated from what was to be expected. Using the Lines of Code metric to exclude small functions, the experiment yielded a much higher number of call paths than the experiment produced where all functions were instrumented, as shown in Table 5.2. An increase in the number of call paths is not possible if functions are excluded, because removing a particular function, thereby also its node in the tree, moves all its child nodes up one level to the function's parent.

Further investigation showed that the call tree generated differed from the one where all functions are instrumented. There were many more nodes located at the root level. This was wrong, because the node representing `main` was still present, and all functions are called between entry and exit of `main` and thus have to be children of its node.

	Optimization			
	O0		O2	
Filter used	**Run time**	**Paths**	**Run time**	**Paths**
All functions	2838	110044	133	10043
MPI Path	106	2815	84	3254
Cyclomatic Complexity ≥ 3	145	7628	117	8400
Cyclomatic Complexity ≥ 6	128	2125	92	4689
Cyclomatic Complexity ≥ 9	114	1504	93	2722
Lines of Code ≥ 5	216	22270	115	16261
Lines of Code ≥ 10	164	10918	109	9933
Lines of Code ≥ 15	145	5589	101	7894
no std:: namespace	1477	38305	127	10098
No nested loop calls			101	11576

Table 5.2: DROPS run time and number of call paths, showing more call paths for less functions instrumented due to wrong order of events from overlapping functions.

Because the call tree nodes still resembled the ones found below the `main` node, it was determined to be a local problem, happening in conjunction with certain functions. The double existence of parts of the call tree was responsible for the great increase in the number of different call paths. Using a different instrumentation to output the names of functions entered and exited, it was observed that exit events triggered during the program execution in some cases did not have a matching enter event. Looking at the properties Dyninst provided for these functions, the overlap property and specific exit locations instrumented, it became apparent that more than one function shared a common exit point. When instrumentation code is placed at a shared exit point, the program will trigger this specific event not only when executing the instrumented function but also when executing the one not instrumented.

Regions of a single binary may be shared between functions, but this is uncommon with exceptions for system libraries and very specific Fortran codes. The reason for Dyninst to report overlapping functions originates from the incorrect parsing of the binary, particularly the detection of where a function ends in the binary. Some function exits are defined through their assembly instructions, whereas others are defined by non-returning functions. Non-returning functions are those who do not return to the function where they were called.

When investigating this issue with the Dyninst developers, non-returning functions were found that Dyninst did not recognize. Analyzing C++ code with exceptions showed that calls to functions used in the process of throwing and handling exceptions were not detected as non-returning functions. This makes Dyninst miss the exit point and eventually continue parsing beyond the real boundary of the function. When continuing the parsing processes into the neighboring binary code, the functions were wrongly marked as overlapping. Thus, exit sites not belonging to the first one of the two functions were instrumented. Table 5.3 shows how many functions in the investigated applications overlap (according to Dyninst) with at least one other function.

Application	Optimization	Overlapping	Total
DROPS	O0	198	12272
	O2	410	2754
PUGH	O0	0	2182
	O2	62	2063
Carpet	O0	29	8164
	O2	435	3683

Table 5.3: Number of overlapping functions reported by Dyninst before adding exception related functions as non-returning.

Using `nm` to extract the symbols present in the DROPS and Carpet binaries, the function names of some of the functions involved in throwing exceptions were identified. The Dyninst code was then modified to recognize these particular functions as non returning function calls and thus marking them as exit sites. For the g++ version 4.3.5 used on the test system the following functions where added:

- `_ZSt16_throw_bad_castv`
- `_ZSt17_throw_bad_allocv`
- `_ZSt19_throw_logic_errorPKc`
- `_ZSt20_throw_length_errorPKc`
- `_ZSt20_throw_out_of_rangePKc`
- `_ZSt21_throw_runtime_errorPKc`
- `_cxa_rethrow`
- `_cxa_throw`

The newly created Dyninst library now reported fewer functions that overlapped with others, as it can be seen in Table 5.4, presenting the number of functions which overlap compared to the total number of functions in the executable.

Application	Optimization	Overlapping	Total
DROPS	`O0`	172	12272
	`O2`	210	2754
PUGH	`O0`	0	2182
	`O2`	62	2063
Carpet	`O0`	27	8164
	`O2`	144	3683

Table 5.4: Overlapping functions reported after adding exception related functions as non-returning functions.

Because Dyninst continued to reported functions as overlapping and instrumenting these particular functions still yielded a wrong order of exit events, the functions reported to overlap were not instrumented during the experiments. At this point the adapter filter, a filter defined within the adapter specification and evaluated after any user filter, was set to exclude all functions that reported overlap by using the *hasoverlap* property (see Section 4.2.1).

Measuring the optimized Cactus Carpet benchmark binary, using optimization level `-O2`, produced one internal error during run time. By determining where the error occurred and which functions were involved, the filter for the Cactus Carpet benchmarks was altered to not instrument the single function responsible for the error, being `vect<T,D>::operator&&`.

5.4.2 Used Filters

The experiments focused on comparing filters using the number of instructions, Lines of Code, and cyclomatic complexity to reduce the overhead by excluding small functions to filters only instrumenting the MPI call path, instrumenting only `main` or instrumenting everything but functions inside the `std` namespace. The *main only* instrumentation allows to examine the minimum overhead, as the `main` function is only called once and the MPI wrappers are present for all instrumentations.

Scalasca's main focus is analyzing communication patterns in large scale parallel applications, directing the focus to functions involved in interprocess communication. With respect to relevant instrumentation points this implies that MPI functions should be instrumented. This is achieved by the MPI wrappers, in case of the

binary instrumenter using library interposition. If one would look at an otherwise uninstrumented executable, the resulting call tree would only include nodes for the MPI functions used. Depending on what one is looking for, this may be enough, e.g., the inefficiency patterns would still be detectable. However, the flat call tree would not help the user in locating the context of the executions in question.

To improve the available information about the calling context, it helps to look at the path of function calls leading to the specific MPI communication call. This allows the developer to determine the locations in the program where, e.g., the send and the receive command are issued who together express inefficient late sender behavior. The call tree generated by instrumenting MPI call paths can still be insufficient. It may not explicitly show where the time consuming computations take place, because nodes representing the functions or regions where computation time is consumed are not instrumented and thus do not show up in the call tree. The exclusive time, time spent within a function minus the time spent in its called functions, of nodes on the MPI call path should illustrate where that time is consumed.

The MPI call path filter is used, as the minimalistic solution to instrument all the functions necessary do describe the full calling context for MPI communication calls. The no `std` filter is used to remove many small functions known to be introduced in C++ code through the use of STL containers and other features provided within the `std` namespace.

5.4.3 DROPS

The DROPS application was chosen, because it is written in C++ and is developed at the RWTH Aachen (see Gross et al. [2002] and [Drops, 2010]). The DROPS code relies on two external libraries, DDD [Birken and Bastian, 1994] for data distribution and ParMetis [Karypis et al., 1998] for parallel graph partitioning and distribution. DROPS itself is a Computational Fluid Dynamics (CFD) application used to simulate two-phase flows.

The BrickFlow test case was run for 15 iterations. The overall time reported by DROPS itself was used to compare the run-time overhead introduced with regard to how the filters improve it. The run time of the compiler-instrumented executable was measured to evaluate possible benefits of the binary instrumenter. In addition to the time consumed, the number of call paths was taken from the Scalasca summary report.

In Table 5.5 the huge difference in run time between the instrumented and the uninstrumented binary can be observed. Keeping in mind the results from the overhead measurements, the expected overhead by binary instrumentation is even higher.

Instrumentation	Optimization O0	O2
none	184 s	42 s
compiler	4299 s	4222 s

Table 5.5: DROPS run time without instrumentation compared to the compiler instrumentation version, with disabled and enabled compiler optimization.

To figure out why there was so little improvement from the unoptimized to the optimized compiler-instrumented executable, compared to the improvement gained by the uninstrumented binary, the generic instrumenter was used to count the number of functions that call `__cyg_profile_func_start()`. This is the function called by the GCC instrumentation at function entries. The optimized executable contains 11282 functions calling it, whereas the unoptimized binary contains 11879. The compiler instrumentation seems to be less successful, especially with regards to removing small functions, when instrumentation is enabled (see Table 5.6 for comparison). The existence of that many functions instrumented in both binaries explains why they show similar run times, since both will trigger about the same amount of measurement events.

Filter used	Optimization O0	O2
All	12100	2544
$\geq$ 20 instructions	4405	1767
$\geq$ 40 instructions	1982	1334
$\geq$ 80 instructions	1082	866
Cyclomatic complexity $\geq$ 3	1256	1294
Cyclomatic complexity $\geq$ 6	590	937
Cyclomatic complexity $\geq$ 9	419	711
Lines of Code $\geq$ 5	3819	1804
Lines of Code $\geq$ 10	2189	1453
Lines of Code $\geq$ 15	1354	1244
MPI call path	505	291
No `std` namespace	5329	2131

Table 5.6: Resulting number of functions instrumented for different filters applied to the DROPS binary at the two levels of optimization.

Analyzing the filter results for the DROPS binary for the unoptimized version, the large amount of small functions, in regards to the Lines of Code metric and the low cyclomatic complexity value, stands out. The introduced overhead depends on two factors: One is the number of times the function is executed, while the other is the fraction of time spent inside the instrumentation compared to the time spent inside

the function itself. Having such a large percentage of small functions, where much more time is spent in the measurement system than in the code itself, provides one explanation for the large overhead observed in the experiment.

The cyclomatic complexity filter yields more functions to be instrumented in the optimized binary than in the unoptimized one. This can be attributed to two possible types of optimization. Due to inlining of small functions the cyclomatic complexity of the function making the call increases, because instead of a call it now contains the target function's code. Possible loop unrolling employed during optimization can yield a higher cyclomatic complexity score, too, because of newly introduced control structures for handling remaining iterations.

As more than 5000 functions are members of the `std` namespace, the filter to remove those from instrumentation may result in better performance and a more useful result not cluttered with calls to `std` namespace functions. For the optimized binary this is no longer the case. Due to the enabled optimizations in `O2`, the compiler reduced the number of functions that are members of `std` to 413, which still makes 16 percent of all functions.

	Optimization				
	O0		O2		
Filter used	**Run time**	**Paths**	**Run time**	**Overhead**	**Paths**
All functions	12172 s	≥ 100k	207 s	392%	8759
≥ 20 instructions	1524 s	22824	195 s	364%	6373
≥ 40 instructions	521 s	7550	132 s	214%	4243
≥ 80 instructions	382 s	3269	86 s	104%	2496
Cyclomatic complexity ≥ 3	340 s	4715	161 s	283%	4533
Cyclomatic complexity ≥ 6	253 s	1860	77 s	83%	3083
Cyclomatic complexity ≥ 9	199 s	1318	75 s	78%	1766
Lines of Code ≥ 5	667 s	17405	185 s	340%	6122
Lines of Code ≥ 10	421 s	8603	167 s	297%	4626
Lines of Code ≥ 15	338 s	5576	140 s	233%	3792
`main` function	186 s	22	47 s	12%	22
`MPI` paths	183 s	2744	48 s	14%	1482
No `std` namespace	6515 s	36381	195 s	364%	6478

Table 5.7: Observed DROPS run times for the different filters applied to the unoptimized and optimized binary, showing overhead in percent, and number of call paths.

Comparing the results for the fully instrumented unoptimized binary, using binary instrumentation (see Table 5.7), with the results from the compiler instrumentation, confirms that the binary instrumentation produces more overhead than compiler instrumentation. Removing the `std` namespace yields a 47 percent decrease in

run time, showing that indeed a lot of time is spent inside `std` member function's instrumentation. Although the filter removed 413 of the 2544 functions for the optimized binary, the improvement in run time shows only a 6 percent speedup compared to the fully instrumented binary.

The run time comparison of the *main only* instrumentation and the MPI call path instrumentation illustrates that adding MPI path instrumentation only increases the overhead from 12 to 14 percent but at the same time provides more detailed context information.

The proposed properties to exclude small functions from instrumentation improve the run time performance at the cost of reducing granularity, expressed by fewer observed call paths. The results measured from the unoptimized binary show significant improvements, as the number of instrumented functions is much smaller for all properties compared to the total number of functions. A correlation between the number of functions instrumented and the overhead is possible, because the full instrumentation showed that no single function stood out in the number of executions.

However, only the property cyclomatic complexity, with the chosen values, yields a run time that comes close to the run time of the uninstrumented binary. At the same time it is also more selective than the other properties, instrumenting fewer functions and showing fewer call paths. For the optimized binary all filters reduce the run time, but no filter reaches the run time of the uninstrumented binary. Even for the cyclomatic complexity of at least nine or the number of instructions above 80, the overhead is still 79 percent or 105 percent, respectively.

Inspecting in detail the result of the experiment using the cyclomatic complexity of nine or greater, the number of visits shows that 35 percent of all recorded visits originate from one particular function in the `std` namespace: `std::tr1::_Hashtable<>::_M_insert_bucket`. In direct comparison with the total number of visits for the fully instrumented binary it is responsible for 5.8 percent of all visits. The number of visits is the same for both instrumentations. The second most visited function `DROPS::...::quadBothParts<double>` contributes 14 percent. The contribution to the overall run time is only 5.7 percent and 2.9 percent, respectively.

Taking a look at the `O0` results, the `_M_insert_bucket` function is called the same number of times. However, with the cyclomatic complexity filter requiring a complexity value of three or more the `_M_insert_bucket` function is filtered out. In the optimized case the number of instructions filter shows that it is still instrumented for a minimum value of 80, and as its definition is about 30 lines of code, it is not filtered by the Lines of Code metric either. The cyclomatic complexity of that particular function in the optimized binary is 22.

This provides one example of a function whose run time is dominated by the instrumentation although according to the property values it does not appear small. Multiplying the number of visits with the overhead measured in Section 5.3 is even bigger than its recorded run time (39 seconds to 36.4 seconds).

Combination of Properties

To evaluate whether there are benefits of combining different properties, the DROPS binary was instrumented using a selected set of combinations. For all filters, the `std` and `__gnu_cxx` namespace was excluded, because they are considered less relevant as they are not user implemented. Lines of Code (LoC) was combined with cyclomatic complexity (CC). Additionally, the effect of removing any function that is called in an inner loop or lies on call paths from there on is observed.

Table 5.8 shows the results gained by combining different properties and instrumenting the optimized executable. The overhead is compared to uninstrumented run time and the improvement shown is compared to the full instrumentation overhead.

Filter used	Run time	Paths	Functions	Reduction	Overhead
LoC $\geq$ 5, CC $\geq$ 3	161 s	2977	921	22%	283%
LoC $\geq$ 5, CC $\geq$ 6	72 s	1949	693	65%	71%
LoC $\geq$ 5, CC $\geq$ 9	63 s	1343	562	69%	50%
No loop call sites	133 s	2021	1723	35%	216%
CC $\geq$ 3, LoC 10, -loop	95 s	607	475	54%	126%
CC $\geq$ 6, LoC 10, -loop	47 s	449	391	77%	12%

Table 5.8: Observed DROPS run times for the different filters using a combination of properties.

LoC of 5 and CC of 6 yields a small benefit over using either of them alone. Utilizing loop nesting to exclude functions, where the expectation was that everything nested within loops might get called more often and thus creates more overhead, shows a 35 percent improvement over the full instrumentation, but a total overhead of 216 percent still remains.

A combination of the loop criteria, the CC and the LoC property, yields an improvement of 54 percent, but an overhead of 126 percent remains.Note, that this filter shows worse results than using CC $>$6 alone, with respect to overhead and number of tracked call paths, although the observed call paths may be of different value to the developer.

Table A.1 and Table A.2 (Appendix) show the results for the DROPS binary where I/O operations were enabled. The instrumented binaries, depending on the filters, show a better result with regard to the overhead reduction. This may be attributed to other factors, especially I/O where more time is consumed and thereby reduces the relative amount of overhead. The overhead was contributed mostly by functions residing in the `std` and `__gnu_cxx` namespace plus `DROPS::less1st()`, where the latter one is used to compare the first entry of `std::pair`. The involved `std` functions are all related to maps used to store custom data structures that are compared using `less1st`.

5.4.4 Cactus

The following two benchmarks are based on the Cactus code framework. Written in C++, the central core component called flesh is extended by thorn modules, hence the name Cactus. Thorns implement different extensions from the fields of science and engineering, in addition to providing support for parallel I/O, distribution of workload, and snapshot functionality. Promoted highlights are support for HDF5 parallel file I/O, the PETSc library and adaptive mesh refinement. As a member of Cactus the CactusADM benchmark became part of the SPEC2005 CPU benchmark. More detailed investigations of the Cactus benchmarks can be found in Madiraju [2006] and Allen et al. [2007].

One benchmark, `BSSN_PUGH`, evaluates performance by rendering numerical relativity simulations. With help of the CactusEinstein thorn infrastructure it evolves a vacuum space time, computing the BSSN formulation of the Einstein equations to permit long running numerical simulations. The second benchmark evaluated, `BSSN_Carpet`, is also a numerical relativity code, but uses the Carpet driver for its mesh refinement, instead of a homogeneous grid used in PUGH.

Table 5.9 shows the initial observations of running both the PUGH and the Carpet benchmark, with and without optimization. In addition, the run time with the compiler instrumentation is shown.

	Type of Instrumentation			
	none		compiler	
Benchmark	`O0`	`O2`	`O0`	`O2`
PUGH	1361 s	519 s	1333 s	525 s
Carpet	791 s	290 s	7358 s	6817 s

Table 5.9: Cactus benchmark run times without instrumentation compared to compiler instrumentation with and without optimization active.

PUGH

This benchmark shows almost no observable performance dilation with the compiler instrumentation. Looking at the information gained from the Scalasca profile, it can be noticed that most of the time is spent in a function `adm_bssn_source`. Further investigation showed that this is the Fortran implementation of the BSSN computations which are executed within one large function.

In Table 5.10 the number of functions instrumented by the different filters is presented. The apparent lack of any functions from the `std` namespace is a good explanation for not observing a similar difference in the number of functions when comparing the unoptimized and the optimized binary, because these functions did contribute to most of the reduction of functions in the DROPS binary.

For both versions of the binary, the amount of functions on call paths to `MPI` functions is very small, contributing only with approximately 3 percent of all functions. Looking at the difference between the Lines of Code and the cyclomatic complexity, both properties exclude about the same number of functions from the optimized and unoptimized binary, contrary to the DROPS binary where the optimized binary showed less functions excluded by cyclomatic complexity.

	Optimization	
Filter used	`O0`	`O2`
All functions	2181	2001
Cyclomatic complexity ≥ 3	560	508
Cyclomatic complexity ≥ 6	336	311
Cyclomatic complexity ≥ 9	265	251
Lines of Code ≥ 5	1241	1012
Lines of Code ≥ 10	913	804
Lines of Code ≥ 15	794	701
MPI call path	47	39
No `std` namespace	2181	2001

Table 5.10: Resulting number of functions instrumented for different filters applied to the PUGH benchmark at the two levels of optimization.

After seeing almost no performance perturbation with Scalasca's compiler instrumentation, the binary-instrumented version expresses the same run-time behavior, i.e., showing very little to no overhead (see Table 5.11). The optimized binary run time stays within 528 seconds $\pm$ 1 percent for all filters. Any deviations are more likely to originate from run-to-run variation than instrumentation overhead. Especially the fully instrumented binary shows the shortest run time, which is equal to the run time achieved by the Lines of Code filter, excluding functions with four or less lines.

	Optimization			
	O0		O2	
Filter used	**Run time**	**Paths**	**Run time**	**Paths**
All	1357 s	10320	524 s	3910
Cyclomatic complexity ≥ 3	1368 s	3706	526 s	1553
Cyclomatic complexity ≥ 6	1363 s	831	526 s	505
Cyclomatic complexity ≥ 9	1409 s	388	528 s	326
Lines of Code ≥ 5	1386 s	9743	524 s	3362
Lines of Code ≥ 10	1481 s	5759	531 s	2302
Lines of Code ≥ 15	1365 s	5374	526 s	2083
`main` function	1428 s	12	526 s	12
`MPI` paths	1362 s	35	531 s	44

Table 5.11: Observed Cactus PUGH run time for the different filters used applied to the unoptimized and optimized binary.

The number of call paths recorded differs, showing that the instrumentation itself was inserted and executed. The reduced amount of triggered events can still prove beneficial when tracing is enabled, because the amount of data recorded scales with the number of events traced. Compared to DROPS, the PUGH benchmark has very few call paths leading to MPI functions.

Carpet

The instrumented Carpet binaries ran on JuRoPa using two compute nodes with eight cores each, resulting in 16 MPI processes per measurement. The benchmark was run with the default number of iterations, which is 3072. The problem size is set to increase with the number of processes, resulting in the same workload per process independent of the number of processes used. In this configuration, the benchmark targets the analysis of its weak scaling behavior. For all experiments the benchmark was run ten times and the average total simulation time is used for comparison purposes. The number of call paths is identical in all runs.

Looking at the filter results (see Table 5.12), the first observation is that the unoptimized Carpet binary contains a lot more functions than the optimized one, and both contain more functions than the PUGH benchmark. However, the difference is much lower when comparing both of the optimized executables. Similar to DROPS the unoptimized binary contains a lot of functions within the `std` namespace. The unoptimized binary also contains many functions with very few lines of code.

Enabling optimization shows similar effects to the ones observed with DROPS. Many small functions disappear, especially from the `std` namespace relative to the

Filter used	Optimization O0	O2
All	8137	3539
$\geq$ 20 instructions	3558	1891
$\geq$ 40 instructions	1552	1247
$\geq$ 80 instructions	862	716
Cyclomatic complexity ≥ 3	973	1043
Cyclomatic complexity ≥ 6	518	656
Cyclomatic complexity ≥ 9	401	501
Lines of Code ≥ 5	3045	1786
Lines of Code ≥ 10	1765	1406
Lines of Code ≥ 15	1195	1236
MPI call path	301	226
No `std` namespace	4582	3244

Table 5.12: Resulting number of functions instrumented for different filters applied to the Carpet benchmark at the two levels of optimization.

total number of functions. The increase in number of functions being more complex in respect to their cyclomatic complexity is also present. The MPI call path filter is very restrictive, removing all but 226 of the 3539 functions in case of the optimized binary.

The run time results present a similar picture compared to DROPS when all functions are instrumented. The unoptimized binary suffers from a huge performance penalty, because many small functions are instrumented, whereas the optimized executable, containing significantly less functions that are equally short, does not get slowed down in the same proportion. However, there is still a 61 percent increase in total run time observable (see Table 5.13). The number of visits per function shows that there are two Carpet internal functions responsible for 29.96 percent of all visits and one `std::vector<[..]>at` function is responsible for another 10.35 percent. However, with full instrumentation these three together are responsible for only 5.94 percent of the run time measured when full instrumentation is applied. All three functions are removed by the Lines of Code property for a minimum of five or the cyclomatic complexity property with a minimum of three.

All filters but the one excluding the `std` namespace achieve a significant improvement in the observed performance. For the filter where only the `main` function is instrumented, together with the MPI wrappers registering all MPI calls, the average run time matches the one measured for the uninstrumented version, showing that the overhead introduced through the MPI wrappers is very small. Adding the complete MPI call path to the functions instrumented increases the run time by only 2 percent.

	Optimization				
	O0		O2		
Filter used	**Run time**	**Paths**	**Run time**	**Overhead**	**Paths**
All functions	11160 s	101162	466 s	61%	12698
$\geq$ 20 instructions	2302 s	36553	398 s	37%	7742
$\geq$ 40 instructions	862 s	9272	363 s	25%	4527
$\geq$ 80 instructions	812 s	3103	300 s	3%	2442
Cyclomatic complexity $\geq$ 3	900 s	7626	305 s	5%	4277
Cyclomatic complexity $\geq$ 6	839 s	1808	303 s	4%	2156
Cyclomatic complexity $\geq$ 9	832 s	796	301 s	4%	1444
Lines of Code $\geq$ 5	937 s	24104	314 s	8%	7842
Lines of Code $\geq$ 10	879 s	12261	311 s	7%	5566
Lines of Code $\geq$ 15	835 s	9332	308 s	6%	4681
`main` function	816 s	13	290 s	0%	13
`MPI` paths	867 s	1361	296 s	2%	868
No `std` namespace	3239 s	50658	406 s	40%	10609

Table 5.13: Observed Cactus Carpet run time for different filters applied to the unoptimized and optimized binary.

The 13 call paths in case of the `main` function instrumentation are exactly the nodes created by the Scalasca MPI wrappers plus `main` and therefore may limit the usability of the generated result. Adding the MPI call path nodes generates a path showing more detail by providing the calling context of the different MPI calls. Compared to the total number of paths for full instrumentation, the granularity is still limited.

Comparing the resulting overhead, the amount of functions instrumented and the resulting call paths, the filter instrumenting functions with five or more Lines of Code shows the best result for the optimized Carpet binary. It yields almost twice the number of call paths compared to the cyclomatic complexity, with an overhead increase of only 3 percent.

Effects of Instrumentation on Scaling

To evaluate the effects of the instrumentation on the scaling behavior of the Carpet benchmark, the number of iterations was first set to 500 to reduce the overall run time. The benchmark was executed on 16, 32, 64, and 128 processes using 2,4,8, and 16 compute nodes, respectively. Only the `O2` optimized binary was analyzed. To verify the effects shown for the full instrumentation, the compiler-instrumented version was measured as well.

	Number of Processes			
Filter used	16	32	64	128
No instrumentation	62 s	66 s	73 s	103 s
All	103 s	179 s	416 s	1244 s
Cyclomatic complexity ≥ 3	65 s	80 s	124 s	285 s
Cyclomatic complexity ≥ 6	65 s	80 s	118 s	267 s
Lines of Code ≥ 5	70 s	91 s	167 s	403 s
Lines of Code ≥ 10	68 s	87 s	153 s	344 s
`main` function	62 s	66 s	72 s	105 s
`MPI` paths	62 s	70 s	81 s	123 s
LoC >5 and CC >3 - `std`	63 s	71 s	85 s	138 s

Table 5.14: Carpet benchmark run times for the different number of processes and filters used.

The run time results are shown in Table 5.14. The run times measured for the uninstrumented binaries behave almost in accordance with the Cactus benchmark webpages, mentioning that the Carpet driver, responsible for the mesh refinement, scales well for up to about 100 processes [Cactus, 2010]. Nevertheless, with 128 processes the performance degrades significantly, more precisely with a 41 percent increase in run time when switching from 64 to 128 processes. However, with analyzing communication and scaling behavior being the focus of Scalasca, or in general performance optimization, this may be a valid scenario for an investigated application, though often employed on a much larger scale.

The run time of the fully instrumented binary does not resemble the observed behavior for the uninstrumented binary. It presents a threefold increase in run time when doubling the number of nodes from 64 to 128, and consumes 12 times the run time measured with 16 processes.

The time spent in selected MPI functions, illustrated in Table 5.15, shows that the twofold increase in processes is not reflected in a twofold increase in the time spent within MPI functions. This presents one possible reason why Carpet does not scale well; due to the decreased parallel efficiency.

The excessive growth of the time spent in `MPI_Waitall` and `MPI_Allreduce` with full instrumentation shows that the instrumentation significantly perturbs the experiment. Using the `cube3_score` tool to look at percentages of total time spent in MPI functions vs. non-MPI functions, it calculates that the percentage is the same for both experiments with 64 and 128 processes. The increase of function executions by about six times, meaning three times the executions per node as we go from 64 to 128 processes, could explain why we see three times the total run time. With the MPI fraction staying about the same, the overhead is not only influencing compu-

Number Processes	32	64	128
MPI_Waitall	Total run time[s]		
All	1138	4788	28268
main function	356	821	2562
MPI paths	362	901	2835
MPI_Allreduce	Total run time[s]		
All	81	663	4048
main function	12	67	259
MPI paths	16	79	349

Table 5.15: Total time in spent in selected MPI functions, aggregated over all processes, by the number of processes and filter used for instrumentation.

tations on the single process, but it also directly influences the time spent waiting for other processes, or more generally in communication and synchronization.

Whether or not one filter is better than the other is difficult to argue. The results gained from the experiment where only main is instrumented illustrate that the cost of communication does not scale well with the number of processes. However, both the MPI path filter and the full instrumentation, especially the latter one, provide additional insight as to why the increasing number of processes requires more computational work that ultimately results in the increase of time spent in communication.

5.4.5 Gadget

The Gadget code [Gadget, 2010] is an open source code for N-body/smoothed particle hydrodynamics (SPH) simulations and was developed at the Max Planck Institute (see Springel [2005]). It was part of the Millennium XXL simulation on JuRoPa and ran on two thirds of the cluster. It is written entirely in C and can take advantage of special purpose libraries, such as the Fastest Fourier Transform in the West (FFTW) library [Frigo and Johnson, 2005], to further enhance its run-time performance. Build with the GCC compiler and the FFTW library linked statically, the application was instrumented with both compiler instrumentation and binary instrumentation to evaluate the performance dilation. It is worth mentioning that there were no functions reported by Dyninst to overlap, possibly indicating that overlapping functions are limited to C++ code, as there where no overlapping functions in analyzed Fortran code either.

Because there was no default benchmark supplied, the experiment parameters were modified to make Gadget run for a short period of simulation time, providing a fixed number of iterations. To evaluate the run time and subsequently the performance overhead, the time reported by Scalasca, including closing the experiment, was used. The maximum time consumed by Scalasca to close the experiment was 0.33 s, which is 0.4 percent of the uninstrumented run time. All Gadget experiments were conducted on one compute node utilizing eight processes.

Filter used	Run time	Paths	Functions	Overhead
No instrumentation	64 s			
All functions	104 s	911	402	63%
Cyclomatic complexity ≥ 3	77 s	476	209	20%
Cyclomatic complexity ≥ 6	75 s	400	156	17%
Lines of Code ≥ 5	92 s	261	106	44%
Lines of Code ≥ 10	76 s	231	90	19%
Instructions ≥ 20	78 s	682	326	22%
Instructions ≥ 40	77 s	554	296	20%
`main` function	68 s	18	1	7%
`MPI` paths	69 s	173	83	8%
At least one call site	72 s	699	219	13%
No FFTW prefix	92 s	427	204	44%
No Compare	78 s	907	396	22%
Inst, CC and LoC	74 s	194	66	16%

Table 5.16: Observed Gadget run time for the different filters showing the number of call paths and the number of instrumented functions alongside the introduced overhead in percent.

Looking at the first results (see Table 5.16), the performance penalty for full binary instrumentation is severe, adding up to 63 percent. However, filters using the number of instructions or the cyclomatic complexity achieve significant improvements of more than 65 percent, reducing the remaining overhead to about 20 percent, even with the lowest of the chosen threshold values. The `cube3_score` utility reported three different functions to be responsible for roughly 65 percent of all recorded function calls. This coincides with the improvements gained by the proposed filters, which excluded among others those three functions and removed approximately 65 percent of the observed overhead.

The three functions, including their number of calls and the percentage they contributed to the total number of calls are presented in Code 13.

A filter exploiting one property common to those three functions, that they do not call other functions, performed very well. It excludes all functions that do not call any other functions, thus forming leaf nodes in the call tree. It showed a very

```
USR   254090929   33.60 grav_tree_compare_key
USR   118719711   15.70 domain_compare_key
USR   124651232   16.48 compare_key
```

Code 13: Part of the cube3_score output for the fully instrumented Gadget binary, showing number of visits per function and percentage of total recorded visits for the complete application.

low overhead of 13 percent, but managed to leave 699 of the 911 total call paths visible. Recording the full calling context for MPI function calls by limiting the instrumentation to MPI call paths creates a low 8 percent overhead.

These three functions are executed many times, they have a very low cyclomatic complexity and they consist of very few instructions. However, they were not inlined by the compiler, because they are not called directly in the Gadget source code but rather passed via functions pointers to the GCC quick sort implementation `qsort()`. In this case, the cyclomatic complexity property and the number of instructions property yielded the desired effect of removing a function that consumed very little time per execution and contributed only little to the overall run time. Additionally, using another property, e.g., to check whether it is called in a loop to defer the number of executions, would not be successful, because there is no identifiable call site.

Using the Lines of Code filter, while instrumenting functions with five source lines or more, did not exclude the compare functions and therefore only managed to reduce the overhead by 30 percent. The binary instrumentation for this specific build configuration illustrates the benefit of the binary instrumenter, being to enable instrumentation inside the FFTW library, which was statically linked into the Gadget executable. However, the FFTW library is open source and could be analyzed using compiler instrumentation, too.

Filter used	Run time	Paths	Functions	Overhead
No instrumentation	64			
All functions	104	911	402	63%
Compiler instrumentation	73.6	318	208	15%
Run time filter: compare functions	73	313	208	14%
Run time filter: all functions	69.4	18	208	8%

Table 5.17: Observed Gadget run time for compiler instrumentation, compared to full binary instrumentation.

The compiler-instrumented binary showed less overhead for full instrumentation, compared to the binary-instrumented version, but the compiler instrumented less functions, because it did not instrument the statically-linked FFTW library

(see Table 5.17). Comparing the compiler-instrumented binary with the binary-instrumented one, without the FFTW functions instrumented, still shows that the compiler generated binary performs faster. Binary instrumentation introduces about four times the overhead compared to the compiler instrumentation, which is the same factor that was measured in Section 5.3, where compiler and binary instrumentation were compared using a small example. Additionally, the compiler-instrumented Gadget binary was executed with a run-time filter, excluding the three comparison functions. Even though the filter excludes more than 50 percent of all events, the overhead only decreases by 7 percent.

The evaluation of a heuristic to remove all the functions from instrumentation that might be called frequently by the filter listed in Code 14, which was also used in the DROPS experiments (Chapter 5.4.3), did not show the desired effect of reducing the overhead. Due to the limitations of the call graph, constructed by static analysis, this filter was not able to identify any call sites of the three most called functions.

```
<filter name="notbeyondloops"
    instrument="functions=epik" start="all">
  <exclude>
  <or>
    <property name="path" direction="forward">
      <property name="calledinloop" level="2" />
    </property>
    <property name="calledinloop" level="2" />
  </or>
  </exclude>
</filter>
```

Code 14: Filter to remove functions that are called in loops, or functions called by those functions.

In cases where the call graph is incomplete, changing the rule to include only those matching an explicit callee property may improve the result compared to rules excluding functions, which will be insufficient because of missing links in the call graph.

5.5 Conclusion

When looking back at the evaluation, one has to distinguish between the C++ code examples and the C code example. For the C++ codes the binary instrumenter shows the great benefit of being able to instrument the optimized binary, while the compiler instrumentation interferes with the GCC compiler optimization.

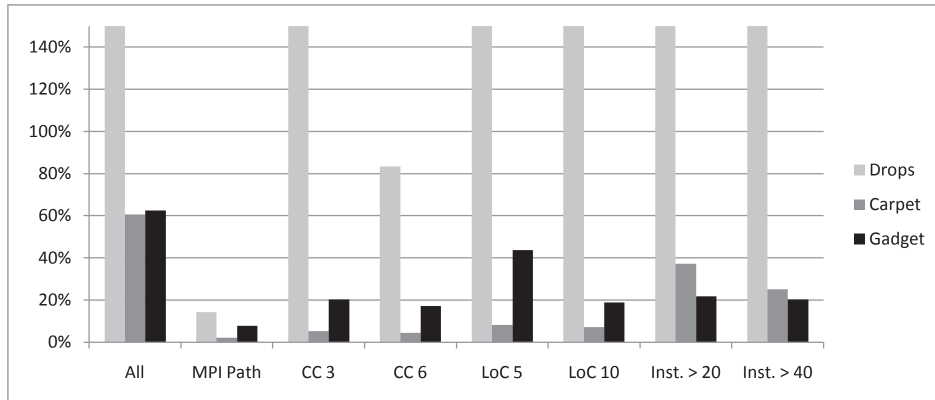

Figure 5.1: Percentage of overhead introduced into the applications for different filters, cut of at 150% to better show small values.

Figure 5.1 summarizes the overhead introduced into three of the applications: Carpet, DROPS, and Gadget. These numbers are taken from the optimized binaries, and cut off above 150 percent. The introduced overhead with full instrumentation is similar for Gadget and Carpet (about 60 percent), whereas DROPS shows much more overhead (392 percent).

Figure 5.2 illustrates the effectiveness of the proposed properties to exclude functions which are mainly responsible for the overhead. Additionally, it presents the benefit of selecting only functions that are relevant to Scalasca's communication analysis, reducing the overhead by more than 80 percent in all applications. The cyclomatic complexity property with a minimum value six proved to be the property that succeeded in all application, removing 70 percent of the overhead. But one has to keep in mind that it also significantly reduces the number of call paths and functions instrumented.

Knowing that most functions that were responsible for the large overhead were functions involved in comparisons, and their codes all include no more than one if statement, the property cyclomatic complexity with a minimum value of three looks like a suitable and generally applicable property. It produces good results by reducing the instrumentation overhead by more than 65 percent (though less effective in DROPS) and on average leaves about half the call paths compared to the full instrumentation. A minimum cyclomatic complexity of three requires a function to contain three control flow paths, thus functions with only one if (such as the compare functions), which results in two paths, are excluded.

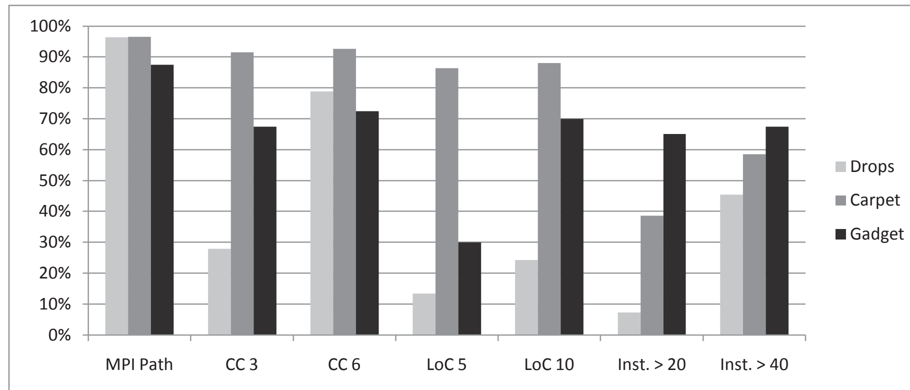

Figure 5.2: Reduction of overhead compared to full instrumentation for different filters, showing the percentage of improvement compared to the total overhead.

The number of instructions property, with a minimum of 40 instructions, also showed good improvements yielding more than 40 percent less overhead, including DROPS. The evaluation of combined properties illustrated some benefit for the DROPS code, thus combining the three properties with their lowest threshold values might create less overhead, but at the same time reduce the granularity.

Any filter should remove the namespaces `std` and `__gnu_cxx`, because of them being probably less interesting and their contribution to the overall overhead with small functions. These functions are prone to introduce overhead, because they are involved in user data structures and are called very often. This improves the usefulness the cyclomatic complexity with a low value for DROPS, where three was too low and six presented much better results, but at the same time also removing much more functions for all observed applications.

One filter that was not evaluated, although the aforementioned key compare functions shared that property, would exclude all functions that do not have a callee function in the call graph. However, this filter would also remove the main BSSN kernel in the PUGH benchmark, a Fortran function, called through a function pointer. The other property they have in common forms another set of functions, those that are leafs of the call graph, because they do not call other functions. This filter proved valuable with the Gadget code, but did not show similar improvement with Carpet and DROPS, this is partly because the small functions responsible for the DROPS overhead are members of the STL and even with inlining still call other STL member functions.

Considering the criticism of removing small functions (see Chapter 3 and Adhianto et al. [2009]), users and tool developers should make use of the rule-based filter to define the proper exceptions. These exceptions could include call paths to synchronization functions to counteract the loss of context information concerning relevant events. Especially since all codes had in common that the number of functions which contributed most of the overhead was small, compared to the total number of uninstrumented small functions.

All these insights make good arguments that it is beneficial to have a rule-based filter which enables users and tool developers to combine properties and patterns. Using the call graph properties, the user has the means to extend the call tree with more detail beyond a certain node, by choosing the forward direction or to select call tree branches, leading to the user's functions of interest. This can prove helpful when extending, e.g., the MPI call path filter to include areas of interest. The heuristics, a combination of properties and patterns to form a filter, that were analyzed so far have not directly selected relevant points, but rather not selected the irrelevant points, explicitly those of functions that do contribute very little to the run time. To decide exactly what is relevant is therefore left to users and tool developers, providing them with means necessary to select those points more easily. For Scalasca the MPI call path provides such an example of a low overhead instrumentation, which sill contains the calling context for relevant communication calls.

Comparing the process of instrumentation, on the one hand compiler based instrumentation, where modifying the build process is necessary, to the generic instrumenter on the other hand, the generic instrumenter can reduce the work necessary before analysis. Combined with the dynamic library containing the Scalasca measurement system it is possible to imagine a system-wide installation, where the user only invokes the instrumenter with the proper parameters and receives a binary ready for analysis. Because the filters evaluated did not contain user code specific criteria, besides the `__gnu_cxx` namespace, this can produce an executable well suited for a first experiment. As the binary instrumenter permits to select different instrumentations, through the filter's `instrument` attribute, a user might even choose between several measurement systems by selecting the appropriate filter or specification file. In case of a broader usage, one may consider filters that not only instrument MPI call paths, but also call paths leading to function calls to other libraries of importance, such as the FFTW libraries or the Intel Math Kernel (MKL) libraries.

Chapter 6

Summary

The increasing difficulty to develop applications that efficiently utilize today's available computing power in HPC clusters motivated the development of various tools for performance analysis. These tools focus on several aspects of performance optimization and employ different measurement techniques. All tools using direct measurement need to instrument the target application at some level. In general their instrumentation can influence the original performance negatively, because it introduces overhead and can interfere with code optimization. Therefore, it is necessary either to refine the scope of the instrumentation or to enable run-time filtering to reduce some of the overhead and the amount of data gathered.

The generic binary instrumenter was introduced in Chapter 4. The instrumenter features an adapter specification and a filter specification. Tool developers use the adapter specification to define code fragments that the instrumenter inserts at selected instrumentation points. The defined code fragments can be inserted at function entry and exit locations, before and after call sites, and loops, to surround both the iteration or the complete loop. Exposing Dyninst's code generation, by parsing a subset of C, ensures good compatibility with available and future measurement system APIs.

The filter specification uses rules to limit the scope of instrumentation. Patterns are used to include or exclude functions based on their identifiers, or to limit instrumentation to specific files. Additionally, rules include properties to extend the possibilities beyond white and black listing of functions and files. These properties aid the user or tool developer in creating rules that are no longer application specific, but generally applicable. Tool developers can exploit this by delivering filters that focus on areas of interest to their analysis. For example Scalasca can provide a filter that limits instrumentation to MPI call paths.

The effects of static binary instrumentation were observed and the effectiveness of several filters was evaluated in Chapter 5. The analysis of different codes, written in different languages, illustrates that there cannot be a general advice whether binary instrumentation proves beneficial in comparison to compiler instrumentation. Drops and the Carpet benchmark suggest that there are significant differences between the binary-instrumented executable and the compiler-instrumented executable, because of the negative influences on the GCC optimizations when instrumentation is enabled. This suggests binary instrumentation can provide better results for C++ codes, if combined with limited filtering. When optimization is not affected and full instrumentation is desired, the compiler instrumentation results in better performance, which is caused by lower overhead per instrumented location. The binary instrumenter shows that avoiding run-time filtering creates better run time performance in cases where the instrumentation is very limited, e.g., MPI call paths. Under such circumstances rule-based filtering requires less user effort to handle and also does not change with the inspected application.

The evaluation also showed that small functions were present in all observed applications, which led to significantly increased run times throughout the experiments. Even with enabled compiler optimization the most problematic small functions did remain present. In all cases these functions placed within the top functions, regarding the number of visits, and therefore contributed the biggest chunk of the instrumentation overhead. Again, common to all codes was that these functions where mostly involved in sorting and accessing larger datasets, some of them providing the comparison function for custom data structures.

The heuristics used to identify small functions, that are deemed to be less relevant because of their very small contribution to overall runtime, did in all cases reduce the overhead, but did not consistently reach overheads lower than 10 percent. However, using the MPI call paths, knowing their relevance to Scalasca's communication analysis, yielded in all cases good results below 10 percent except for Drops, which still presented a 14 percent overhead. The results that were obtained by running the instrumented Carpet benchmark with an increasing numbers of processes underlines the negative impact of instrumentation overhead, but it also illustrates well the importance and benefits of limiting instrumentation, reducing the maximum overhead by over 90 percent.

Chapter 7

Future Work

Adding the instrumenter to the Scalasca tool set will become the next step, which means to incorporating it into the build environment after evaluating the necessary changes with respect to: the libraries used, the dynamic library containing Scalasca's measurement system and modifications to the `scan` utility.

With Scalasca's new tracking ability for iteration dependent behavior, using phase instrumentation, the adapter specification will have to be extended, in combination with an evaluation of how to improve the loop selectivity to enable targeting of specific loops containing the relevant iteration code. Targeting of all outer loops or loops until a certain level in the target function where the iterations are computed may not be precise enough.

As seen in the evaluation, it proves beneficial to instrument only paths leading to MPI calls, limiting the overhead if one is interested the communication behavior. Similar advantages can be expected if one looks at other specific interests. One of them being OpenMP regions. While in case of MPI call paths the target criteria is clear, to locate call sites to MPI functions, a similar easy property might help for OpenMP applications to select particular call paths. Some OpenMP constructs are identifiable, because, e.g., compilers create functions whose identifiers include `omp` to isolate parallel regions. The CATCH tool will most likely provide a good starting point for the analysis.

Current tools, the TAU source code instrumenter and its derivate the generic source code instrumenter, the generic binary instrumenter, and run-time filters all use different specification formats, showing their respective advantages and disadvantages. This raises the question, whether one can unify these formats in a community effort, to allow better interoperability between the different tools.

Appendix A

Appendix

```
Available Options:

Input and Output files:
  --filters arg          filter file to use
  --adapter arg          adapter file to use
  --bin arg              binary to modify
  --out arg              output file
  --use arg              named filter to use
                         (default use all filters in specified file)
  --report arg           file name html report
  --scalascafilter arg   filter file with functions instrumented
  --notscalascafilter arg filter file with functions not instrumented

Flags:
  --help                lists available options
  --preview             no instrumentation done, just show results
                        of filter and produce report if specified
  --test                run parser tests and exit
  --graphTest           run graph tests and exit
  --loglevel arg (=0)   loglevel
```

Code 15: Generic binary instrumenter command line options.

Instrumentation	Optimization $O0$	$O2$
none	667	437
compiler	8095	8049

Table A.1: DROPS run time without instrumentation compared to compiler instrumentation version with and without optimization active, with I/O operations enabled for snapshot and visualization.

	Level of Optimization			
	O2		O2	
Filter used	**Run time**[s]	**Paths**	**Run time**[s]	**Paths**
All functions	16453	106291	710	8995
Cyclomatic complexity >3	854	4780	629	4604
Cyclomatic complexity >6	756	1880	534	2906
Cyclomatic complexity >9	710	1336	550	1810
Lines of Code >5	1262	17697	654	6314
Lines of Code >10	896	8687	647	4700
Lines of Code >15	800	5629	614	3856
`main` function	679	23	512	23
`MPI` paths	678	2812	497	1513
No `std` name space	8874	37398	700	6598

Table A.2: Observed DROPS run time by filter used for the unoptimized and optimized binary, with serialization for snapshots, and Ensight visualization enabled.

Bibliography

L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2009.

G. Allen, E. Caraba, T. Goodale, Y. El Khamra, and E. Schnetter. A scientific application benchmark using the cactus framework. Technical report, Louisiana State University, Center for Computation & Technology, 2007.

Gene Amdahl. Validity of the single processor approach to achieving Large-Scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

Klaus Birken and Peter Bastian. Dynamic distributed data (DDD) in a parallel programming environment - specification and functionality. 1994.

Boost. Spirit 2.3. http://www.boost.org/doc/libs/1_43_0/libs/spirit, 2010.

Cactus. Cactus code. http://www.cactuscode.org, 2010.

B. P.M.M.D Callaghan, J. M.C.J.K Hollingsworth, R. B.I.K.L Karavanic, and K. K.T Newhall. The paradyn parallel performance measurement tools. 1994.

E. D Collins and B. P Miller. A loop-aware search strategy for automated performance analysis. 2005.

L. DeRose and F. Wolf. CATCH—A Call-Graph based automatic tool for capture of hardware performance metrics for MPI and OpenMP applications. *Euro-Par 2002 Parallel Processing*, page 167–176, 2002.

Luiz DeRose, Ted Jr Hoover, and Jeffrey K. Hollingsworth. The dynamic probe class library - an infrastructure for developing instrumentation for performance tools. *Proceedings of the International Parallel and Distributed Processing Symposium*, 2001.

Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

Drops. DROPS homepage | institut für geometrie und praktische mathematik. http://www.igpm.rwth-aachen.de/DROPS, 2010.

Dyninst. Dyninst API. http://www.dyninst.org, 2010.

Martin Fowler. *Refactoring*. Addison Wesley, 8th edition, 2000.

Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Gadget. Gadget 2. http://www.mpa-garching.mpg.de/gadget, 2010.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *ECOOP '93*, pages 406—431, 1993.

GCC. GCC manual code generation options. http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Code-Gen-Options.html, 2010.

M. Geimer, S. Shende, A. Malony, and F. Wolf. A generic and configurable source-code instrumentation component. *Computational Science–ICCS 2009*, page 696–705, 2009a.

Markus Geimer, Felix Wolf, Brian J. N Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009b.

Markus Geimer, Felix Wolf, Brian J. N Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

S. Gross, J. Peters, V. Reichelt, and A. Reusken. The DROPS package for numerical simulations of incompressible flows using parallel adaptive multigrid techniques. Technical Report 211, RWTH Aachen, 2002.

Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf. Scalable detection of MPI-2 remote memory access inefficiency patterns. In *Proc. of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, volume 5759 of *Lecture Notes in Computer Science*, page 31–41, Espoo, Finland, October 2009. Springer.

Oscar Hernandez, Haoqiang Jin, and Barbara Chapman. Compiler support for efficient instrumentation. *NIC Series*, 38:661–668, 2007.

J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. MDL: a language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, page 201. IEEE Computer Society, 1997.

Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of the 7th international conference on Supercomputing*, pages 185–194, Tokyo, Japan, 1993. ACM.

Jeffrey K. Hollingsworth and Barton P. Miller. An adaptive cost system for parallel program instrumentation. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, pages 88–97. Springer-Verlag, 1996.

Intel. Intel 64 and IA-32 architecture optimization reference manual, 2009. Appendix D.

G. Karypis, V. Kumar, and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71—95, 1998.

Monica S Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63—74, 1991.

Chunhua Liao, Oscar Hern, Barbara Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: an optimizing, portable OpenMP compiler. *In: 12th Workshop on Compilers for Parallel Computers*, page 2006, 2006.

K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing, ACM/IEEE 2000 Conference*, page 49–49, 2000.

S. Madiraju. *Performance Profiling with Cactus Benchmarks*. Master thesis, Louisiana State University, 2006.

A. D. Malony, S. Shende, A. Morris, and F. Wolf. Compensation of measurement overhead in parallel performance profiling. *International Journal of High Performance Computing Applications*, 21(2):174–194, 2007.

Allen D Malony and Sameer S Shende. Overhead compensation in performance profiling. *In Proceedings of the 10th International Euro-Par Conference on Parallel Processing (EURO-PAR '04)*, pages 119—132, 2004.

Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, page 407, San Francisco, California, United States, 1976. IEEE Computer Society Press.

B. Mohr, A. D Malony, S. Shende, and F. Wolf. Towards a performance tool interface for OpenMP: an approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, 2001.

MPI Forum. MPI: a Message-Passing interface standard. Technical report, University of Tennessee, 1994.

OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 3.0 edition, 2008.

Matthias Pfeifer. *Automatische Konfiguration der Analyse von Messungen paralleler Programmabläufe*. Diploma thesis, RWTH Aachen, 2008.

Giridhar Ravipati, Andrew Bernat, Nate Rosenblum, and Barton Miller. Toward the deconstruction of dyninst. Technical report, June 2007.

Markus Schordan and Dan Quinlan. A Source-To-Source architecture for User-Defined optimizations. In *Modular Programming Languages*, pages 214–223. 2003.

S. Shende, A. Malony, and A. Morris. Optimization of instrumentation in parallel performance evaluation tools. *Applied Parallel Computing. State of the Art in Scientific Computing*, page 440–449, 2006.

S. S. Shende. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

S. S Shende, A. D Malony, and R Ansell-Bell. Instrumentation and measurement strategies for flexible and portable empirical performance evaluation. In *Proceedings Tools and Techniques for Performance Evaluation Workshop*, volume 3, pages 1150–1156, 2001.

Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364, 2005.

Zoltán Szebenyi, Brian J. N Wylie, and Felix Wolf. SCALASCA parallel performance analyses of SPEC MPI2007 applications. In *Proc. of the 1st SPEC Int'l Performance Evaluation Workshop (SIPEW)*, volume 5119 of *Lecture Notes in Computer Science*, pages 99–123, Darmstadt, Germany, June 2008. Springer.

Zoltán Szebenyi, Brian J. N Wylie, and Felix Wolf. Scalasca parallel performance analyses of PEPC. In *Proc. of the 1st Workshop on Productivity and Performance (PROPER) in conjunction with Euro-Par 2008*, volume 5415 of *Lecture Notes in Computer Science*, pages 305–314. Springer, 2009.

N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. HPC-Toolkit: performance tools for scientific computing. In *Journal of Physics: Conference Series*, volume 125, page 012088, 2008.

Top500. Top500 list - june 2010. http://top500.org/list/2010/06/100, June 2010.

C.C. Williams and J.K. Hollingsworth. Interactive binary instrumentation. *IEE Seminar Digests*, 2004(915):25–28, January 2004.

Felix Wolf, David Böhme, Markus Geimer, Marc-André Hermanns, Bernd Mohr, Zoltán Szebenyi, and Brian J. N Wylie. Performance tuning in the petascale era. In Gernot Münster, Dietrich Wolf, and Manfred Kremer, editors, *Proc. of the NIC Symposium 2010*, volume 3 of *IAS Series*, page 339–346, Jülich, February 2010. John von Neumann-Institut for Computing.

Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, 26(6):30—44, 1991.

Brian J. N Wylie, David Böhme, Bernd Mohr, Zoltán Szebenyi, and Felix Wolf. Performance analysis of Sweep3D on blue Gene/P with the scalasca toolset. In *Proc. 24th Int'l Parallel & Distributed Processing Symposium and Workshops (IPDPS, Atlanta, GA, USA)*. IEEE Computer Society, April 2010.

Brian J. N Wylie, Markus Geimer, Mike Nicolai, and Markus Probst. Performance analysis and tuning of the XNS CFD solver on BlueGene/L. In *Proc. of the 14th European PVM/MPI User's Group Meeting (EuroPVM/MPI)*, volume 4757 of *LNCS*, pages 107–116, Paris, France, September 2007. Springer.

Brian J. N Wylie, Felix Wolf, Bernd Mohr, and Markus Geimer. Integrated runtime measurement summarisation and selective event tracing for scalable parallel execution performance diagnosis. In *Proc. of the 8th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, volume 4699 of *Lecture Notes in Computer Science*, pages 460–469, Ume\aa, Sweden, June 2006. Springer.

Jül-4335

Jülich Supercomputing Centre (JSC)

A Generic Binary Instrumenter and Heuristics to Select Relevant Instrumentation Points

Jan Mußler

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Jül-4335
November 2010
ISSN 0944-2952