From Valid Measurements to Valid Mini-Apps

Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
Genehmigte Dissertation von Jan-Patrick Lehr aus Darmstadt
Tag der Einreichung: 20. Juli 2021, Tag der Prüfung: 03. September 2021

1. Gutachten: Prof. Dr. Christian Bischof (Technische Universität Darmstadt)
2. Gutachten: Prof. Dr. Martin Schulz (Technische Universität München)
3. Gutachten: Prof. Sunita Chandrasekaran, Ph.D. (University of Delaware)
Darmstadt – D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Scientific Computing

From Valid Measurements to Valid Mini-Apps

Accepted doctoral thesis by Jan-Patrick Lehr

1. Review: Prof. Dr. Christian Bischof (Technische Universität Darmstadt)
2. Review: Prof. Dr. Martin Schulz (Technische Universität München)
3. Review: Prof. Sunita Chandrasekaran, Ph.D. (University of Delaware)

Date of submission: 20. Juli 2021
Date of thesis defense: 03. September 2021

Darmstadt – D 17

Erklärungen laut Promotionsordnung

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 20. Juli 2021

Jan-Patrick Lehr

Preface

This preface is to remind me that putting together a dissertation during a pandemic, with a newborn, was not the easiest thing to do.

First and foremost I want to thank my wife for her unbelievable and outstanding support over the years, despite whatever was thrown our way. You are a true hero.

I also want to thank my family and my friends for their support throughout the years and their understanding when I was “busy with a paper, but will visit once it’s done”.

Moreover, I want to thank my colleagues, and my supervisor Christian Bischof for their support. I want to especially thank my colleagues at our institute: Alexander Hück, Christian Iwainsky, Tim Jammer, Yannic Fischler, Michael Burger, Sebastian Kreutzer, Nam Nguyen, Moritz Schwarzmeier, and Iris Schüßler-Janzon. In addition, I want to thank my colleagues from the HPC group at the computing center at TU Darmstadt: Andreas Wolf, Armin Jäger, Benjamin Juhl, and Christian Griebel, and Steffi Vass and Thorsten Reimann from the Hessian Competence Center for High-Performance Computing (HKHLR).

I would also like to thank every student who worked with me over the years, be it as a student research assistant, or doing a theses at our institute under my supervision. Every interaction contributed eventually to this thesis. So thank you Sachin Manawadi, Jonas Focken, Verena Sieburger, Matthäus Kiehn, Leonard Götz, Usman Ahmad, Peter Arzt, Moritz Fischer, Julian Hindelang, Aylin Müller-Cin, Janis Mittelstädt, and Jonas Rickert.

Abstract

In high-performance computing, performance analysis, tuning, and exploration are relevant throughout the life cycle of an application. State-of-the-art tools provide capable measurement infrastructure, but they lack automation of repetitive tasks, such as iterative measurement-overhead reduction, or tool support for challenging and time-consuming tasks, e.g., mini-app creation. In this thesis, we address this situation with (a) a comparative study on overheads introduced by different tools, (b) the tool PIRA for automatic instrumentation refinement, and (c) a tool-supported approach for mini-app extraction.

In particular, we present PIRA for automatic iterative performance measurement refinement. It performs whole-program analysis using both source-code and runtime information to heuristically determine where in the target application measurement hooks should be placed for a low-overhead assessment. At the moment, PIRA offers a runtime heuristic to identify compute-intensive parts, a performance-model heuristic to identify scalability limitations, and a load imbalance detection heuristic. In our experiments, PIRA compared to Score-P's built-in filtering significantly reduces the runtime overhead in 13 out of 15 benchmark cases and typically introduces a slowdown of $< 10\,\%$.

To provide PIRA with the required infrastructure, we develop MetaCG — an extendable lightweight whole-program call-graph library for C/C++. The library offers a compiler-agnostic call-graph (CG) representation, a Clang-based tool to construct a target's CG, and a tool to validate the structure of the MetaCG. In addition to its use in PIRA, we show that whole-program CG analysis reduces the number of allocation to track by the memory tracking sanitizer TypeART by up to a factor of $2{,}350\times$.

Finally, we combine the presented tools and develop a tool-supported approach to (a) identify, and (b) extract relevant application regions into representative mini-apps. Therefore, we present a novel Clang-based source-to-source translator and a type-safe checkpoint-restart (CPR) interface as a common interface to existing MPI-parallel CPR libraries. We evaluate the approach by extracting a mini-app of only 1,100 lines of code from an 8.5 million lines of code application. The mini-app is subsequently analyzed, and maintains the significant characteristics of the original application's behavior. It is then used for tool-supported parallelization, which led to a speed-up of $35\,\%$.

The software presented in this thesis is available at `https://github.com/tudasc`.

Zusammenfassung

Im Hochleistungsrechnen sind Leistungsanalyse, –optimierung und –exploration während des gesamten Lebenszyklus einer Anwendung relevant. Aktuelle Werkzeuge bieten zwar eine leistungsfähige Messinfrastruktur, jedoch fehlt ihnen die Automatisierung sich wiederholender Aufgaben wie die iterative Reduzierung des Messaufwands oder die Werkzeugunterstützung für anspruchsvolle und zeitaufwändige Aufgaben, z.B. die Erstellung von Mini-Apps. In dieser Dissertation adressieren wir die Situation mit (a) einer vergleichenden Studie zu Overheads, die durch verschiedene Werkzeuge erzeugt werden, (b) dem Werkzeug PIRA zur automatischen Instrumentierungsverfeinerung und (c) einem werkzeuggestützten Ansatz zur Mini-App-Extraktion.

Wir präsentieren PIRA zur automatischen iterativen Verfeinerung der Leistungsmessung. Es führt eine Analyse des gesamten Programms durch, wobei sowohl Quelltext- als auch Laufzeitinformationen verwendet werden, um heuristisch zu bestimmen, wo in der Zielanwendung Messhaken für eine Messung mit geringem Overhead platziert werden sollten. Derzeit bietet PIRA eine Laufzeit–, eine Leistungsmodell–, sowie eine Lastungleichgewichtsheuristik. In unseren Experimenten reduziert PIRA den Laufzeit-Overhead in 13 von 15 Testfällen erheblich und führt meist zu einer Verlangsamung von $< 10\,\%$.

Um PIRA die nötige Infrastruktur bereitzustellen, entwickeln wir MetaCG — eine erweiterbare, leichtgewichtige Call-Graph-Bibliothek für *C*/*C*++. Sie bietet eine Compiler-agnostische Call-Graph (CG)-Darstellung, ein Clang-basiertes Werkzeug zum Konstruieren des CG sowie ein Werkzeug zum Validieren der Struktur des MetaCG. Zusätzlich zur Verwendung in PIRA zeigen wir, dass die CG-Analyse des Programms die Anzahl der durch das Werkzeug TypeART zu verfolgenden Allokationen um bis zu $2.350\times$ reduziert.

Abschließend kombinieren wir die Werkzeuge in einem werkzeuggestützten Ansatz, der relevante Anwendungsregionen identifiziert und in repräsentative Mini-Apps extrahiert. Dazu präsentieren wir einen neuen Clang-basierten Source-to-Source-Übersetzer und eine typsichere, vereinheitlichte Checkpoint-Restart (CPR)-Schnittstelle für bestehende Bibliotheken. Zur Evaluation extrahieren wir eine Mini-App mit nur 1.100 Codezeilen aus einer 8,5 Millionen Codezeilen großen Anwendung. Eine Analyse zeigt, dass die Mini-App einen Großteil des ursprünglichen Verhaltens beibehält. Anschließend wenden wir werkzeuggestützte Parallelisierung an und erzielen eine Beschleunigung von $35\,\%$.

Contents

1. Introduction

The increasing complexity of high-performance computing (HPC) systems challenges application developers, performance analysts and system users. Moreover, the increasing heterogeneity of these systems poses additional demands on software w.r.t. performance and portability [89]. Efficient programming of today's HPC systems commonly includes challenges concerning distributed memory parallelization, shared memory parallelization, and accelerator parallelization [65]. These different parallelization paradigms and emerging architectures, such as ARM [91], pose specific demands each w.r.t. performance analysis and optimization as well as correctness validation. As a consequence, performance analysis and performance exploration are important aspects of software development [132]. While the analysis and tuning of a target application seemed to be an art that followed intuition, more structured approaches are used today and software teams and compute centers apply *performance engineering* to maintain and improve execution performance. The increasing complexity of architectures, programs, and applications, highlights the need for automated and elaborate tool support.

Analysis, understanding and exploration of performance characteristics of (parallel) applications has been the target of research for a while and a variety of tools exist. However, in many cases the tools require significant manual work in applying them and in interpreting their results. While the interpretation of the behavior of complex software systems can be notoriously challenging, necessary preparation steps are performed manually albeit being highly repetitive. It is, hence, desirable to increase the degree of automation of typical steps in such workflows. In addition, tools should enable and support the transition from the initial assessment of a target application to detailed experimentation, e.g., via mini-apps, more seamlessly.

Performance Engineering Performance engineering, see Figure 1.1, is the structured approach to the measurement, analysis and improvement of application performance [113]. Since the infrastructure cost associated with a target application performing inefficiently is significant, in particular for frequently-used or long-running codes, devoting time to performance analysis and subsequent tuning is generally beneficial [15]. Specialized analysis tools, such as Scalasca [36], Vampir [101], or Cube [35], have been developed to

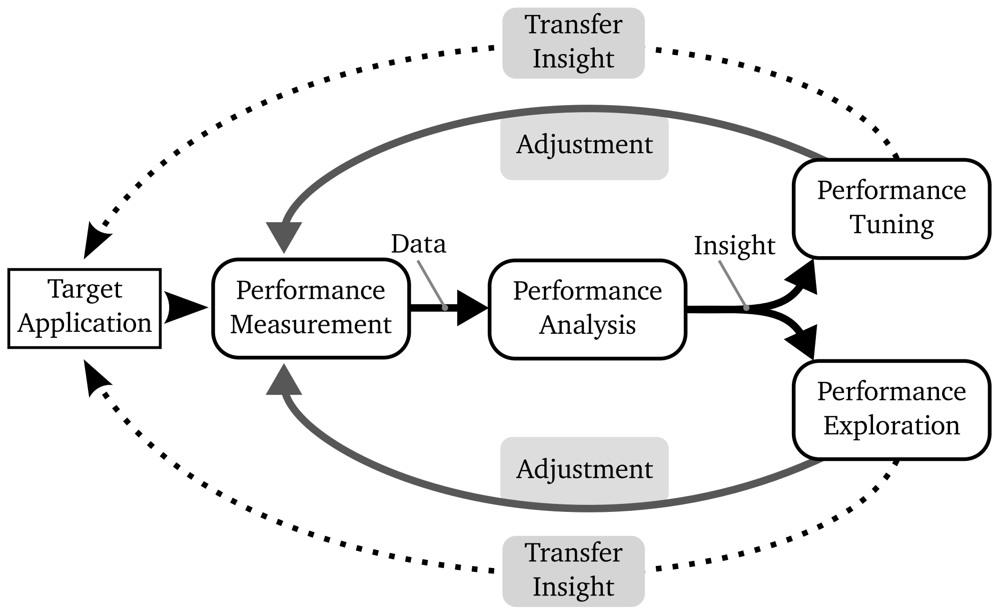


Figure 1.1.: Performance engineering cycles for a *target application*. Performance measurements are conducted to generate data about the behavior of the target application. The data is subsequently analyzed to identify limiting factors and performance bottlenecks. This insight can be used for application tuning, or performance exploration. The insights gained are transferred back to the original target application.

handle large-scale multi-level parallelism. They enable developers and analysts to capture the execution behavior of complex scientific applications that run on heterogeneous machine architectures. Characteristics captured range from basic timing behavior to more elaborate technical details, such as hardware performance counter (HWPC). Some tools even integrate metrics that are particularly HPC-related, such as communication-specific data. Hence, current tools address technical challenges to capture performance data from highly parallel applications.

An important second aspect is the manual effort required to use these tools and the human time associated with it [15]. In particular, the time that application developers and performance analysts spend with manually configuring and adapting a given tool for a specific use-case can be extensive. Hence, it is of high interest to reduce the time spent by persons for highly repetitive work. Consider, as an example, the repetitive task of identifying an application's hot-spots, i.e., regions in the target program that consume considerable amounts of resources and limit the overall performance of the target application. As part of this process, the analyst — conceptually — either (1) starts from overview measurements and refines the focus onto a particular component of an application by repetitive measurement and inspection of the profile, or (2) starts from extensive measurements and iteratively excludes regions that are not of interest. In reality, an analyst is likely to follow a combination of the two approaches to speed up the process.

Tool Support for Performance Measurement To capture the performance of HPC applications at the beginning of performance engineering, HPCToolkit [5], [127] and Score-P [64] are two well established tools. Depending on the specific analysis situation both have their strengths and weaknesses. For advanced analysis, i.e., to investigate a target application's behavior on a detailed machine-level, HWPC are captured for specific regions using specialized libraries, such as PAPI [18]. Such measurement libraries inevitably interact with the hardware, thus potentially skewing the measurement data — particularly for low-level measurement facilities. Since runtime overhead is an often observed influence of the measurement system, it is often used to rate the measurement impact on other metrics, such as HWPC. However, it is unclear whether all HWPC are influenced to the same extent as the runtime is influenced by the measurement system.

Instrumentation, as an intrusive way to capture data, influences the execution of the target program [88], and with more instrumentation, the observable influence typically increases. As an example, consider a *function instrumentation* that adds a function call to every function in the target program to notify a measurement system that it has been entered. Obviously, this will influence the performance of the target application. The effect becomes more visible in applications that are implemented in a programming style that entails many small functions, and particularly pronounced with high call counts [77]. In such cases, it is important to filter irrelevant functions from the instrumentation and restrict it to only relevant regions, e.g., actual runtime hot-spots. Otherwise, the performance data gathered may be misleading or even useless. Currently, however, non-trivial filters are typically created manually, e.g., via function names or loop depth [54], [99]. More recently, PerfTaint [27] uses taint analysis, i.e., an analysis to determine which parts of a program are influenced by input variables, to reduce the amount of instrumentation to only such regions. While these approaches are valuable, they lack support to *adjust* instrumentation automatically to relief the analyst from time-consuming manual effort. Paradyn [44], [93] implemented such adjustments using a measurement feedback system and binary instrumentation [20]. Unfortunately, Paradyn is no longer available to developers and performance engineers.

Tool Support for Performance Analysis Typical questions in HPC consider the identification of application hot-spots or load imbalances for a particular execution configuration. In addition, the scalability of the target application is an important aspect in performance analysis, and performance models have shown useful in uncovering scalability limitations. However, their manual construction, typically based on the application's source code, is time-consuming and tedious. The advent of tools such as Extra-P [23] or SimAnMo [21] allows the construction of performance models using profile data. These *empirical performance models* offer an accessible way to analyze an application's behavior beyond the currently available hardware capabilities for both parallelism and data set size. They are, hence, well-suited for studying how a target program will behave when either parallelism or problem size is increased. For each of these analyses, however, sufficiently accurate low-overhead profiling data is of the essence. Depending on the measurement tool and the target application, the picture of the target's execution hot-spots may be significantly perturbed by measurement overhead, leading to wrong conclusions. This currently requires performance analysts to perform manual filtering and adjustments to the measurement system. Hence, tool-support beyond the current state of the art is needed.

Tool Support for Performance Improvement Once identified, performance-relevant regions are analyzed in detail [55], e.g., a function's ability to efficiently use the processor's cache, and subsequently improved. Such analysis and tuning is often performed on the original application, although it may be considerably time-consuming due to the software's complexity. An auspicious approach is to use mini-apps for analyses and improvement together with a transfer of the insight gained back to the original application. Mini-apps are self-contained and embody essential performance characteristics [82], i.e., they reflect the design and algorithmic choices of full-scale applications. Shalf et al. present a complexity hierarchy [119] with mini-apps being smaller than the original application, but larger compared to skeleton-apps or single kernels. This reduction in complexity enables mini-apps to play an important role in the development process of large-scale software, as they allow, e.g., the application of emerging tools and practices [139] or the exploration of new programming models [90]. Likewise, they are used for demonstrating the capabilities of tool prototypes, such as new instrumentation mechanisms [56]. Unfortunately, their construction usually is prohibitively labor-intensive and requires expertise in many different domains, limiting their application to experts and a small number of projects. Albeit code extraction techniques exist [25], [62], [122], they (1) focus on single kernels, (2) extract code at the compiler intermediate representation (IR) level, or (3) slice code for analysis without being able to generate *C/C*++ code again. Given their advantages, tool-support for the creation of mini-apps is much desired.

Current Challenges Given these scenarios, we identify the following current challenges and impediments for performance analysis tools and tooling support in HPC:

1. Lack of a comparative study for the measurement perturbation introduced by different *state-of-the-art* measurement tools w.r.t. hardware performance counters and runtime as well as their potential correlation.
2. Lack of available *smart* instrumentation tools that use profiling data as feedback mechanism to alleviate performance analysts from repetitive manual tasks. In particular, tools that fully automatically determine relevant regions to instrument via static source-code features or dynamic profile-data analysis.
3. Lack of available tool-supported approaches to automatically identify driving kernels in scientific applications and subsequently create mini-apps in an automated way from these applications.

Contributions The contributions of this thesis concern the fields of instrumentation-based performance measurement, program-analysis representation, compiler-assisted correctness-checking, and compiler-based source-extraction tools. In particular, this thesis makes the following contributions:

Measurement Perturbation Study In a comparative study, we investigate the influence of the state-of-the-art measurement tools HPCToolkit [127] and Score-P [64]. In particular, we investigate the influence of the systems on the validity of values obtained for hardware performance counters and the runtime overhead introduced. The experiments are run across different hardware generations, compilers, and Score-P versions to enable a discussion about trends between different versions of the tools. We also briefly reflect on the tools' ability for automatic filtering and the manual creation of filter lists to reduce measurement perturbation.

MetaCG A whole-program call graph (CG) tool for an extendable program representation that can be annotated with user-defined meta-information for use across different program analysis and transformation tools.

It addresses the need for a lightweight program representation to exchange data between different tools that use a common underlying compiler. The CG is implemented in a reusable *C++* library that can also serialize the graph and attach the user-defined information, i.e., metadata. The MetaCG toolsuite consists of a CG extractor based on the Clang-tooling layer, and we discuss particularities and challenges for CG generation at the compiler's abstract syntax tree (AST) level.

PIRA A tool suite for iterative instrumentation refinement and overhead reduction.

To address the current lack of support for automatic instrumentation refinement, we introduce the PIRA framework. It addresses the need for automatic instrumentation refinement and automated filter list creation. In particular, we present its extendable *Profile Guided Instrumentation Selection* tool which uses static and dynamic heuristics to refine the instrumentation. Its static heuristics take into account source-code features, while its dynamic heuristics take into account profile data. We evaluate PIRA's ability to automatically keep measurement overhead within acceptable limits on a variety of benchmarks, and, finally, investigate PIRA's ability to reduce the influence on hardware performance counters.

Mini-AppEx A tool-supported approach to extract mini-apps from large-scale *C* and *C*++ software packages.

Using automatically determined kernels, the tool extracts *C* and *C*++ source code into an executable mini-app for further analyses. It addresses the bottleneck of mini-app creation being limited to experts, and strives to make the technology available much more broadly. Mini-AppEx is presented as part of a tool-supported approach to extract mini-apps from existing *C* and *C*++ applications using PIRA, MetaCG and a type-safe checkpoint-restart abstraction.

Structure Chapter 2 provides relevant technical background information. Chapter 3 presents the measurement perturbation study. Thereafter, in Chapter 4, we introduce the MetaCG tool, before Chapter 5 explains PIRA in detail. Finally, Chapter 6 presents a tool-supported approach to extract mini-apps from existing applications. We summarize the thesis in Chapter 7 and outline further avenues of research in Chapter 8.

2. Performance and Program Analysis

Since the analysis of a target application is at the heart of many of our contributions, we briefly introduce some general concepts for program analysis in Section 2.1. In particular, we explain the program representations abstract syntax tree (AST), call graph (CG) and control-flow graph (CFG). Our compiler-based tools use the Clang/LLVM toolchain, thus we borrow their representation in examples, unless stated otherwise.

Subsequently, Section 2.2 outlines a high-level approach to performance engineering and the different technological approaches to program measurement, i.e., statistical sampling and instrumentation. We explain the fundamental differences and respective implications that an analyst must take into account. Thereafter, we present in more detail the state of the art in performance measurement and analysis approaches, and whether an approach is available as a tool. In addition, the different approaches are contrasted based on the manual interaction required.

Moving beyond performance measurement, we summarize the current state of the art for automatic source extraction and, more specifically, for mini-app extraction. This technique is especially of interest for performance tuning and exploration.

2.1. Program Representations

Three of the most fundamental representations used within compilers are (1) the abstract syntax tree (AST), a high-level intermediate representation (IR) within compilers, (2) the call graph (CG), commonly used to understand how functions are connected, and, (3) the control-flow graph (CFG), which represents the static control flow in a program. All analyses presented in this thesis use at least one of the three representations.

Abstract Syntax Tree

An important aspect throughout this thesis is the AST, i.e., the compiler-internal representation of the input program. The two most-popular open-source compilers expose their AST to tool developers either through a specialized application programming interface

```
1   int foo(int a, int b) {
2     return a + b;
3   }
4
5   int bar(int k) {
6     if (k < 0) {
7       return foo(-k, k);
8     }
9     return foo(k, k);
10  }
11
12  int main(int argc, char **argv) {
13    int a = 1;
14    int b = 2;
15    foo(a, b);
16    return bar(0);
17  }
```

(a) Example *C++* code.

```
|-FunctionDecl: bar 'int (int)'
| |-ParmVarDecl: k
| `-CompoundStmt
|   |-IfStmt
|   | |-BinaryOperator: '<'
|   | | |-ImplicitCastExpr
|   | | | `-DeclRefExpr: 'k'
|   | | `-IntegerLiteral: 0
|   | `-CompoundStmt
|   |   `-ReturnStmt
|   |    `-CallExpr: 'int'
|   |      |-ImplicitCastExpr
|   |      | `-DeclRefExpr: 'foo'
|   |      |-UnaryOperator: '-'
|   |      |`-ImplicitCastExpr
|   |      |  `-DeclRefExpr: 'k'
|   |      `-ImplicitCastExpr
|   |        `-DeclRefExpr: 'k'
|   `-ReturnStmt
|    `-CallExpr: 'int'
|      |-ImplicitCastExpr
|      | `-DeclRefExpr: 'foo'
|      |-ImplicitCastExpr
|      | `-DeclRefExpr: 'k'
|      `-ImplicitCastExpr
|        `-DeclRefExpr: 'k'
```

(b) Clang AST for function `bar`.

Listing 2.1: Example input program with three functions and a much simplified textual representation of Clang's AST for function `bar`.

(API) or a plugin mechanism. Since our tools rely on the Clang compiler, we present its AST. Other notable compiler infrastructures are the ROSE source-to-source compiler [107] and the GNU compiler collection (GCC) [128]. While, conceptually, ASTs across compilers are similar, their technical details differ.[1]

Listing 2.1 shows a simplified Clang AST for the *C++* function `bar`. While we omitted many details for brevity, the high level of detail is obvious and enables the compiler to perform its analyses. For example, the AST stores that the symbol `k` in line 6 references the variable `k` declared as a function parameter, i.e., the `ParmVarDecl`. Such semantic con-

[1]Depending on the particular GCC version, the main representation of the compiler may not be the AST, but a lower-level representation that is closer to assembler.

nections allow implementing (domain) specific program analysis tools. For example, given the type deduction rules of the programming language, an analysis tool can determine whether a type change, e.g., for algorithmic differentiation (AD, [40]), would result in compilation errors [47].

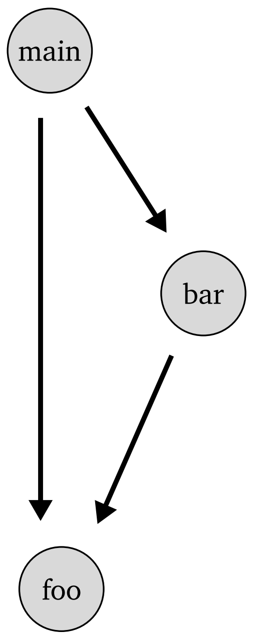
Call Graph

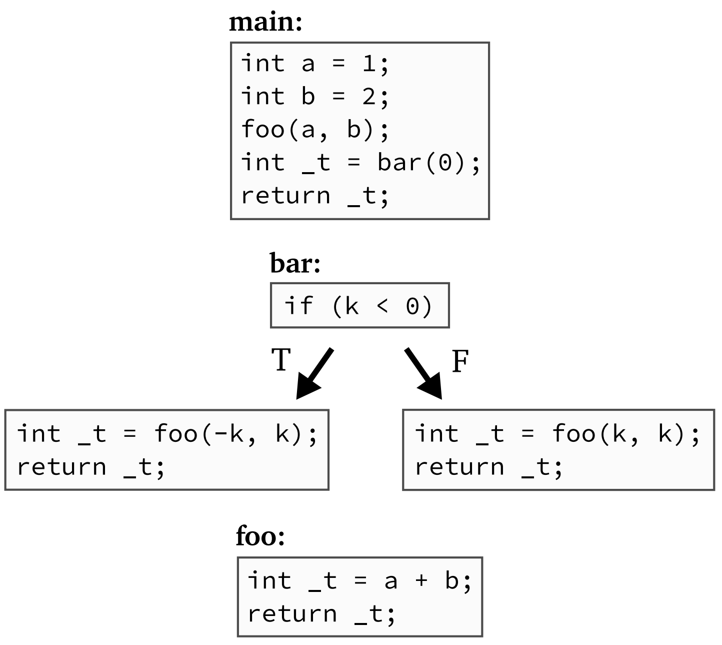
A second important representation for program analysis is the program's CG, which represents the potential call relations between the functions in the program as a directed graph, *cf*. Figure 2.2a. It can be used, e.g., to determine reachability between two program entities and is frequently used for analysis and optimization, such as function devirtualization [129] or function inlining [45]. While its construction for simple programs is straightforward, the correct construction of the CG for *C/C*++ programs in general is challenging due to the language's complexity. Moreover, the modular design of large software systems pose additional technical challenges on CG generation tools. Finally, CGs are typically tool specific and cannot be easily exported for another client to be useful.

CG construction has received attention in the past for various languages. Murphy et al. present an empirical study of the capabilities of different CG generation tools for *C* in [98]. At the time, most of the tools did not rely on compiler infrastructure, but on string matching and similar low-level techniques. This limits their ability to construct the correct CG significantly, as one of the challenges in languages such as *C* and *C*++ is the presence of pointers and pointers to functions. Their particular influence is investigated in [8]. Phasar [115] is a framework for static analysis based on LLVM that also implements different CG construction algorithms. It focuses on security analysis and one of its key components is, thus, the construction of accurate CGs of the target program.

A methodology to compare and rate the quality of CGs is presented by Lhoták in [79]. The method proposed aims to quantify the impact that a missing edge has on validity of the CG generated. Recently, the unsoundness of CG construction algorithms for Java programs was systematically evaluated by Reif et al. [108].

An important aspect, particularly for languages where all methods are virtual, such as Java, is to determine the set of potential call targets as accurately as possible. Tip and Palsberg present a large study on the quality of different CG construction algorithms for Java programs in [129]. The authors compare the algorithms *class hierarchy analysis* (CHA), *rapid type analysis* (RTA), and two different versions of *control-flow analysis* (CFA). The algorithms presented vary in complexity with CHA being the simplest one, and extensive CFA being the most computationally-expensive one.





```
main:
int a = 1;
int b = 2;
foo(a, b);
int _t = bar(0);
return _t;

bar:
if (k < 0)
T:
int _t = foo(-k, k);
return _t;
F:
int _t = foo(k, k);
return _t;

foo:
int _t = a + b;
return _t;
```

(a) The CG shows the potential reachability between the different functions in the target program.

(b) The control-flow graph (CFG) commonly reflects the control flow within each function, but not across functions.

Figure 2.2.: Example graphs for input *C*++ code from Listing 2.1a. The CFG follows LLVM IR's notion that a call is not considered control flow, thus, calling functions `foo` and `bar` does not introduce additional edges.

Control-Flow Graph

A CFG represents the possible control flow through the program and has been studied extensively [6], [106]. It consists of *basic blocks* and directed edges connecting the blocks. All statements within a basic block are executed in sequence from top to bottom, and a block ends with a branch to at least one successor block, see Figure 2.2b for an example. To facilitate the analysis of the flow of data through the CFG, the static single assignment (SSA) form was created [7]. This representation is a typical foundation to data-flow analyses. Note that binaries after compilation and linking are in the form of a CFG, i.e., they consist of basic blocks connected via (un)conditional jumps.

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE MPI_*
INCLUDE bar
INCLUDE foo MANGLED _Z3fooiii
SCOREP_REGION_NAMES_BEGIN
```

Listing 2.2: Example of a Score-P instrumentation filter file. The function `bar` is explicitly included, while all `MPI_*` functions are excluded. The function `foo` is included with an explicitly spelled name mangling, which includes the types into the function name in *C*++.

LLVM IR, the intermediate representation of the LLVM compiler, is based on the CFG in SSA form [69]. The deliberate choice was made to facilitate program analysis and transformation. With its modern API, it attracts a lot of attention and is used in many program analysis research tools, e.g., PARCOACH [112], MACH [59], or CERE [25].

2.2. Performance Analysis

In HPC, the analysis of a target program's performance and its tuning are important tasks. Since performance analysts are not always part of the software development team, the way a target program is approached can greatly vary. Iwainsky et al. [55] propose a structured approach of the process of performance analysis and tuning itself, as well as the interaction between performance analysts and developers. Important is that the workflow distinguishes between overview and focus measurements. For an overview measurement, a lightweight measurement methodology, e.g., statistical sampling, should be used, whereas for a focus measurement a more heavyweight approach can be used, e.g., program instrumentation. The process to shift from the overview to the focus measurement is carried out manually and typically involves manual source-code instrumentation or the creation of filter lists that name functions that should be instrumented or the ones that should be excluded, *cf*. Listing 2.2 for an example. In both cases, however, the user must create such lists and pass them to the measurement system.

More focused measurements are performed to investigate specific hypotheses about the performance of small parts of the target application. Hence, the performance analyst may consider the capturing of hardware performance counter (HWPC) data to investigate, for example, the target's ability to utilize the processor's cache-hierarchy. For such measurements, the influence of the measurement system should remain as small as possible to capture the target's behavior as accurately as possible. To ease the transition from overview measurements to focused measurements an elaborate framework for program

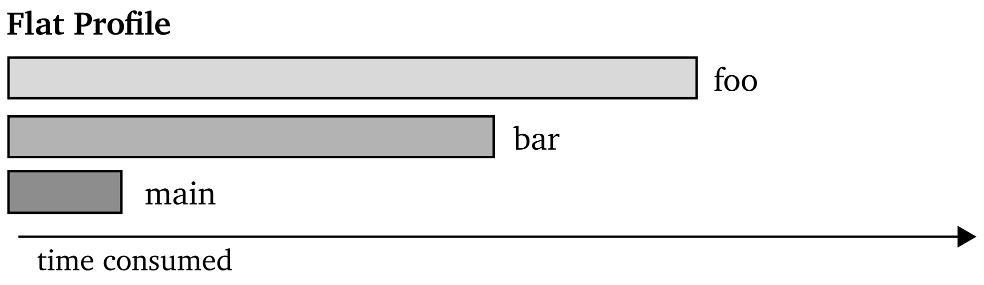


Figure 2.3.: A flat profile shows each function individually with its accumulated runtime irrespective of the respective calling contexts and can be used for an initial overview of which functions are most relevant.

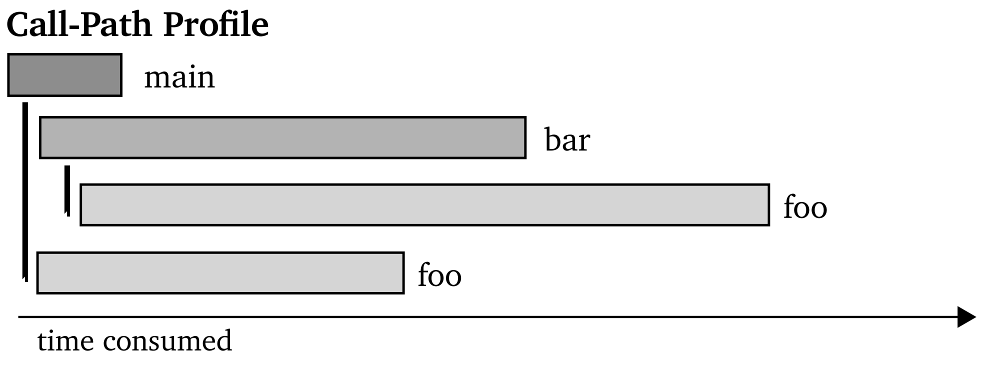


Figure 2.4.: A call-path profile shows each function's accumulated runtime individually per calling context and allows a more detailed analysis compared to the flat profile.

instrumentation is presented by Iwainsky in [54]. However, the process to adjust the instrumentation and move from an overview measurement to more focused measurements is not automated, but left to the user.

The data obtained through either statistical sampling or instrumentation is then visualized or interpreted by subsequent tools. Typical kinds of profiles, on a conceptual level, are (1) a *flat profile*, i.e., a list of functions and their accumulated metric value, *cf*. Figure 2.3, (2) a *call-path profile*, i.e., a tree of all functions and the accumulated metric value within the subtree, *cf*. Figure 2.4, and, (3) a *trace*, i.e., a time sequence of all measurement events that happened during program execution, *cf*. Figure 2.5.

Different tools offer different measurement capabilities and tool-ecosystems. HPCToolkit [127] is a widely-used sampling based profiler, similar to Intel vTune [52], and allows different types of analyses. Score-P [64] serves as a joint measurement layer for a

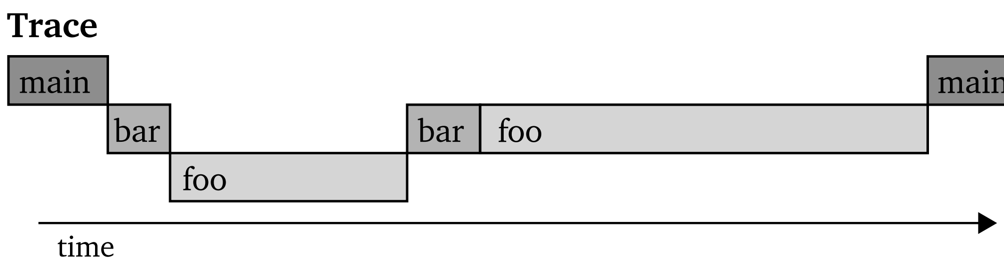


Figure 2.5.: A trace records each individual event and makes the exact relation between the events accessible. This provides the most detail, as individual invocations can be analyzed.

variety of analysis and visualization tools, e.g., Cube [35], Scalasca [36], or Vampir [101]. In addition, the tool Extra-P [23] allows constructing empirically determined performance models from Score-P profiles. This enables a user to analyze a target program's behavior w.r.t. one or multiple varied input parameters, such as the number of processes. Hence, an application's scaling behavior can be extrapolated, e.g., to determine whether a larger number of processors will speed-up the computation, or result in a performance penalty. As such, the Score-P ecosystem offers tools for analysis for different problems, e.g., communication analysis with the Vampir trace analyzer, or hot-spot analysis using Cube GUI, on different granularities given its possibility for compiler and manual instrumentation.

Score-P and HPCToolkit are only two of many tools, and most use their own output format. Given this diversity, Hatchet [13] consolidates the different representations. The Hatchet representation is exposed via an API to perform common metric evaluations, e.g., difference, on the respective call-path profiles. Hence, it allows for programmatic inspection of application profiles from different data sources via a common API.

The next sections elaborate on existing instrumentation tools and approaches, as well as tools that use statistical sampling, and briefly touch on the existing work about measurement perturbation of instrumentation-based measurements.

2.2.1. Instrumentation

Instrumentation adds statements into the target application to obtain information about the state or the progress of it and can be used for performance measurements, correctness analysis, logging, and similar tasks.

Technologically, we distinguish between source instrumentation, compiler instrumentation (frontend and backend) and binary instrumentation, see Figure 2.6. Source instrumentation means that the additional statements are added in the (high-level language)

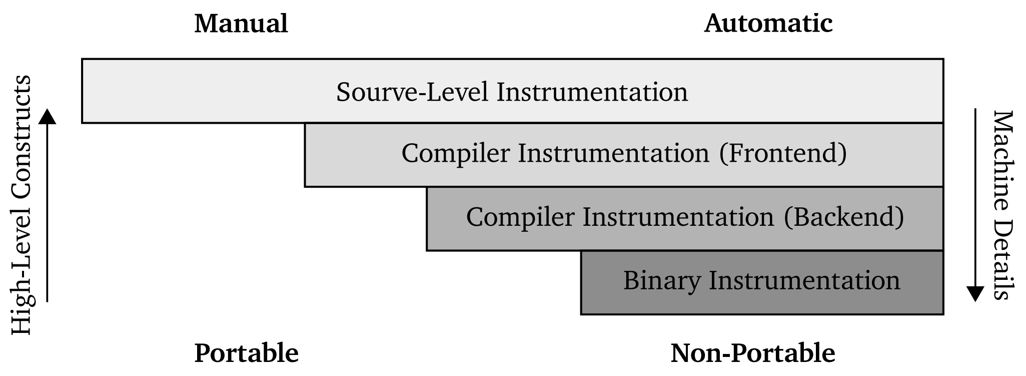

Manual
Automatic
Sourve-Level Instrumentation
Compiler Instrumentation (Frontend)
Compiler Instrumentation (Backend)
Binary Instrumentation
High-Level Constructs
Machine Details
Portable
Non-Portable

Figure 2.6.: Levels at which instrumentation can be added with their typical degree of automation and the respective portability of the instrumentation inserted. Frontend compiler-instrumentation means processing and transformation at the AST level, whereas backend compiler-instrumentation refers to adjusting the final code generation.

```
int foo(int a) {
  __instrumentation_enter(&foo);
  /* Block of original user code without function exits*/
  __instrumentation_exit(&foo);
}
```

Listing 2.3: Example function-level instrumentation at source level. The interface receives the pointer to the function entered as argument for its `XX_enter` and `XX_exit` events.

source-code, e.g., *C/C*++, see Listing 2.3 for an example. The additional statements, thus, interact with the compiler's optimizer during the regular compilation process. We refer to compiler instrumentation at the IR level as compiler-frontend instrumentation, whereas changes to the compiler's code generation is regarded as compiler-backend instrumentation. While the former operates independently of the target platform, the latter may be specific to the target processor, and, thus, less portable. Finally, binary instrumentation means that the instrumentation is added at the binary level, e.g., using x64 assembly.

A second important consideration is available tool support to introduce any desired instrumentation into a target. This is most prominent when approaching new and, previously, unknown large-scale targets, as finding reasonable well-suited spots to place instrumentation may be very time-consuming and hard to find. Hence, the manual introduction of well-suited instrumentation may be prohibitively labor-intensive.

Source Instrumentation

Source instrumentation inserts the additional statements into the source code of the target application. This can be done manually as well as automatically. It is desirable in the case that developers want to insert and maintain specific instrumentation points in their software projects, e.g., enable timing of particular phases in the application, and also allows to freely choose the compiler used. In some software projects such timers are implemented using the programming language built-in time routines, such as `std::chrono` for *C*++, whereas other software projects rely on dedicated libraries for such timers.

To ease the access to low-level HWPC the Performance Application Profiling Interface (PAPI) [18] has been proposed. The library introduces a common interface, implemented as so-called events, to access the many and machine-specific HWPC. It allows capturing values for the counters by registering event sets, i.e., a collection of abstract or machine-specific events. While it offers a rich but low-level API, it requires the user to add the respective instrumentation statements in the source code.

Caliper [16] offers a lightweight instrumentation library to annotate the program source with convenience macros and does not require a specialized compiler to be used. The user inserts calls to Caliper and either captures a start/stop pair for a full function, or defines a named region within two Caliper calls. Before the target application is executed, the user defines which metrics, e.g., wall-clock time, or HWPC using PAPI, the Caliper system should record. This allows for easy exchange of metrics to capture for the already annotated regions in the source code. A more recent library that allows to capture a variety of metrics is timemory [86], which provides a modern *C*++ API.

A similar interface for user annotations is provided by the tool Score-P [64], which we introduce in more detail in Section 2.2.1. Convenience macros are used to insert measurement points and capture full functions or regions defined on a more granular scale, e.g., around a loop. The likwid measurement tool [130] also offers an annotation API that lets a user annotate specific code regions for measurement. likwid is particularly built for measurements of low-level events, similar to PAPI.

The libraries and interfaces mentioned so far require the user to insert the instrumentation. This can be limiting when approaching a new application for analysis. Hence, various approaches have been proposed to insert source-level instrumentation automatically.

The TAU [120] framework enables selective program instrumentation by facilitating the Program Database Toolkit (PDT) [84]. Given PDT's representation of the target application, an analyst can programmatically define insertion points for instrumentation calls. The approach can be regarded as a specialized and domain-specific adaptation of the pointcuts specified in aspect-oriented programming [61]. An aspect-inspired GCC plugin for program instrumentation was proposed by Seyster et al. in [118].

An even more elaborate way to define selective instrumentation is introduced in the InstRO framework [53], [54]. This compiler-based tooling framework allows specifying program parts of interest using their properties. It introduces the concept of a *Construct Set*, which can refer to any behavior-exposing part of a target application. The construct sets can then be freely combined using boolean operations, and — eventually — are passed to an instrumentation back-end. The separation of selection and instrumentation allows to easily introduce new measurement back-ends, and, e.g., combine thread-safe measurement routines for parallel regions with non-thread-safe routines for sequential regions. Hence, fairly advanced instrumentation tools can be constructed using InstRO. A prototype heuristic for InstRO that performs statement aggregation across the application's CG was proposed in [56] and builds the basis for one heuristic in PIRA, see Chapter 5.

Compiler Instrumentation

Most modern compilers offer automatic instrumentation for different interfaces during the compilation process. A well-known interface is the *mcount* interface that is used by the GNU profiler gprof [39]. In each function, the compiler insert a call to the `mcount` function to record the function visit-count, i.e., how often the function has been called. Thus, the gprof profiler uses the instrumentation to capture exact function visit-counts and keep track of the current calling context. The runtime distribution across all functions is, however, obtained using statistical sampling.

Another instrumentation interface available across major compilers[2], is an `enter/exit` instrumentation that we will refer to as *Cyg Instrumentation*. The interface was introduced in GCC and adopted by the other, GCC compatible, compilers. It inserts a call to an *enter* hook at every function start and calls to an *exit* hook at every exit of the function. Client tools can implement and link a runtime library, e.g., to measure the time between the respective enter and exit call to determine the runtime of the function. A user can give a list of function names or paths to GCC to exclude them from the instrumentation, while in Clang, the user has additionally the option to apply this instrumentation before or after the optimizer's inliner passes. Applying the instrumentation before the inliner typically results in a significant performance penalty, which is alleviated to some extent when instrumenting after inlining. Section 3 will discuss this in more detail when different measurement approaches are compared for their impact.

Score-P [64] provides a runtime-library implementation for the Cyg Instrumentation API. However, for GCC, it provides a plugin that implements a different API. With the introduction of the GCC plugin, Score-P also includes compile-time filtering of functions.

[2]GCC, Clang, and Intel

In addition, functions marked **`inline`** or those that are inlined by the compiler are filtered by default. Hence, a user can provide a filter list to exclude or explicitly include certain functions in the measurement. The functionality was also presented for the Clang compiler in [131], but is, unfortunately, not yet part of the Score-P release.

As an approach to lightweight instrumentation, the X-Ray instrumentation [12] in LLVM is noteworthy. The approach inserts *nop*-slides at the start of functions, which, in this case, generate multiple potential entry points into the function. Its first entry point contains instrumentation, whereas the second entry point does not. All calls are then indirected through look-up tables and can be enabled and disabled at runtime. This allows to enable the observation of a particular part of the target application on demand and for a user-chosen amount of time. The insertion of nop-slides is also available in GCC[3] and allows to subsequently insert instrumentation via binary instrumentation tools.

Binary Instrumentation

The last approach for instrumentation presented in this thesis is binary instrumentation, i.e., injecting machine instructions into an already compiled and linked executable. One of the advantages is that the compilation is not affected by additional program statements, hence, the binary-after-compilation is unchanged compared to a production-run binary.

With probe-based binary instrumentation, the target binary is changed at instrumentation time, as the approach uses so-called trampolines to insert the additional instructions. Trampolines can be thought of as (sequences of) assembly instructions that replace an existing instruction and allow branching to the instrumentation inserted. As a result, probe-based binary instrumentation tools have to address several technical challenges, in particular, with complex instruction sets and instructions of varying size, as in `x64`.

Just in Time (JIT)-based approaches perform on-the-fly binary transformation, e.g., assembly to assembly, or first into an IR and then to the target assembly. Hence, such approaches do not modify the existing binary, but duplicate and modify it at runtime. The modified version is then executed. While this addresses some of the complexities of trampoline insertion, depending on the process and its IR, the overhead may be significant.

The probe-based DynInst tool [20] provides a class-based API to inject instrumentation into a target binary. Its API is machine-code independent, meaning that the user does not need to take care of instruction-set differences. Hence, the translation to the target binary assembly language is fully transparent and the technical details are handled by the DynInst tool. This allows to implement portable instrumentation tools. Other tools that implement this approach are DTrace [24], DynamoRIO [19], or PEBIL [70].

[3]see compiler flag `-fpatchable-function-entry` at `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`.

A JIT-based binary instrumentation tool is PIN [85], which allows to insert instrumentation into a running target binary. The user, similar to DynInst, implements a PIN tool, i.e., high-level *C++* code, that is compiled and injected at user-selected points into the binary. When the target application is started under PIN's control, each basic block is JITed and the user-defined functions are invoked or inserted, respectively. The resulting code is stored in a code cache to improve overall execution performance.

Probably one of the most well-known binary instrumentation tools is Valgrind [102] with its tools Call-Grind [136] or memcheck [117]. In contrast to PIN, Valgrind translates a binary into its own machine-independent IR. The IR is manipulated and the user-defined instrumentation is inserted in a JIT-like fashion. Compared to PIN, Valgrind's approach introduces considerably more overhead in many cases.

The COBI [99] tool performs binary instrumentation and relies on the DynInst library. Hence, it allows inserting instrumentation instructions after the compiler has fully optimized the target application. Moreover, the tool introduced a specification language that allows to restrict the instrumentation to particular points in the application. This, similar to TAU or InstRO, introduces a possibility for a high-level description of which parts of an application to instrument.

The Paradyn [93] tool relies on binary instrumentation using DynInst to insert its instrumentation. Most importantly, however, it proposes the W^3 model to analyze the program and determine which parts of it to instrument, following the three model axes *What*, *Where*, and *Why*. Paradyn, thus, provides a feedback-driven approach to automatically identify program parts that limit its performance and change instrumentation towards such regions. The tool, unfortunately, is no longer available as it is incompatible with modern DynInst versions. Its automatic adaption of instrumentation is similar to our approach in PIRA, which we discuss in Chapter 5.

2.2.2. Statistical Sampling

Statistical sampling denotes the interruption of a target application based on specific events, e.g., a timer or a certain HWPC crossing a threshold, and is commonly used for application performance profiling. It is mostly used in conjunction with call-stack unwinding, in which the current calling context is obtained by walking the program stack "upwards", i.e., from the current function towards the `main` function. A popular library to implement call-stack unwinding is libunwind [96], which is also used in the Clang/LLVM compiler to print stack traces in case of program assertions failing.

For a runtime profiler, statistical sampling works as follows: When a sample is taken, it records the current function together with its calling context, e.g., using call-stack unwinding. Over time, the statistical picture of the target application shows which

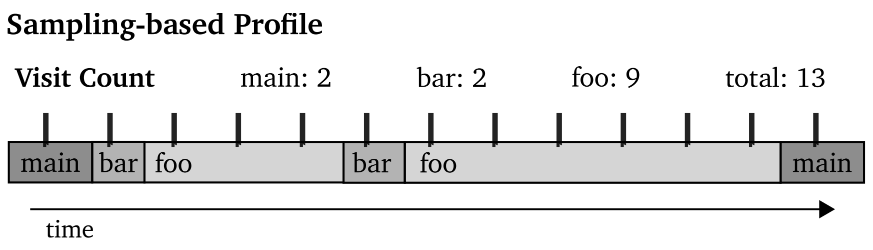


Figure 2.7.: Statistical sampling interrupts the application at a specific frequency (black bars) and counts how many samples are taken for each function to compute the runtime share of each function.

functions are hit frequently, hence, where the application spends most of its time. While call-stack unwinding is employed by the popular tool HPCToolkit [5], [127], other approaches to maintain the calling context, or aid to reconstruct it, have been proposed.

The tool gprof [39] uses a combination of statistical sampling and instrumentation. While the call relationships, i.e., the reconstruction of the calling context, is obtained by a full instrumentation, it captures the runtimes by employing statistical sampling. Its instrumentation surrounds every function call with a call to the `mcount` interface. This has significant runtime overhead for (object-oriented) programs that prefer a programming style with many small functions.

A combination of both techniques to reduce the overall measurement impact is proposed by Morris et al. with their system TAUebs [95]. It uses instrumentation to mark not only functions, but also more abstract regions, e.g., phases of the application. This information can be used by the sampling-based tool to enrich the profiling information it obtains. In addition, the call context obtained through the instrumentation is stored and can help to unwind the call-stack faster by comparing the addresses to the already stored context.

Similarly, Iwainsky et al. [57] use compiler-based CG analysis to place instrumentation calls in functions such that every path through the CG can be uniquely identified by only these probes. In addition, they allow a cut-off depth from which no more instrumentation is added to keep instrumentation overhead small. Hence, an unwinding library only needs to unwind to the last instrumented node, from which it can fully reconstruct the context. This can significantly reduce the runtime overhead introduced by the measurement.

2.2.3. Influence of Measurement

In performance analysis for HPC, even low-level behavior of the target machine is assessed to derive an as-complete-as-possible picture of the target's performance. These low-level details are typically obtained by measuring so-called hardware performance counter

(HWPC). HWPC are special purpose registers built into the processor and present in every recent major model. Their purpose was initially to provide feedback to hardware engineers during the development of a processor architecture [135]. Since then, these counters emerged and evolved as a useful vehicle for performance analysts to obtain and derive characteristics of an application's ability to saturate a particular processor component [51, Appendix B.7]. However, HWPC are prone to measurement perturbation [135], as they count any occurrence of a particular event and do not distinguish between the target application and the measurement system. In addition, small changes to the working environment can have significant impact on experiment results [100]. Hence, HWPC require careful experiment setup for valid results [66] that may even go beyond the generally required rigorous experimental setup for reliable benchmarking results [43].

Specifically for automated instrumentation approaches it is important to note that the additionally inserted instructions can perturb the target application. Moreover, these additional instructions may not only perturb the execution, but also influence a compiler's decision during the optimization stages. However, the influence of the measurement system on the target application's behavior is not limited to instrumentation-based measurements.

The influence of measuring the performance of a target application has received attention in the past. Since in most experiments the measurement infrastructure is running on the same physical device, it naturally competes for resources with the target application [87]. Hence, it influences the behavior of the physical system, which eventually, influences the execution of the target application. It becomes apparent when analyzing timing data that also lists how long, e.g., measurement buffer flush operations, take. For example, runtime overhead can lead to a misleading picture when assessing communication behavior of parallel applications, i.e., affect the temporal alignment in tracing data [88]. The authors also propose on-the-fly mitigation algorithms to account for observed time-skew due to the measurement overhead.

2.3. Kernel and Mini-App Extraction

Once relevant regions in the target application are identified, a developer or analyst wants to investigate the root cause, improve the code base, or perform performance exploration. Due to the vastness of many large-scale scientific software projects, this can be a challenging endeavor. Thus, reducing the amount of code to enable or improve its analysis is of interest not only in HPC. Slicing [138] has been proposed and introduced as the fundamental technique to extract a subset of statements. It has been used and implemented in several research tools, such as change impact analysis [3], software product-line extraction [67], or elaborate type analysis [4]. Typically, a specific statement in the source code is selected and the slice contains (1) all statements impacting the selected one (backward slicing), or (2) all statements impacted by the selected one (forward slicing). For mini-app extraction, however, it is of more interest to extract the compute-heavy parts, i.e., the kernels of an application, and all code required to execute them. In addition, although a substantial body of work exists for slicing, working tools for *C* and *C*++ seem to be missing.

The approach presented by Liao et al. [81] aims at whole-program auto-tuning and consists of a profiling stage, an outlining stage and auto-tuning at runtime. First, the target application is profiled. Second, the results are used to identify relevant parts of the code that limit its performance. Third, the ROSE compiler is used to extract such code parts into their own (parameterized) functions. The parameters are subsequently used by an auto tuner to generate different kernel versions and select the best performing one.

Isolating and extracting single kernels, i.e., functions or loops that dominate the execution of a target application w.r.t. a certain metrics, e.g., runtime, has been tackled in the past. PEMOGEN [14] is an approach for the automatic identification of kernels and a subsequent construction of empirical performance models. The kernel identification is performed on a static graph that contains all loops and function calls, hence, it is referred to as loop call-graph (LCG). Suitable kernels are considered as the call chains within the LCG, and instrumentation is added for the subsequent performance model creation at runtime. This approach does not extract identified kernels, but leaves this task to the user.

The CERE tool [25] is an LLVM-based codelet extractor, which performs profiling of target application, extracts the most significant loops or loop nests, i.e., hot spots, and enables their replay. Compared to the approach mentioned before, it does not generate performance models, but uses thresholding on the profile generated to determine which hot spots to extract. It, subsequently, extracts LLVM IR, hence, it is not primarily suitable for manual inspection, but allows, e.g., compiler-flag tuning. It performs elaborated memory capturing to enable the most similar replay of execution across all replays, i.e., it restores complete memory pages and performs cache warming before taking measurements. Other slicing tools also operate on LLVM IR, e.g., Slabý et al. [123], and focus on the analysis and cannot extract *C*/*C*++ code.

KGEN [62] is an automatic kernel extraction tool for Fortran programs. Compared to the CERE tool, it does not operate on LLVM IR, but extracts high-level language source code for subsequent analysis and processing. It separates the process into the two conceptual stages of (1) identifying the relevant set of source code, and, (2) capturing and restoring the required machine and system state for execution. The identification of source code step requires the user to specify which kernels should be extracted, together with potentially required paths to Fortran modules and macro definitions. Given this input, the tool automatically extracts the kernels and its dependencies, together with code to capture the execution state of the original application. As a final step, KGEN performs validation of the generated kernels through perturbation measurement of captured metrics.

Gong et al. [37] extract single loops and loop nests for subsequent empirical analysis of compiler stability. In their work, they mutate the extracted loops in a source-to-source fashion with semantics-preserving transformations and compile these different versions with the same compiler and flags. Their results show that the compilers are very sensitive to loop structuring w.r.t. the performance of the resulting code.

All aforementioned approaches for mini-app or kernel extraction use either instrumentation to capture the data, or export the full process memory. Moreover, the extraction typically is done at such a low-level that a manual correction or enhancement is impossible, e.g., at the LLVM IR level. A different method to capture application data is checkpoint/restart (CPR). For CPR, several well-tested libraries exist that offer additional benefits, such as fault-tolerant checkpointing schemes [11], or multi-level checkpointing [103]. However, these libraries rely on low-level interfaces, in which a user needs to (correctly) specify data type and size of an allocation. This comes with similar challenges as the Message Passing Interface (MPI) [92] API.

3. The Influence of Measurement

The chapter is based on the following publications.

> **Lehr, Jan-Patrick** and Iwainsky, Christian and Bischof, Christian. 2017. *The Influence of HPCToolkit and Score-p on Hardware Performance Counters* [77]

> **Lehr, Jan-Patrick**. 2016. *Counting Performance: Hardware Performance Counter and Compiler Instrumentation* [71]

For the analysis of a target program, further briefly referred to as *target*, a performance analyst runs measurement experiments to observe the target's behavior. In such measurements, the data is obtained using either statistical sampling or instrumentation. Both techniques have advantages and disadvantages and both influence the behavior of the target. To obtain a valid picture of its execution, it is relevant to the analyst to know of the potential effects of the measurement. The effect most commonly observed is runtime increase in the target's execution, i.e., runtime overhead. It is, however, unclear to which extent low-level events, e.g., hardware performance counters, correlate with such increase.

In this chapter, we investigate the influence of both measurement approaches in more detail, and, more importantly, perform these measurements with state-of-the-art measurement tools and hardware. Compared to previous work, which investigated the influence of manual instrumentation, we focus on the effects of automatic compiler instrumentation. This means that we investigate (1) the static change in the instruction mix of the target binary generated by the compiler, and, (2) the change in measured hardware performance counter events generated at runtime. For the experiments, we use HPCToolkit and Score-P in recent versions and obtain values for runtime and PAPI events on the hardware architectures of Intel Sandy-Bridge and Haswell.

```
void __cyg_profile_func_enter(void *f_addr, void *ret_ip);
void __cyg_profile_func_exit(void *f_addr, void *ret_ip);
```

Listing 3.1: Interface functions of GCC: `f_addr` is the address of the target function, and `ret_ip` is the calling instruction pointer, i.e., the return address within the target binary.

3.1. Influence of Automatic Compiler Instrumentation

Different tools use automatic compiler instrumentation for various use cases, e.g., GProf for function visit counts [39], TypeART for memory allocation tracking [48], or Clang's thread sanitizer for inspection of synchronization errors in multi-threaded programs [116]. Many Score-P versions use the compiler flag `-finstrument-functions`, available with many compilers[1], which inserts calls to the API from Listing 3.1 at the beginning and the end of each function, respectively. This can be problematic in multiple non-obvious ways, as the command-line flag and the additionally inserted instructions (1) change the actually activated optimizations the compiler performs, (2) influence the compiler's code generation, as well as, (3) change the execution characteristics of the target application. The influence is particularly articulated in code bases that make use of many small functions. Hence, applications written in *C*++ with, e.g., many template functions being instantiated to only forward arguments, suffer significantly from measurement overhead.

While the subsequent section assesses the influence of these different approaches to instrumentation w.r.t. instruction selection on the generated binary, we focus on the change in observed, i.e., measured, values of HWPC throughout the chapter. Consider the example function to compute the factorial of a number given in Listing 3.1a. Switching on automatic compiler instrumentation via `-finstrument-functions` results in the code presented in Listing 3.1b actually being compiled. Another observation that we can make from this example is that the instrumentation completely disables the tail-call optimization. As a consequence of this, the subsequent vectorization cannot be performed, which, finally, leaves us with a fairly unoptimized version of the recursive *C*++ implementation. Note that manually adding such instrumentation statements may not interfere as much with the compiler's decisions during optimization.

Influence on Target Binary: Static Instruction Mix

To assess the influence, we developed a tool for static instruction mix comparison between two different binaries. Binary instruction mix comparison has previously been used to

[1]As of this writing, the flag is available in Clang, GCC and Intel compilers.

```cpp
int factorial ( int n) {
  if (n == 0) {
    return 1;
  }

  return n*factorial(n-1);
}
```

(a) Original *C++* implementation.

```cpp
int factorial ( int n) {
  __cyg_profile_func_enter(_,_);
  if (n == 0) {
    __cyg_profile_func_exit(_,_));
    return 1;
  }
  int _t_val = n*factorial(n-1);
  __cyg_profile_func_exit(_,_);
  return _t_val;
}
```

(b) Instrumented *C++* implementation.

Listing 3.2: Recursive implementation to compute the factorial of n and its instrumented counterpart. The arguments to the instruction calls are omitted for brevity.

study the performance differences between different compiler versions and optimization selections [97]. However, we are not interested in the changes of performance for particular regions, but want to quantify the overall impact of automatic compiler instrumentation. Compared to other binary analysis approaches, e.g., for malware identification in driver code [63], the recovery of program structure [26] or software-clone detection [109], our approach is fairly straightforward. The tool[2] relies on the assembly analyzer framework MAQAO [30] to read the target binary. It then assigns each instruction read to its respective instruction category. We introduce the abstract categories *Arith*, *Mem*, *Calls*, *Branches*, *Unconditional Branches*, *Stack Ops* and *Misc*. Note that the Mem category only includes **`mov`** operations with at least one operand going to memory, i.e., register-to-register **`mov`** instructions are not counted. The Misc category includes instructions that should be less frequent in compute-intensive code, such as system calls.

We apply automatic compiler instrumentation to the *C/C++* subset of the SPEC CPU 2006 benchmark suite [42]. Thereafter, we apply the MAQAO-based static instruction mix comparison tool. We restrict the benchmark suite to only the *C/C++* benchmarks, since our toolchain for automatic instrumentation refinement is, due to the compiler employed currently, limited to those two languages. Also, we exclude the XalancBMK XML parser benchmark, due to Score-P being unable to perform measurements for it. The problem is that the benchmark uses *C++* exceptions while parsing, which results in inconsistent region stacks in Score-P and, eventually, Score-P terminates the execution.

In Table 3.1, we can see the change, i.e., increase or decrease, of the different instruction mix categories for the SPEC CPU benchmarks, when compiled with GCC 9.1 and

[2]The tool will be released on github in the near future.

Benchmark	Arith	Mem	Calls	Branches	U-Branches	Stack	Misc
403.gcc				— n/a —			
429.mcf	15	64	3	2	-5	-3	-4
433.milc	43	269	-23	-50	-27	-19	-53
444.namd	-17	195	20	-129	-7	-4	-27
447.dealII				— n/a —			
450.soplex	36	480	-45	-202	-21	20	-57
453.povray	-3191	2620	-538	-4692	-996	-16	-149
456.hmmer	-423	33	-106	-499	-26	10	-30
458.sjeng	-17	124	25	-34	6	-12	-57
462.libquantum	-9	236	21	1	-7	5	-30
464.h264ref	-557	61	131	-826	-140	-103	-181
470.lbm	12	22	5	0	-5	7	-8
473.astar	-48	286	19	-89	-28	-2	-19
482.sphinx3	54	601	46	-69	-19	52	-34

Table 3.1.: Accumulated changes in statically computed instruction mix after automatic compiler instrumentation for the *C/C*++ SPEC CPU 2006 benchmarks when using GCC 9.1. The MAQAO tool exited with a segmentation fault for 403.gcc and 447.dealII.

Score-P 6. We compute the inclusive instruction mix for the target application based on the static CG for both versions — the vanilla and the automatically instrumented version. Subsequently, we compare the instruction mix for the `main` function. The additional calls to the instrumentation functions are excluded, to focus on the changes compared to the original binary. For both 403.gcc and 447.dealII the MAQAO based tool failed to compute the instruction mix, and exited with a segmentation fault.

We find that for all binaries a statically determinable difference exists. The degree of influence varies from almost no influence, e.g., 429.mcf, to heavily influenced, e.g., 453.povray. The Mem category increases for all targets, while the number of conditional and unconditional branches decreases for almost all targets. Other metrics show a mixed picture of influence from the automatic compiler instrumentation. Since these different influences exist across the benchmark suite, it is ill-advised to assume a certain influence the automatic compiler instrumentation may have on the target binary.

To investigate how large the impact of the instrumentation on the target's behavior is, however, the instrumented binary is executed to generate measurement data that is subsequently analyzed. Hence, it is of significant interest to investigate the influence that the instrumentation inserted has on the measurement. In the next section, we compare the influence in observed HWPC values for different measurement techniques.

3.2. Influence of Score-P and HPCToolkit on Hardware Performance Counter

The different measurement approaches, instrumentation and sampling, have different strengths and weaknesses, and which one to use depends on the task at hand. Thus, it is of interest to know the degree of perturbation that these techniques introduce while generating data. We developed a thin library layer[3] on top of PAPI [18] that allows to measure HWPC through a *C++* API. Moreover, it implements constructor and destructors at the library level, meaning that it can be *preloaded* to measure the PAPI values over the target's full execution time.

In our experiments, we focus on the instrumentation-based measurements using Score-P. Therefore, we consider the target application (1) without any instrumentation, i.e., the *vanilla* version, (2) with instrumentation inserted but without actual Score-P measurement system, called *finstr*, (3) with *filtered* instrumentation inserted and Score-P measurement system attached — referred to as *scorep*, and, (4) with instrumentation inserted and Score-P measurement system attached, referred to as *scorep-no-filter*, as the different flavors of the target application. However, for the purpose of comparisons, we also capture the influence of the sampling-based measurement tool HPCToolkit [127]. These sampling-based measurements are performed with two different sampling frequencies, i.e., the number of measurement events per second.

Measurement Setup For the measurements, we use Score-P version 3.0 that was originally used for the experiments, but also the more recent version 6.0. Score-P introduced the possibility for compile-time filtering of inlined functions using a GCC plugin, and Clang offers a post-inline function instrumentation. Hence, our measurements for the newer versions of Score-P include both filtered and non-filtered measurements. Score-P version 3.0 is used with the older GCC (version 4.9), while the newer Score-P version is used with GCC in version 9.1 or Clang 10.0. The particular version of Clang is used, since the automatic instrumentation framework PIRA, explained in detail in Chapter 5, uses this version as the backend compiler. Finally, due to the evolving hardware platforms, we obtain the HWPC values on different processor architectures. The measurements are

[3]Available at `https://github.com/jplehr/papi-wrap` in version 0.1.0.

Name	GCC		Clang	Instrumentation		Record
	4.9	9.1	10	Full	Filtered	
<BM>.gcc4_9_4.vanilla	×					
<BM>.gcc4_9_4.finstr	×			×		
<BM>.gcc4_9_4.scorep	×			×		×
<BM>.gcc9_1_0.vanilla		×				
<BM>.gcc9_1_0.finstr		×		×		
<BM>.gcc9_1_0.scorep		×			×	×
<BM>.gcc9_1_0.scorep-no-filter		×		×		×
<BM>.clang10_0_0.vanilla			×			
<BM>.clang10_0_0.finstr			×	×		
<BM>.clang10_0_0.scorep			×	×		×
<BM>.clang10_0_0.scorep-no-filter			×		×	×

Table 3.2.: The different configuration names indicate that they represent (1) automatic compiler instrumentation w/o filtering and **without** a measurement library attached, (2) automatic compiler instrumentation w/o filtering and **with** a measurement library attached, and, (3) automatic compiler instrumentation w/ filtering. Note that GCC 4.9.4 is used with Score-P 3.0, whereas GCC 9.1.0 and Clang 10.0 are used with Score-P 6.0. Also note that for measurements on Intel Sandy Bridge processors the **scorep** flavor is used **without** the inline filter.

obtained on Intel Xeon processors with Sandy-Bridge, and Haswell microarchitecture. In particular, the measurements with GCC 4.9.4 are performed on the older Sandy-Bridge and the Haswell processors, while the newer compiler versions are used solely on processors of the Haswell microarchitecture. Nodes are used exclusively with fixed frequencies, and HyperThreading disabled.

We follow the naming scheme `<BM>.<compiler>.<flavor>` to distinguish the benchmark versions and show the differences in Table 3.2.

Given the large number of HWPC available in current processors, we focus on specific events that are commonly used in performance analysis. The investigation of the memory subsystem is important, hence, the influence on HWPC that measure the second and last-level cache is of interest. On Haswell-based machines, PAPI offers the events `L3_TCM`, `L2_TCA`, `L3_TCA`, `L2_TCM`, `L2_DCM`, and `L2_DCA` to count the level three total cache misses and accesses, and level two total misses, data cache misses, and data cache accesses. Two other important metrics are instructions per cycle‘(IPC), as they indicate how well the

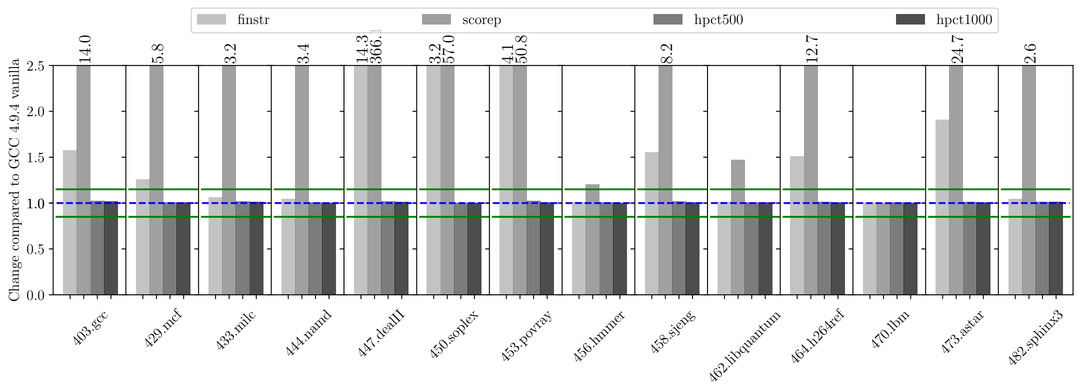


Figure 3.2.: The change of runtime for GCC 4.9.4 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep) for Score-P 3.0. For Score-P the **`inline`**-filter is disabled. The influence of HPCToolkit is shown for two sampling frequencies 500, and 1000 samples per second, respectively. Experiments run on Intel Xeon E5-2670 (Sandy Bridge).

code can leverage the concurrent execution in super-scalar processors, and the number of branch mispredictions. Branch mispredictions may lead to pipeline flushes, which are expensive and undesirable. A user can measure these events using the PAPI events `BR_CN`, `BR_MSP`, `TOT_INS`, and `TOT_CYC`. In many cases, not only the bare numbers are important, but rather their relationships. Thus, from our benchmarks, we also compute derived metrics, such as the *cache miss rate*, which is typically used to determine if the code shows efficient memory behavior. As other HWPC events are, however, of interest in specific situations, the full set of measurements can be found in the appendix, see Chapter A.

Raw PAPI Metrics

We present the raw PAPI event counts first, and derive the aforementioned ratios thereafter. All values shown in these sections and plots are measured on Intel Xeon E5 2680v3 (Haswell microarchitecture) processors, unless stated specifically otherwise. The data for GCC 4.9.4 and Score-P 3.0 was also obtained on Intel Xeon E5 2670 (Sandy Bridge microarchitecture) processors, which gives us the opportunity to look at results for different hardware generations. Please note that different hardware generations offer different HWPC. Hence, we do only present some raw PAPI counter metrics in this section, as the events obtainable differ between the machines. The subsequent sections present the measurement results for the different software versions.

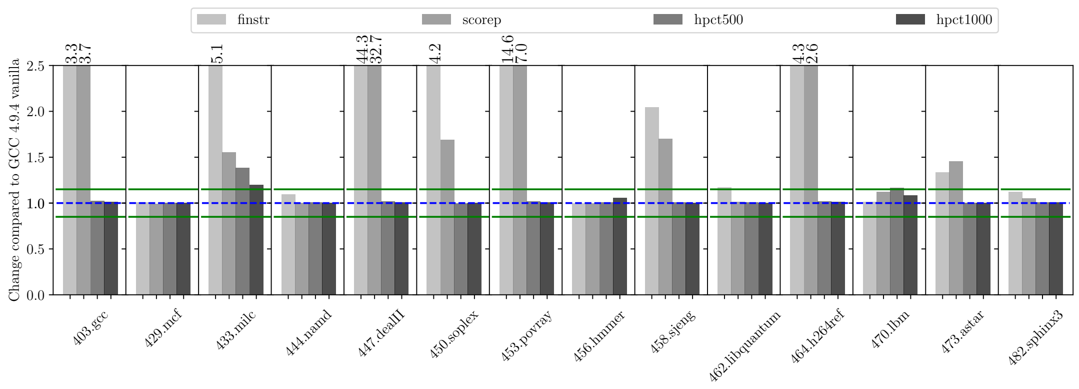


Figure 3.3.: The change of mispredicted branches for GCC 4.9.4 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep) for Score-P 3.0. For Score-P the **inline**-filter is disabled. The influence of HPCToolkit is shown for two sampling frequencies 500, and 1000 samples per second, respectively. Experiments run on Intel Xeon E5-2670 (Sandy Bridge).

GCC 4.9.4, Score-P 3.0 and HPCToolkit In our initial experiments, we measured the influence of Score-P 3.0 and HPCToolkit across the SPEC CPU 2006 *C*/*C*++ benchmarks for GCC 4.9.4. These experiments were run on Intel E5-2670 processors based on the Sandy-Bridge microarchitecture. The goals are (1) preparing a study of state-of-the-art tools for performance measurement and their influence on runtime and HWPC, and, (2) investigating to what extent dilations in HWPC measurements correlate to runtime increases. Furthermore, the study of the tools' influence on the values obtained for HWPC serves as a point of reference for all subsequent measurements presented in this chapter.

For all our plots, a range of 15 % around the vanilla median (blue line) is indicated by green lines. We found this range mentioned as acceptable runtime overhead for performance analysis in literature [99], [114]. In general, less perturbation is preferable, as it means that the measurement much better reflects the application's original behavior.

In Figure 3.2, we see the heavy impact of the automatic compiler instrumentation on the target's runtime. However, the amount of influence varies greatly across the different benchmarks. Particularly the *C*++ codes are heavily influenced, due to the instrumentation added to small accessor functions, such as `operator[](const int idx)` of the standard library's `vector` class. As the most prominent example, 447.dealII suffers the largest slowdown with almost a factor of 500×. The 447.dealII benchmark is implemented in *C*++ and relies on various template meta-programming techniques [2]. Such techniques typically lead to many small functions that individually do not contribute much runtime.

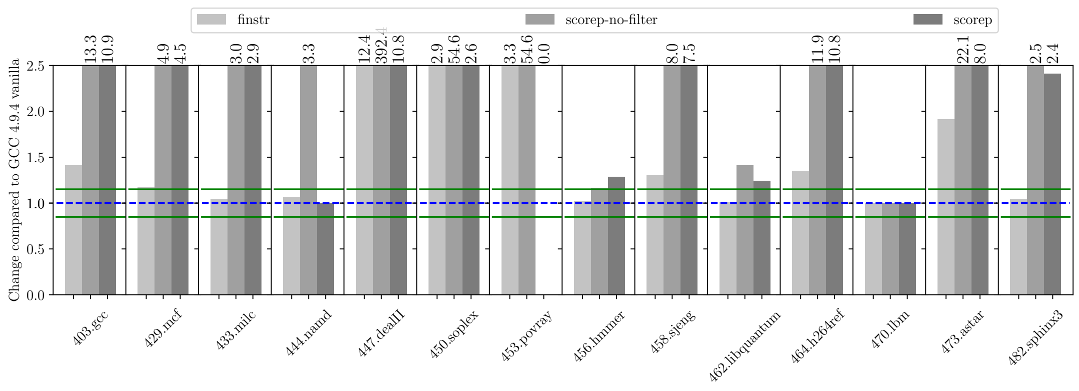


Figure 3.4.: The change of runtime for GCC 4.9.4 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep-no-filter and scorep) for Score-P 3.0. In scorep-no-filter the **`inline`**-filter is disabled. Experiments run on Intel Xeon E5-2680v3 (Haswell).

Thus, when automatic compiler instrumentation is enabled, each function is instrumented and significantly contributes to runtime overhead.

On the other hand, 470.lbm suffers only minor runtime penalty. The benchmark is implemented in *C* and consists of only a few functions that implement deeply nested and computationally demanding loops. Hence, the functions exhibit enough work to cover the overhead introduced by the — comparably — few instrumentation statements for the timing measurements introduced by Score-P.

Given that developers often use the runtime overhead observed for a performance measurement as an indicator for the perturbation introduced to the target's behavior, it is of interest to compare the influence of the measurement system onto different metrics. Figure 3.3 shows the changes for the `BR_MSP` PAPI event, i.e., the number of mispredicted branches, compared to vanilla. The important observation is that although in a few cases the `BR_MSP` PAPI event increases to an extent that would render it useless, in other occasions, the counts seem still valid. In particular, the cases 429.mcf and 444.namd are examples in which the runtime overhead is unacceptably large, but the PAPI value is almost not perturbed.

We continue the experiments with keeping the compiler and Score-P version fixed, but move to the more modern Intel Xeon E5-2680v3 processor. This allows us to compare any potential differences that the change of hardware may have introduced into the experiments. We have, however, omitted the measurements for HPCToolkit, as we focus on instrumentation and the development and benefits of the Score-P filtering, before we explain how PIRA Chapter 5 improves the filtering for Score-P. Hence, we enable Score-P's inline-filter using the GCC plugin for these experiments to evaluate the benefit of the filtering on the runtime overhead and the other metrics.

Similar to the measurements on the Sandy-Bridge processor, using automatic compiler instrumentation can have a significant impact on the runtime of the target application, such as for 447.dealII, see Figure 3.4, flavor finstr. However, we also find small differences in the degree of perturbation introduced, e.g., the overhead for 429.mcf on Sandy-Bridge was $5.8\times$ whereas it is $4.9\times$ on Haswell. On the other hand, for 447.dealII, the slowdown increased to $\approx 392\times$ on Haswell from $\approx 366\times$ on Sandy-Bridge. Since the actual perturbation observed is very similar between the two hardware architectures, we refer the interested reader to Appendix A that provides the respective data.

GCC 9.1.0 and Score-P 6.0 Similar to the preceding paragraph, we perform measurements with Score-P with and without inline filter. Score-P's filtering is implemented as a GCC plugin and uses a different API to eventually perform the measurement. We do not consider the different API to result in tremendous performance advantages. It addresses the technical challenge that Score-P has to identify the function names in the binary, which was previously performed using either the utility tool `nm`, or the utility library `libbfd`, both from the binutils package.[4] Both approaches struggle in certain cases to resolve the target's symbols reliably, e.g., when a software is built from many shared libraries.

The runtime increase for instrumentation with Score-P 6.0 and GCC 9.1 is similar to those of Score-P 3.0 and GCC 4.9. When we compare the slowdown introduced, we find that the newer Score-P version introduces a slightly higher slowdown. This may be due to a lower base runtime in the vanilla version of the target application when compiled with GCC 9.1, which we observed for all targets. In particular, the runtime was reduced by $\approx 2\,\%$ when compiling with GCC 9.1, with a maximum decrease of $\approx 7\,\%$. Also, our measurements suggest that for some of the HWPC, the values obtained for the vanilla flavor of the benchmarks are considerably different. However, we consider a thorough investigation on the root cause of these changes to be beyond the scope of this work. Our measurements are attached in Appendix A for further reference.

Clang 10.0.0 and Score-P 6.0 Finally, we want to establish an understanding of the influence of measurement for the Clang compiler, as it is (1) a popular compiler, and, (2) the base compiler for PIRA, see Chapter 5. While the Score-P plugin for Clang [131] is not yet publicly available, Clang itself offers the option to perform the automatic compiler instrumentation after its inline optimization. Hence, in this section, we consider this

[4]`https://www.gnu.org/software/binutils/`, accessed July 2021.

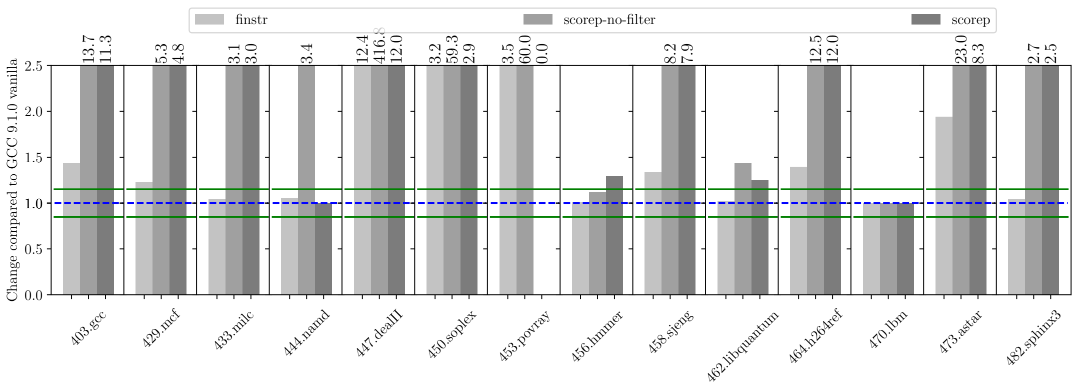


Figure 3.5.: The change of runtime for GCC 9.1.0 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep-no-filter) for Score-P 6.0. For Score-P the **inline**-filter is disabled. Experiments run on Intel Xeon E5-2680v3 (Haswell).

option equivalent to the Score-P filtering mechanism, although Clang still relies on the Cyg Instrumentation API.

From Figure 3.6, we see that the impact of the full Score-P instrumentation without filtering is very similar to the results with GCC. Minor differences are, due to our understanding mostly because of different vanilla runtimes. However, from our measurements we find that 429.mcf significantly benefits from the instrumentation being applied after the inline pass. With the Score-P GCC plugin, this target still showed considerable runtime overhead. For 453.povray, the measurement did not succeed due to an internal Score-P error. To our understanding this was caused, by regions being recorded in an incorrect order triggering assertions within the Score-P runtime.[5]

In Table 3.3, we see the considerable reduction that automatic filtering has on the number of events generated per second. However, considering that HPCToolkit records 500 samples/seconds, it is of little surprise that the runtime overhead introduced by both the filtered and the unfiltered instrumentation is higher.

Looking at the HWPC values obtained for the `BR_MSP`, i.e., the number of mispredicted branches, we find that the values for this counter are not perturbed to the extent the runtime overhead would suggest. In particular, the code 433.milc, 450.soplex, 473.astar, and 482.sphinx3 maintain a `BR_MSP` value within the $\pm 15\,\%$ range of the vanilla's execution. For other target applications, the values are much less perturbed than the runtime overhead suggests. Hence, from our measurements we conclude that for some HWPC no clear

[5]We plan to investigate this further to submit a proper bug report to the Score-P developers.

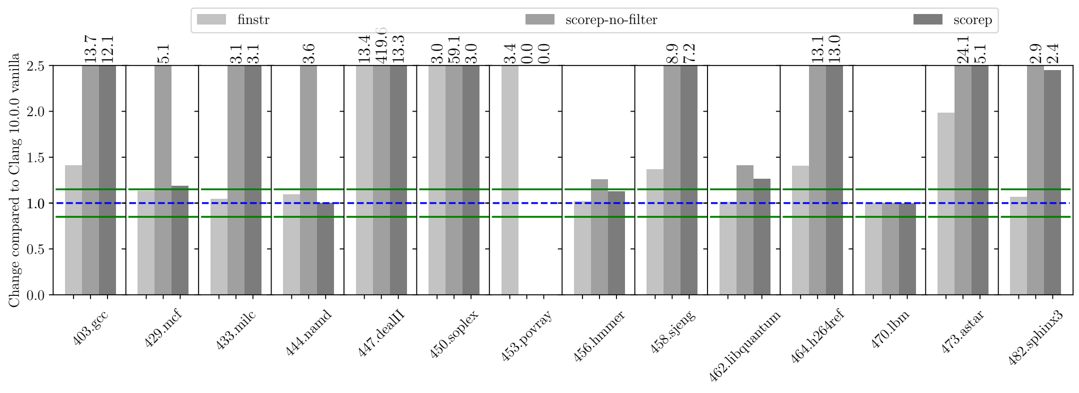


Figure 3.6.: The change of runtime for Clang 10.0.0 for automatic compiler instrumentation without and with **inline** filtering, respectively. Experiments run on Intel Xeon E5-2680v3 (Haswell).

correlation between the runtime overhead and the counter value perturbation exists. This means that a performance analyst needs to perform baseline measurements for each metric they want to inspect for performance analysis.

Derived PAPI Metrics

While the previous section showed the influence of automatic compiler instrumentation for single events, commonly, derived ratios are of interest to an analyst. For example, the total number of cache misses may not be particularly telling, but the ratio between cache accesses and cache misses is. If only one of the events is perturbed heavily, the ratio is, however, no longer usable for analysis purposes. Hence, this section considers the derived metrics *instructions per cycle*, *cache miss rate* for the second level data (L2D) cache, and the *branch misprediction rate*. We limit the figures shown in this section to the values obtained for the Clang compiler, and refer the interested reader to Appendix A for the data obtained for the other compilers.

Instructions per Cycle The metric Instructions per Cycle (IPC) is commonly used to quantify how well the target application can utilize and saturate the available functional units in super-scalar processors, which can perform more than one instruction per cycle. As an example, consider that an Intel Xeon 2670v3 based on the Haswell microarchitecture can issue up to four instructions per cycle. One important factor that limits the number of instructions to dispatch are (data) dependencies between different instructions. However,

Benchmark	Normalized to Profile		Normalized to Vanilla	
	w/ filter	w/o filter	w/ filter	w/o filter
403.gcc	7,986,609	8,266,616	80,294,531	112,479,600
429.mcf	760,770	5,495,703	832,905	34,183,211
433.milc	5,922,057	5,930,878	15,821,067	15,847,823
444.namd	7,363	8,222,170	7,354	32,335,363
447.dealII	8,862,098	7,761,603	127,653,218	3,363,968,006
450.soplex	7,072,296	11,156,501	20,575,488	638,534,291
453.povray	–	–	–	–
456.hmmer	2,558,326	3,396,061	3,338,407	4,994,915
458.sjeng	7,910,639	8,222,733	59,058,689	76,793,948
462.libquantum	3,112,402	4,093,760	3,929,778	6,348,063
464.h264ref	10,364,745	10,392,210	140,063,680	145,097,818
470.lbm	29	29	29	29
473.astar	8,750,442	10,452,247	48,218,560	276,855,838
482.sphinx3	6,078,200	6,881,301	14,377,437	19,434,047

Table 3.3.: Number of function visits recorded per second via a Score-P instrumentation for Clang 10 and Score-P 6. The table lists the number of events per second w.r.t. the profiling runtime (Normalized to Profile) and to a vanilla execution of the target (Normalized to Vanilla). In the case of 447.dealII, the runtime overhead is large enough to reduce the number of events recorded per second. Measurements are obtained on Intel Cascade Lake processors.

due to the architecture – different functional units are accessed via so-called dispatch ports — the instruction mix also influences whether a target application can saturate the processor. Hence, a low IPC is an indicator that many dependencies exist, or the instruction mix is suboptimal for the target processor.

Looking at the data in Figure 3.8, we find that the IPC increased to a value outside the acceptable range in five out of 14 cases when filtering is applied and in six out of 14 cases when no filtering is applied, respectively. In many cases, the IPC, however, stayed within the acceptable limits and none of the measurements resulted in smaller values for the IPC. Given that the instrumentation adds significantly many function calls this was surprising. For each function call function arguments are moved into respective registers for function argument passing, which may even require saving live registers for later restore. Even with the runtime increase we awaited more perturbed data for the IPC of the application.

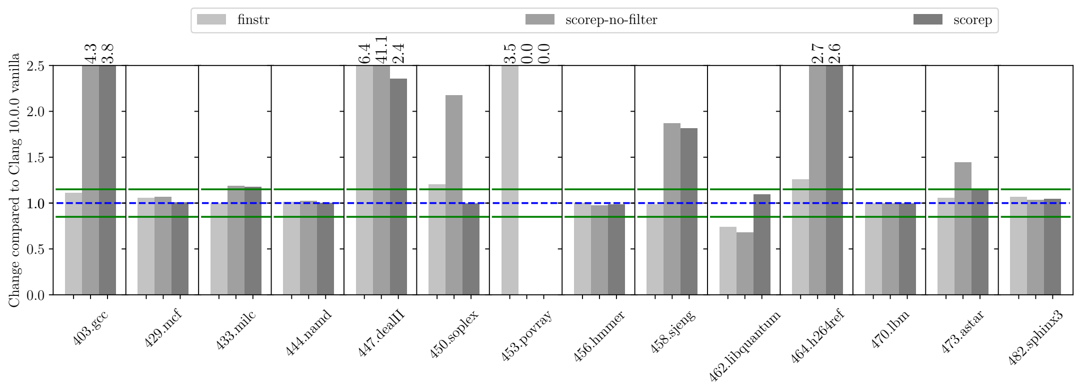


Figure 3.7.: The change of conditional branches for Clang 10.0.0 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep-no-filter) for Score-P 6.0. For Score-P the **`inline`**-filter is disabled. Experiments run on Intel Xeon E5-2680v3 (Haswell).

Cache Miss Rate The cache miss rate is used to quantify the target application's ability to use the fast on-chip memory instead of requiring to go out to main memory. It is the ratio between the respective cache accesses compared to the cache misses at the particular hierarchy level. Given the cache hierarchy of modern processors, analysis typically starts with the second-level or the last-level cache. Usually a miss in a lower hierarchy level results in access to the higher level.

Figure 3.9 shows the results for the L2D miss rate for the different target applications. We see that for most benchmarks the miss rate is perturbed when calculated on the HWPC values obtain for the instrumented binaries. For some benchmark applications this perturbation is within the acceptable limits, e.g., 433.milc and 462.libquantum. However, for other benchmark applications, the value of the derived metric is much lower than the vanilla measurement, e.g., 403.gcc and 458.sjeng. In the case of 456.hmmer, the compiler instrumentation resulted in a lower L2D miss rate. We assume that the additional accesses to record and store the profiling data interact with the memory subsystem and lead to this situation. Depending on the cache efficiency of the target application, the resulting artifact is more or less pronounced. For 453.povray, the Score-P runs did not succeed, i.e., the data obtained was invalid.

Branch Misprediction Rate Modern x86 processors rely on out-of-order execution and deep execution pipelines to achieve high performance. A pipeline stall, i.e., the processor has no instruction to feed into the pipelines, reduces the efficiency, and is a reason for

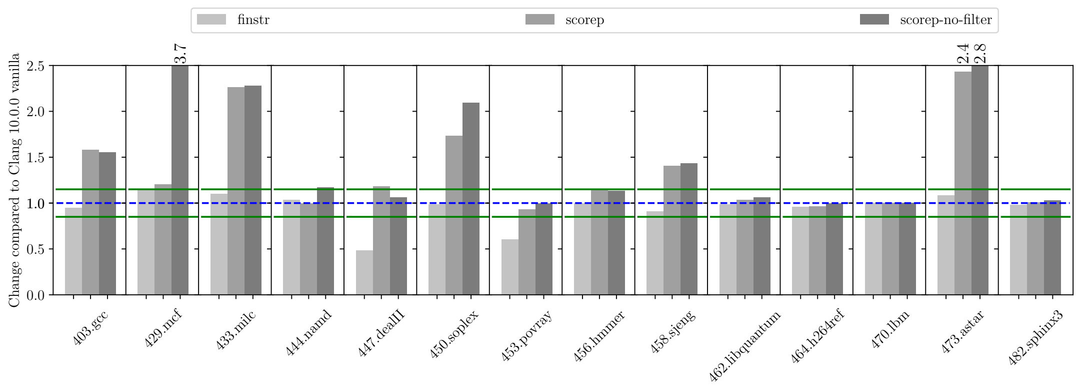


Figure 3.8.: The change of instructions per cycle for Clang 10.0.0 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep-no-filter) for Score-P 6.0. For Score-P the **`inline`**-filter is disabled. Experiments run on Intel Xeon E5-2680v3 (Haswell).

so-called branch prediction. In branch prediction, the processor makes educated guesses which path of a branch instruction to take, and speculatively executes the respective instructions. Should the processor speculate incorrectly, it needs to reverse already performed operations, which is called a rollback and is very expensive. Hence, in performance analysis, the branch misprediction rate is used to investigate a target execution's ability to maintain a working pipeline.

The branch misprediction rate for our measurements is shown in Figure 3.10. We see that in many cases of the instrumented application with measurement system attached, the misprediction rate is much lower than in the vanilla execution. In 10 out of 14 cases (scorep) the metric is significantly lower than in the original execution, and stays within the accepted $\pm 15\,\%$ for the remaining four target applications. For many cases there is little difference between the filtered and the unfiltered Score-P measurement. However, in the case of 429.mcf and 444.namd the difference is significant. Score-P internally performs a check whether the region identifier — in the case of Clang the function's address — is a valid region identifier, i.e., whether Score-P detected a function at that address using `nm`. Given the many measurement events recorded, our assumption is that this check skews the branch predictor significantly.

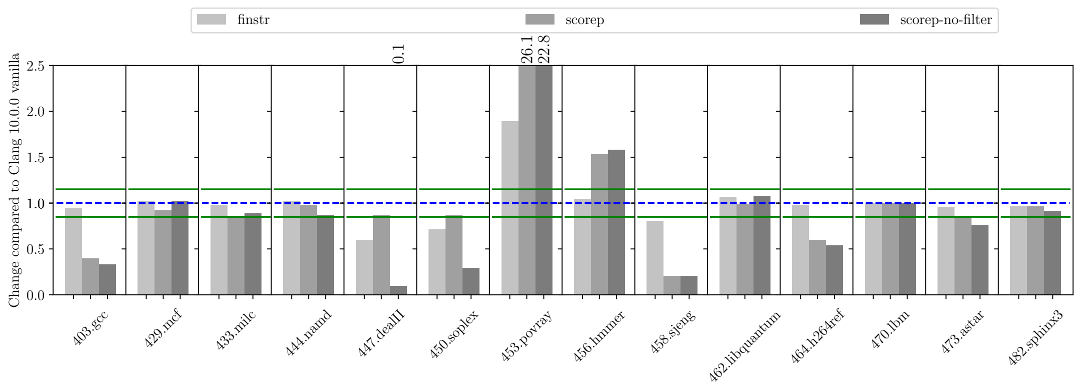


Figure 3.9.: The change of level 2 data-cache miss-rate for Clang 10.0.0 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep-no-filter) for Score-P 6.0. For Score-P the **`inline`**-filter is disabled. Experiments run on Intel Xeon E5-2680v3 (Haswell).

3.3. Summary

We find that current automatic compiler instrumentation via Score-P introduces significant slowdown in 10 out of 14 applications without filtering and in 8 out of 14 cases with filtering. This is particularly prominent in *C*++ applications that use many small functions, e.g., to access data. These small functions generate the largest share of runtime slowdown, and, in addition, lead to a profile that is considerably hard to navigate for an analyst.

We also found that some HWPC event counts obtained were almost not perturbed. As a result, the derived metrics that involved such events were also close to the vanilla measurements. This means that despite the large application runtime slowdown, insight obtained from the particular experiments could be used for performance analysis.

Our data shows that the filtering available in Score-P reduced the runtime overhead significantly in 5 out of 14 cases within the SPEC CPU benchmark suite. As a result, two additional measurements can be considered valid. In addition to the runtime overhead reduction, the resulting profile typically contains fewer regions. However, the filtering impact is not as beneficial as desired w.r.t. the measurement runtime overhead reduction.

In summary, our experiments show that (1) analysts should perform baseline measurements for every metric obtained, (2) runtime slowdown of current automatic compiler-instrumentation can render experiments impractical due to long runtimes, and, (3) the profiling data generated may be hard to navigate given the large number of regions contained in the profile.

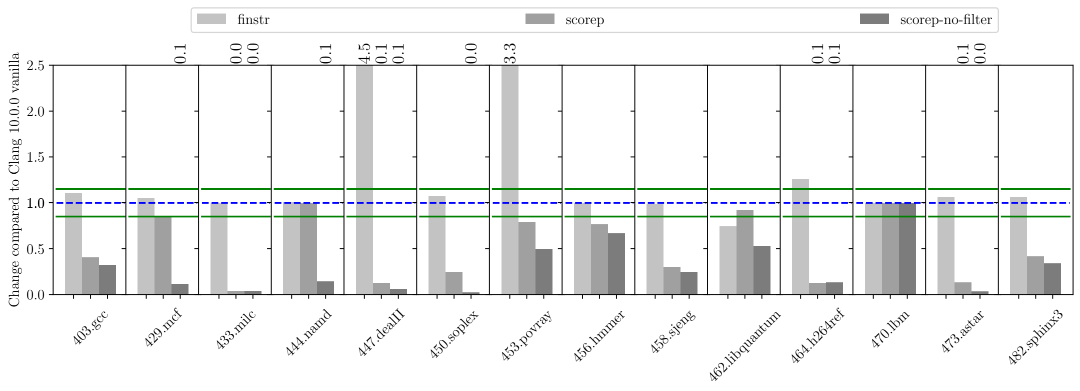


Figure 3.10.: The change of branch-misprediction rate for Clang 10.0.0 for automatic compiler instrumentation with empty hooks (finstr) and with attached measurement system (scorep-no-filter) for Score-P 6.0. For Score-P the **inline**-filter is disabled. Experiments run on Intel Xeon E5-2680v3 (Haswell).

4. MetaCG – Annotated Whole-Program Call-Graphs

The chapter is based on the following publication and contains verbatim excerpts of parts which were contributed by the thesis author. It uses MetaCG 0.2.1[1] for experiments unless stated explicitly otherwise.

> **Lehr, Jan-Patrick** and Hück, Alexander and Fischler, Yannic and Bischof, Christian. 2020. *MetaCG: Annotated Call-Graphs to Facilitate Whole-Program Analysis* [76]

Whole-program analysis is important in many aspects. While we created PIRA, we realized that we need a lightweight program representation that allows to implement analyses at the whole-program level, independent of a specific compiler distribution or version. Moreover, we wanted the representation to be freely annotatable with user-defined information we obtain from analyzing the program's AST. To serve this purpose, we created *MetaCG* — a lightweight and annotatable CG library, see Figure 4.1.

It consists of the graph library as its main component, together with the tools to construct the CG and to validate a constructed CG w.r.t. its edges. A MetaCG graph can be serialized into a JSON file, and, hence, is accessible even for tools that do not explicitly rely on the MetaCG software. As part of MetaCG, the tool CGCollector is used to construct CGs using a Clang-tooling approach, i.e., processing Clang's AST. The construction is performed in a two-step process. In a first step, translation-unit local CGs are constructed. The second step links all partial CGs together to arrive at the final whole-program CG. To validate the constructed CGs we created the tool CGValidate that takes as its inputs a MetaCG file and a Cubex profile of the target application. It then checks if it can find all edges from the profile in the MetaCG graph.

[1]Available at `https://github.com/tudasc/metacg`.

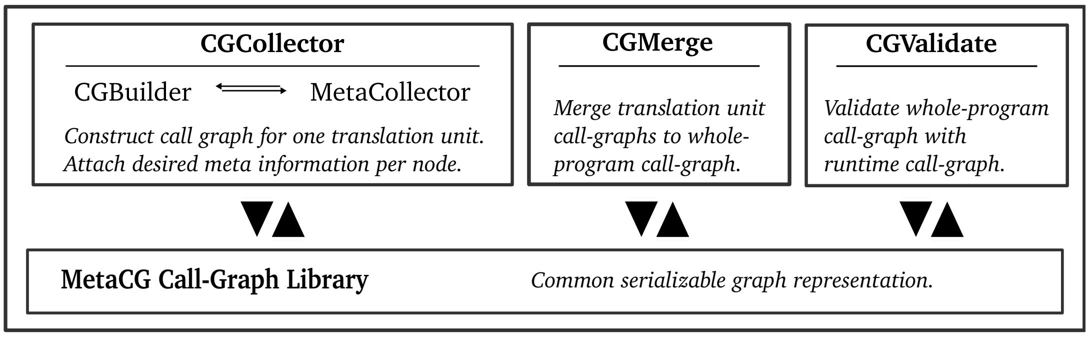

CGCollector
CGBuilder
MetaCollector
Construct call graph for one translation unit. Attach desired meta information per node.
CGMerge
Merge translation unit call-graphs to whole-program call-graph.
CGValidate
Validate whole-program call-graph with runtime call-graph.
MetaCG Call-Graph Library
Common serializable graph representation.

Figure 4.1.: Overview of the MetaCG software components. The *Call-Graph* library is the common graph implementation used for data exchange. *CGCollector* is used to construct the MetaCG for a single translation unit, *CGMerge* merges the partial CGs to a whole-program CG, and *CGValidate* is used to test for missing edges given particular executions of the target program recorded in Cubex files.

4.1. Call-Graph Library

The graph library of MetaCG provides a common interface to parse, annotate, and serialize MetaCG files. This helps tool developers as they do not need to take care of the burden of writing custom parsers and similar boilerplate. In addition, it allows for, generally speaking, more robust testing and facilitates the adoption of such technologies.

In Listing 4.2 we show an example of a simple input program in *C*++ and its corresponding MetaCG serialization. Each node is represented as an entry in the file, with the mangled function name as its key. The mangled name is required for *C*++ and uniquely identifies a function in the program. In the mangled name both the function name and the type signature of the function is encoded, which generally allows overloading based on the type signature of a function's parameter list. Each node has entries for various information that are relevant for the CG construction. The examples show MetaCG version 1.0 file format. MetaCG's version 2.0 file format adds information about the tool that constructed the MetaCG file and which file version it is. The representation of the CG is, however, unchanged and contains the following fields.

Callees: A list of keys, i.e., mangled function names, that are potentially called from this function. This constitutes the list of child nodes in the actual CG.

Callers: A list of keys, i.e., mangled function names, that potentially call this function. This constitutes the list of parent nodes in the actual CG.

```cpp
struct Base {

  virtual int bar() {
    return 1;
  } // some value

};


struct Derive : public Base {

  virtual int bar() {
    return 2;
  } // some value

};


int add(Base *obj, int v) {

  auto value = obj->bar() + v;

  return value;
}
```

Code C++ example input code with an inheritance hierarchy and virtual method calls.

```json
"Base::bar": {
  "callees": [],
  "parents": ["add"],
  "overrides": [],
  "overriddenBy": ["Derive::bar"],
  "meta": {
    "pira": {"numStatements": 1}
  }},
"Derive::bar": {
  "callees": [],
  "parents": [],
  "overrides": ["Base::bar"],
  "overriddenBy": [],
  "meta": {
    "pira": {"numStatements": 1}
  }},
"add": {
  "callees": ["Base::bar"],
  "parents": [],
  "overrides": [],
  "overriddenBy": [],
  "meta": {
    "pira": {"numStatements": 2}
  }}
```

MetaCG Example using demangled names instead of mangled names.

Figure 4.2.: MetaCG has one entry per function: (1) calling functions (**parents**), (2) called functions (**callees**), (3) meta information (**meta**), (4) overriding functions (**overriddenBy**), and, (5) overridden functions (**overrides**).

OverriddenBy: List of functions that override this function, i.e., functions further down the inheritance hierarchy reimplementing this one.

Overrides: Similar, this is the list of functions that this particular function itself overrides, i.e., reimplements.

Meta: The `meta` field allows for annotation with user-defined information. It holds a key-value map that allows to attach meta information for a particular tool, e.g., PIRA. Hence, multiple tools can export required information into a common file format that can be passed between tools and infrastructures.

In its current version, MetaCG does not model edges explicitly. This results in two sim-

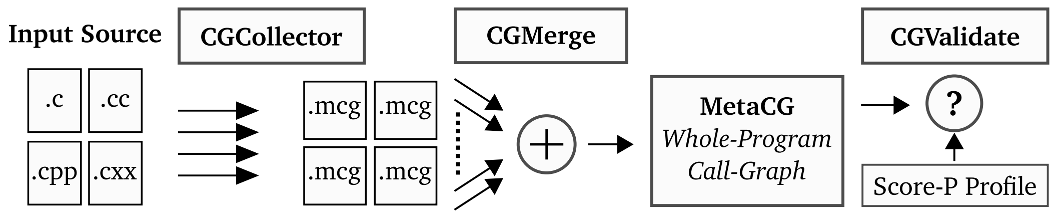


Figure 4.3.: MetaCG: Individual source files are compiled to their respective partial MetaCG file. Subsequently, CGMerge combines the partial CGs and constructs the final MetaCG. Optionally, CGValidate is used to check for, and add missing nodes and edges.

plifications that, however, result in less precision. In particular, the CG cannot distinguish different potential call-sites, and it can not differentiate between types of calls, e.g., direct calls and virtual calls. In *C*, this does not impact the precision to a large extent. In *C*++, however, calls can be direct or virtual. This means that the compiler can bind the target address already to the call instruction (direct call), or it needs to emit instructions to perform an indirect call through the virtual function table of the respective object (virtual call). This distinction can, obviously, reduce the set of potential call targets for a given call-site, should the call-site be direct. However, in our current use case, this distinction is less relevant, as we mainly use the general program structure and which components are glued together. As a result of the current lack of edges, tools that use MetaCG, at the moment, cannot determine whether a specific call is a virtual call or a direct call.

4.2. Call-Graph Construction

The CG construction is performed by the tool CGCollector and a subsequent CGMerge. Thereafter, the use may choose to perform a validation step, in which an application's profile can be used to validate that each recorded call edge is present in the statically constructed MetaCG. In the subsequent sections, we present the CGCollector/CGMerge approach in more detail. An overview of the MetaCG workflow is shown in Figure 4.3.

4.2.1. CGCollector

The construction of a translation-unit-local CG is performed by CGCollector. It uses the Clang-tooling API to process individual ASTs and write partial MetaCG files to be subsequently merged by CGMerge. The processing at the AST level presents some interesting

```c
typedef int (*Func)(int) ;

/* Defined in a separate translation unit */
Func func_one(int x);
Func func_two(int x);

/* Return value determined in separate translation unit */
Func get_func(int a) {
  if (a == 1) return func_one(a);
  if (a == 2) return func_two(a);
}

int main() {
  /* Opaque which value returned here */
  auto f = get_func(1);
  f(42);
  return 0;
}
```

Listing 4.1: An example usage of function pointers across TUs. The call to `get_func` returns a value that depends on two functions defined outside the current TU. Hence, their return values cannot be determined.

challenges w.r.t. the multitude of syntax constructs that programmers can use – particularly in *C*++. In addition to these syntactic challenges, consider the example in Listing 4.1. In `main`, a function pointer is obtained via a call to `get_func`, and subsequently invoked. However, the result of `get_func` depends on functions for which the definition is not present in the current translation unit (TU), and, hence, cannot be analyzed.

In *C*/*C*++, a CG construction tool needs to handle (1) direct calls to a function, (2) calls via function pointers, and, (3) virtual function calls (*C*++ only). The different cases can occur between functions defined within the same TU or across the boundaries of a TU. While some of these cases present no particular challenge, e.g., handling of direct calls to functions within the same TU, others, such as calls through function pointers across TUs, are particularly challenging. Consider a call to a function `foo` that receives a pointer to a function pointer as argument, writes through this pointer and passes the modified value to another function `bar` that calls the function pointer. CGCollector addresses the respective challenges to varying extent by applying different resolution strategies for direct calls, virtual calls and function pointers.

Direct Calls CGCollector considers calls as direct whenever the function symbol can directly be resolved to a declaration. This includes virtual calls of the form `obj.method( parameters)`, as `method` can be resolved to the method declaration present in the type of `obj` or one of its superclasses. Since the target symbol can be resolved, CGCollector adds the corresponding edge to the graph.

Consider a function `f` that invokes method `m` on an object `obj`. CGCollector adds respective nodes for `f` and `m` to the graph, and attaches the corresponding data, e.g., whether it is defined in the current TU, or if it is declared **`virtual`**. Thereafter, CGCollector adds the edge `f -> obj::m` to the graph independent of `m` being virtual.

Virtual Calls In addition to the steps mentioned in the preceding paragraph, CGCollector adds inheritance information for **`virtual`** methods. For all functions marked **`virtual`** in a target program, CGCollector performs class hierarchy analysis (CHA). Therefore, it computes the inheritance hierarchy for the particular class and whether the current function overrides any methods within the hierarchy. Any overwritten methods are added to the function's overrides set. It also stores if the method currently processed is overwritten by any other function, and adds it to the respective set.

Function Pointers For function pointers, CGCollector resolves the function symbol to the function's definition. Moreover, it inspects symbols passed as arguments and builds alias sets [125] to resolve calls via function pointers across function boundaries. While this allows to track function pointers in certain circumstances, it is not as capable as a data-flow analysis and leads to over approximation w.r.t. the different call sites in the program. In the case of CGCollector, this is reasonable as the resulting graph is inherently context independent and does not allow distinguishing different call sites.

4.2.2. Meta Collectors

MetaCG allows to add meta information per function node into the CG. The information is captured at the AST level by so-called `MetaCollectors`, i.e., implementations of Clang analyses that compute certain information and store it for the target function as `MetaInformation`. This `MetaInformation` object itself knows how to unparse its data into the respective JSON file, and how to read the unparsed data again into the MetaCG graph library. Consequently, the addition of new information into an existing tool requires the implementation of two classes. This enables tool-specific information to be added to the MetaCG in a convenient and lightweight way.

```
struct A { // not a statement
  /* Sum of number of statements in body is count for foo: 3 */
  void foo(int *b) {
    int a = 5; // one statement
    for (int i = 0; i < a; ++i) { // one statement
      *b += a; // one statement
    }
  }
};
```

Listing 4.2: Example for the number of statements within a function body as captured by the Number of Statements meta collector and used in PIRA.

The different `MetaCollectors` are registered in the CGCollector and are invoked individually on each function in the Clang AST. The order of the functions follows the order in which they are defined in the TU. Subsequently, we present simple `MetaCollectors` that we use in the subsequent chapters of this thesis.

Number of Statements This `MetaInformation` computes the number of statements within a function definition. The information builds the basis for the static heuristics in PIRA to estimate the amount of work within a function, *cf.* Chapter 5, and which have previously been explored in [56]. Its notion of a statement resembles a *C++* statement that results in observable behavior, see Listing 4.2 for an example.

Unique Types This `MetaInformation` counts the number of unique types used within each function. In addition, it captures the total number of unique types within the program. When counting, the `MetaCollector` does not consider pointer-to or reference types as different types. Instead, it strips pointers and references until a type is found. Note that `typedefs` are considered types, as they add to the complexity that a programmer has to handle by memorizing.

Unique Variables This `MetaInformation` captures the number of unique variables declared within a function. We use the metric, alongside others, in Chapter 6 to evaluate the complexity of a target application and the complexity reduction achieved with the mini-app extraction proposed.

File Properties Being able to determine if a function stems from a system include or user code can be important. Thus, we implemented the `FilePropertyMetaInformation`. While

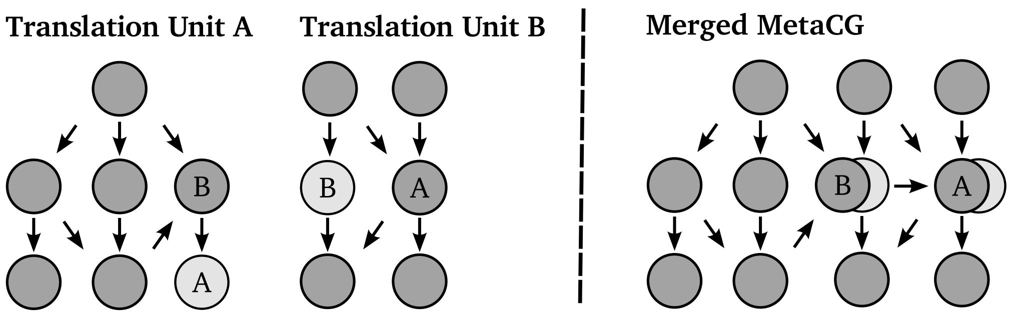


Figure 4.4.: In the example, function `A` is only declared in TU A and function `B` is only declared in TU B, indicated by the light gray circles. CGMerge uses the key to fuse the respective function nodes and to combine the available information. In the merged MetaCG, the connectivity information for both function `A` and `B` are available.

the information is accessible from within the compiler, a tool that is implemented as a stand-alone solution may not have access to the information. Currently, the file properties include whether the function comes from a system directory, together with the full path of the file. This allows to implement, e.g., directory-specific filters.

4.2.3. Merging of Translation Units

Once the TU-local CGs are built, they need to be merged to form the final whole-program CG. This is implemented in the tool CGMerge. The merging is performed in a straightforward way, in which the different node identifiers, i.e., mangled names, are matched and edges are inserted. During the merge, the tool also constructs the inheritance hierarchy specific calls. This means that it, based on the `overrides` and `overriddenBy` fields stored for each method, constructs the **`virtual`** hierarchy. The hierarchy is then used to insert all potentially called methods for a given call site. An example thereof is given in Figure 4.4.

In addition to the merging of the structural CG data, the tool also needs to merge the metadata. The current implementation requires a user to provide a merge strategy, i.e., how potentially different information for the same function, should be treated during the merge. As an example, consider the case of the Number of Statements `MetaInformation`. If a function has only been declared in a TU `A`, the Number of Statement count will be zero as there is no function definition to analyze. For TU `B`, which defines the function, the Number of Statement count will (most likely) be greater than zero. Hence, the merge strategy needs to decide and implement that the Number of Statement count for the function's definition is used and not the one determined for the declaration.

4.2.4. Validation

To rate the quality of the CG construction, the MetaCG package supplies the CGValidate tool. It uses a runtime profile of the target application in Cubex[2] format and checks whether the edges present in the profile are also contained in the MetaCG. If it cannot find an edge, it emits a warning, and allows to patch both missing nodes and missing edges into the CG. The mechanism is sensitive to the input data provided to the target-application's execution, and a more complete picture can only be obtained when using multiple, distinct input data sets for different runs of the target application. Hence, depending on the number of data sets used, it prevents a statement about the general quality of the CG, but allows inspecting whether the CG contains all required edges for a particular execution.

CGValidate uses a call-path profile to validate the CG. This means that it may identify the same missing edge multiple times, e.g., the missing call edge `foo -> bar` can be found within the calling context of functions `A` and `B`, and CGValidate will report both misses. In addition, it will report both directions of the missed edges. This means that it checks for the parent and the child relation in the CG. This is a deliberate choice, and, while we did not encounter a case in which only one of the relations was present in the MetaCG, we believe that distinguishing both is valuable.

4.3. Evaluation

MetaCG is used across our projects PIRA (see Chapter 5), Mini-AppEx (see Chapter 6), and TypeART [48]. While we leave the tool-specific evaluation for PIRA and Mini-AppEx to their chapters, we present the results for our extension to TypeART in this section.

We pay special attention to CGCollector's ability to construct the CG, i.e., the number of missing edges, as well as quantify the impact of these missing edges. The number of missing edges is determined using CGValidate for particular executions of the target application. To quantify the impact of missed edges, we, first, insert missing functions and edges using CGValidate, and, second, execute PIRA's analyzer, called Profile Guided Instrumentation Selection (PGIS), that we present in more detail in Chapter 5.3. PGIS computes and outputs different metrics about the CG, such as the number of reachable functions. Hence, we can estimate the impact of missing edges.

We apply MetaCG to construct the CG for the applications eos-mbpt [31], the Ice-sheet and Sea-level Model (ISSM) [68], and two versions of the LULESH [60] mini-app. For the applications we evaluate the aforementioned completeness and impact of missed edges. In addition, we apply TypeART to the LULESH mini-app, and AD LULESH a version

[2]The profile file format for the Score-P[64] profiler.

of the mini-app that was enhanced with algorithmic differentiation (AD, Griewank and Walter [40]). For a thorough discussion of the particular use-case on AD LULESH see Hück et al. [49] and for a broader discussion on tool-support for AD see Hück [46].

Furthermore, we present a brief evaluation of the benefits that whole-program reachability analysis can provide to the allocation and type tracking and sanitizer tool TypeART. A brief description of TypeART is provided in the next paragraph. TypeART implements a TU-local allocation filtering as its default filter. This limits the analysis capabilities, potentially leading to many more allocations being tracked unnecessarily. Hence, we extend the filtering capabilities with a whole-program reachability analysis using MetaCG, and evaluate the filter's ability to reduce the number of tracked allocations.

We briefly introduce the TypeART framework and how the CG-based filter connects with the other components. Thereafter, we outline the target applications.

TypeART: Type Allocation and Runtime Tracking TypeART implements runtime tracking of type information for relevant memory regions using instrumentation. A high-level overview of the approach is shown in Figure 4.5. It is implemented as an LLVM compiler analysis which identifies relevant memory allocations, their types, and inserts the respective instrumentation. Moreover, it implements *allocation filtering*, i.e., it filters irrelevant memory allocations from tracking to reduce the overall runtime overhead introduced. It considers allocations as irrelevant if they do not reach an API function of interest, e.g., if they are not used as buffer in an MPI call. At runtime, an analysis tool can query the TypeART runtime library for type information of arbitrary pointer addresses. Hence, type mismatches or unallocated memory can be detected.

Allocation Filtering TypeART implements a forward data-flow analysis that determines if an allocation reaches a relevant API function. Therefore, the filter follows the definition-use chain within the current TU. Whenever it can prove that the allocation never reaches a relevant API function, it is filtered from measurement. If it cannot prove that an allocation never reaches a relevant API function, meaning that it reaches an API function or that the analysis is unable to find all uses, TypeART must emit instrumentation. Since large software is typically implemented modularly, it would be beneficial to extend the filtering capability to a whole-program approach.

Call-Graph Filter MetaCG is used to enhance the allocation filtering in TypeART. The whole-program CG information allows TypeART's allocation filter to perform reachability analysis across TU boundaries. To that end, the filter performs the normal forward data-flow analysis of the allocation filter. Should the analysis encounter a call to a function that is defined in a different translation unit, it queries MetaCG

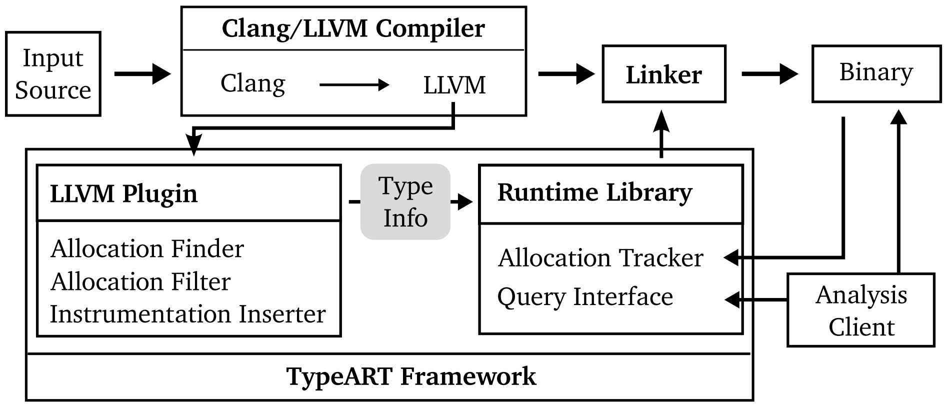


Figure 4.5.: The TypeART workflow: A target application is compiled to LLVM IR, adapted from the original paper [48]. TypeART analyzes the IR for relevant memory allocations, exports type information to a serialized type database, and inserts the required tracking instrumentation. At runtime, the analysis tool queries the TypeART runtime library for the desired type information.

to identify whether a path exists from the called function to any relevant function, e.g., MPI functions. If no such path exists, TypeART can filter the allocation from tracking, as it never reaches any MPI call. Should a path exist, TypeART treats this conservatively and does not filter the allocation.

eos-mbpt eos-mbpt computes the equation of state for nuclear matter by calculating energy diagrams in a perturbative expansion. A large portion of the computation is the calculation of high-order integrals. The software uses a Monte-Carlo scheme for the integration that is implemented in libcuba [41]. For additional spline interpolation it relies on the GNU scientific library [38], which internally uses BLAS routines for which we provide OpenBLAS [133]. The simulation is implemented mostly in *C* and consists of ≈ 8.5 million lines of code. However, as it uses *C++* for allocating memory and some of its console output, it needs to be compiled as *C++* program.

Ice-sheet and Sea-level System Model The Ice-sheet and Sea-level System Model (ISSM) simulates the evolution of ice shields considering the relevant physical phenomena, such as changes of temperature or falling snow. It computes each phenomenon in a dedicated kernel, and predefines sequences of such kernels for specific simulations,

Benchmark	LoC	F	F_R	E_m	E_c
LULESH	5,432	3,596	343	36	593
AD LULESH	36,109	12,073	4,202	460	4,641
eos-mbpt	8,571,813	31,162	470	38	241
ISSM	145,667	11,908	4,469	238	3,705

Table 4.1.: Lines of code (LoC), total number of functions (F), number of functions reachable from `main` (F_R), number of missing edges as detected by CGValidate (E_m), number of unique edges checked by CGValidate (E_c)

e.g., computing the *transient solution*. Since ISSM supports different mesh shapes and, therefore, different solvers, these solvers are selected for the particular use case and the flexibility is implemented via function pointers.

LULESH / AD LULESH The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a proxy application that "is representative for a simplified 3D Lagrangian hydrodynamics on an unstructured mesh" [60]. In our experiments, we use version 2.0 of the mini-app, which is implemented in C++ and uses MPI for parallelization. One of the interesting changes in this version is the possibility to introduce artificial load imbalance that would occur if multiple materials are being simulated. In AD LULESH, the built-in type **`double`** is replaced with the data type provided by the AD tool CoDiPack [110], and the MPI communication is implemented using the adjoint MPI library Medipack [111]. These changes enable the computation of derivatives using operator overloading.

Missing Edges

We use PIRA's analyzer PGIS to output statistics about the targets' call graphs to show the impact of missing edges. In particular, we present as key indicators (1) the total number of functions reachable from `main`, (2) the total number of edges, and, (3) the number of edges missed according to CGValidate. Additionally, we present the median of the number of statements per function. This value is used in PIRA, *cf.* Chapter 5, as the selection threshold for static instrumentation selection. Hence, large changes to this metric after patching the MetaCG may significantly affect PIRA's static selection heuristics.

Table 4.1 lists the project's complexity by the lines of code, as obtained with cloc [28], together with the number of functions and reachable functions found in the MetaCG. It also shows the number of edges identified as missing for a particular execution of the target application, and how many individual edges are checked. Note that the number

Benchmark	F	$\hat{F}$	Change	F_R	$\hat{F}_R$	Change
LULESH	3,596	3,628	32	343	353	10
AD LULESH	12,073	12,082	9	4,202	4,395	193
eos-mbpt	31,162	31,163	1	470	501	31
ISSM	11,908	12,105	197	4,469	4,920	551

Table 4.2.: The number of functions (F) and functions reachable (F_R) when the previously detected missing edges are inserted into the MetaCG, denoted as $\hat{F}$ and $\hat{F}_R$, respectively.

we present as the number of edges checked denotes the number of *unique* checks.[3] For LULESH, CGCollector misses 36 of 593 edges, and for AD LULESH, it misses 460 of 4,641 edges, respectively. In case of eos-mbpt, it misses 38 of 241 edges and for ISSM it misses 238 of 3,705 edges, for the respective target applications' executions.

Impact of Missing Edges

From Table 4.2 we see that CGValidate adds both functions and edges to the MetaCG. For the particular execution of LULESH it adds 32 functions and the 36 edges inserted lead to additional 10 functions being reachable. For AD LULESH another 9 functions are added and the additional 460 edges increase the number of reachable functions by 193. In the case of eos-mbt only a single additional function is added, and additional 31 functions are reachable. For ISSM we find that additional 197 functions are added to the MetaCG which leads to 551 more functions being reachable.

Similarly, Table 4.3 shows the total number of statements reachable, i.e., accumulated number of statements for all reachable functions, before and after the missing edges are patched into the MetaCG. We see that for LULESH both the number of statements and the median number of statements is unaffected by the additional edges. For AD LULESH, the number of statements increases by 300 additional statements, while the median of the number of statements does not change. In the case of eos-mbpt we find that, while the number of statements is increased only slightly (48), the median is affected and decreases from 19 to 17. The largest impact is observed for ISSM in which an additional 28,885 statements are accumulated across the reachable functions in the MetaCG. However, the median of the statement distribution is affected only slightly and decreases from 171 to 169. As a result, we would not expect PIRA's selection heuristics to be affected significantly by this change in the median number of statements.

[3]We use the built-in Linux tools `sort`, `uniq`, and `wc` to arrive at the number of unique missing edges.

Benchmark	S_t	$\hat{S}_t$	Change	S_m	$\hat{S}_m$	Change
LULESH	3,267	3,267	0	27	27	0
AD LULESH	9,959	10,259	300	28	28	0
eos-mbpt	29,904	29,952	48	19	17	−2
ISSM	2,661,552	2,690,438	28,886	171	169	−2

Table 4.3.: The number of statements reachable from `main` (S_t) and the median number of statements (S_m) across all reachable functions in the MetaCG before and after missing edges are inserted.

Benchmark	Filter	Stack	Global	Runtime Stack	T_R
LULESH	CG	19 [64.8]	0 [100]	1,816	1.01
	STD	32 [40.7]	0 [100]	2,624	1.01
AD LULESH	CG	72 [96.4]	2 [99.8]	13,816	1.01
	STD	615 [68.9]	5 [99.0]	32,429,228	1.07

Table 4.4.: TypeART filter and instrumentation statistics. Note: Stack and Global are the filtered count. Filter percentage (of the total) in brackets [%]. Runtime Stack are the total stack variables tracked at runtime (globals are tracked once per instrumentation). T_R is the relative runtime impact w.r.t. vanilla (no instrumentation) — less is better.

Call-Graph Filter

Table 4.4 shows the impact of the CG-based filter (CG) compared to the original filter (STD) for the LULESH and the AD LULESH target. Note that in our evaluation, we also marked functions coming from system includes as non-MPI reaching functions, which allowed for additional filtering.

Modern AD tools extensively facilitate template metaprogramming and inlining for efficiency. However, this causes TypeART to instrument significantly more allocations needed to compute the derivatives. In total, for the STD filter, we track 32 million stack variables at runtime, compared to the non-AD version with about 3,000 for the same benchmark configuration. With the new CG filter, we reduce the number of tracked variables by a factor of about $1.45\times$ and $2{,}350\times$ for the original and AD version, respectively.

```
 1  /* Translation Unit A */
 2  int foo() {
 3    double (*f)(Args);      // Declare function pointer
 4    getSolver(&f, config); // Obtain function pointer
 5    f();                   // Use function pointer
 6  }
 7
 8  /* Translation Unit B */
 9  int getSolver(double (**f)(Args), Cfg c) {
10    if (c) { getSolverWCfg(f, c); } // Forward pointer
11  }
12
13  /* Translation Unit C */
14  int getSolverWCfg(double (**f)(Args), Cfg c) {
15    if (c.ONE) { *f = solverOne; }  // Set function pointer
16    if (c.TWO) { *f = solverTwo; }  // Set function pointer
17  }
```

Listing 4.3: Missed function pointer in ISSM, see line 5.

4.4. Discussion

In our experiments, we found that CGCollector can handle many parts of the *C/C*++ applications tested. The three most challenging constructs for CGCollector are constructors/destructors, heavily templated libraries and function pointers, which account for approximately 57%, 26%, and 7% of the missed edges, respectively. Most of the challenging constructors and destructors in our experiments originate from standard library data structures, i.e., heavily templated code. Moreover, according to the Itanium application binary interface[4], constructors and destructors can have multiple different mangled names, e.g., depending on whether they construct the full object or only a base class. Interestingly, processing the Clang AST, some constructors and destructors in question do not appear explicitly in the tree. This is particularly true for specific instances of templated classes.

In eos-mbpt, the large number of total functions and the relatively small number of reachable functions are due to large arrays of function pointers in the code base. While these arrays are included in the main file, they are not used in our configuration and no paths exist due to various preprocessor macros.

In AD LULESH some missed function pointers are due to the nature of AD reverse-mode computations. For these computations, the AD-library stores function pointers to a so-

[4]available at `https://itanium-cxx-abi.github.io/cxx-abi/abi.html`, accessed July 2021

```
ISSM::funcA
 |--PetSc::funcP
 |   |--MPI_Send
 |--ISSM::funcB
```

```
ISSM::funcA
 |--MPI_Send
 |--ISSM::funcB
```

Figure 4.6.: Left: actual call hierarchy present in the code. Right: call hierarchy recorded by Score-P.

called tape during the regular — or primal — execution of the target program. During the reverse computation, the function pointers are then used to execute the respective calculations at the correct, reversed, point in time. The other major contributor in the AD LULESH case are, again, constructors and destructors.

The most significant missed edge in ISSM comes from a missed function pointer that is used to select the specific solver. The problematic code construct is given in Listing 4.3, i.e., a function pointer is passed as `OUT` parameter through multiple calls in different TUs. Currently, CGCollector does not consider function `OUT` parameters. However, it will be addressed in future releases by computing the respective `OUT` parameters' target sets and using the metadata facility to annotate this information accordingly. The information will be subsequently used in CGMerge to add the corresponding edges to the graph.

The others are standard template library constructors/destructors and the library PETSc [10] functions calling into MPI. In our scenario, PETSc is neither analyzed at compile time by CGCollector, nor is it instrumented by Score-P. Since it is not instrumented, its internals are also not recorded at runtime by Score-P, hence, CGValidate reports a few false positives. In these cases, it seems as if an edge from an ISSM function to an MPI function is missing albeit actually the call is from PETSc to MPI, *cf.* Figure 4.6.

5. PIRA: Performance Instrumentation Refinement Automation

The chapter is based on the following publications and contains verbatim excerpts of parts which were contributed by the thesis author. It uses PIRA 0.3.4[1] for all experiments, unless stated explicitly otherwise.

> Arzt, Peter and Fischler, Yannic and **Lehr, Jan-Patrick** and Bischof, Christian. 2021. *Automatic low-overhead load-imbalance detection in MPI applications* [9]

> **Lehr, Jan-Patrick** and Calotoiu, Alexandru and Bischof, Christian and Wolf, Felix. 2019. *Automatic Instrumentation Refinement for Empirical Performance Modeling* [73]

> **Lehr, Jan-Patrick** and Hück, Alexander and Bischof, Christian. 2018. *PIRA: Performance Instrumentation Refinement Automation* [74]

An important part of the performance engineering workflow is the iterative process of measuring a target application's performance, the analysis of the data and the subsequent refinement of the measurement [55]. Depending on the analysis question at hand, the metrics used to identify relevant regions differ. For an initial hot-spot analysis, the general runtime distribution is valuable. For a scaling analysis, it is of more importance how the different functions in the target program behave with increasing input data. When investigating the load balancing behavior, it is of interest to identify the functions that require considerably different amounts of time to execute across the MPI ranks. To that end, most performance measurement and performance analysis tools, however, take care of only a single measurement within this process. Hence, the user is left with manual

[1]Available at `https://github.com/tudasc/pira`.

work in between to interpret the data and adjust the measurement. This is particularly undesirable for repetitive tasks that are mechanical in nature, such as the creation of instrumentation filter lists for Score-P.

In this chapter, we present PIRA — the Performance Instrumentation Refinement Automation — to address this impediment. PIRA performs whole-program analysis using MetaCG to construct a low-overhead instrumentation and refines this starting-point instrumentation iteratively using profile information obtained from the previous run. First, we give an overview of its conceptual and software components in Section 5.2, before we present its workflow and the components in more detail. We specifically explain the different heuristics used by PIRA to construct low-overhead instrumentation in Section 5.3. The description is followed by an evaluation of the different heuristics on both sequential and MPI-parallel applications in Section 5.4. Thereafter, we discuss the results obtained.

5.1. Approach

As introduced in Section 2.2, an analyst commonly starts with an overview measurement, and subsequently refines the measurements. For many instrumentation tools, e.g., Score-P, these refinements are performed by creating filter lists, a so-called *instrumentation configuration (IC)*. The filter is either applied at runtime (dynamic filtering) or at compile time (static filtering), with the latter typically resulting in less overhead, but the need to recompile after every change to the IC. Currently, this step is (mostly) performed manually. For example, the analyst performs an initial measurement and, based on a function's call-count, adds the function to the IC, reruns the measurement and inspects both the runtime overhead introduced and whether the picture obtained from the target application is sufficiently accurate. This is time-consuming and tedious given the complexity and diversity of state-of-the-art solver packages.

To advance the current state and assist the analyst in finding a suitable instrumentation, PIRA automates initial refinement of the IC by performing the steps

1. Build a vanilla, i.e., an uninstrumented, version of the target application.
2. Perform baseline measurements to subsequently compute runtime overhead.
3. Create an initial IC using static, i.e., source-code, selection heuristics.
4. Perform a performance measurement to generate profile information.
5. Create a subsequent IC by analyzing the dynamic, i.e., profiling data to filter-out irrelevant functions, and the static source-code heuristics to expand the IC towards interesting regions.

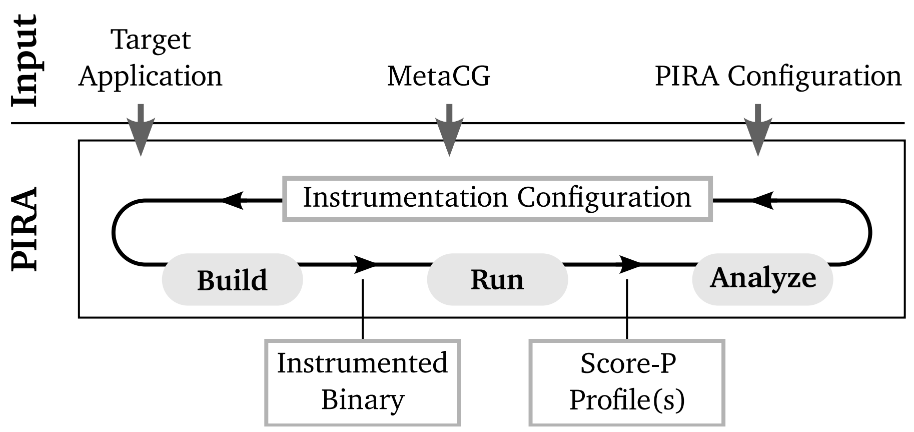


Figure 5.1.: PIRA constructs and analyzes a whole-program call-graph (CG) to create instrumentation filters. Its approach follows a build–run–analyze cycle: The initial analysis uses source-code features only, whereas subsequent analyses use dynamic, i.e., profile information, in addition.

The last two steps are iterated for a user-defined number of iterations. PIRA implements this iterative refinement of the IC in its *Build–Run–Analyze* cycle, *cf*. Figure 5.1.

Build — Run — Analyze Cycle The Build–Run–Analyze cycle is the iterative process to determine a suitable instrumentation for a specific performance analysis question. The process starts with an initial IC to instrument the target for an overview measurement. Thereafter, the instrumented target is executed to generate the runtime profile. This profile is subsequently analyzed and *relevant* functions are maintained in the IC, while irrelevant functions are removed from the IC. The profile analysis applies different heuristics which inspect the runtime information gathered for metrics relevant to them.

The initial IC is constructed statically, and referred to as 0-th *PIRA iteration*. Its subsequent iterations use both static and dynamic information to construct the next IC. Within each iteration multiple *repetitions* may be performed, i.e., multiple executions of the same target binary, to increase the profile information fidelity. This is increasingly important for the Performance Model Heuristics explained in greater detail in Section 5.3.3.

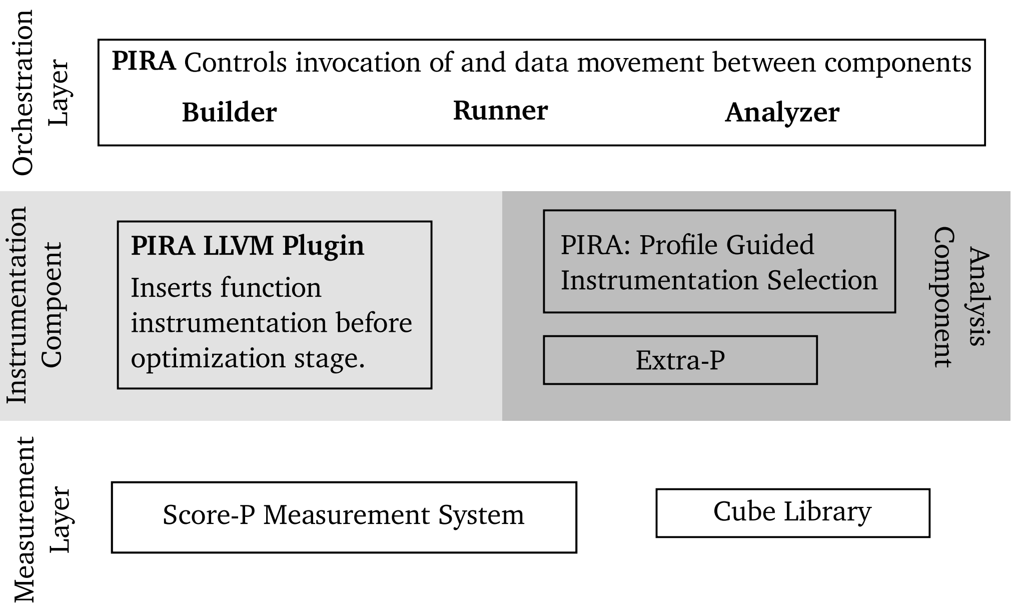

Orchestration Layer
PIRA Controls invocation of and data movement between components
Builder
Runner
Analyzer
Instrumentation Compoent
PIRA LLVM Plugin
Inserts function instrumentation before optimization stage.
PIRA: Profile Guided Instrumentation Selection
Extra-P
Analysis Component
Measurement Layer
Score-P Measurement System
Cube Library

Figure 5.2.: PIRA is split into the components for orchestration, instrumentation, analysis, and measurement. The measurement layer relies on the Score-P libraries with the Cube backend to capture and record the application's profile. The instrumentation layer provides an LLVM plugin that inserts a function instrumentation before the optimization stage. In the analysis component, PIRA provides the PGIS tool. PGIS uses Extra-P for its performance-model heuristics. At the high level, PIRA orchestrates the different components and ensures that the relevant data is moved between components.

5.2. Software Architecture

PIRA's high-level software architecture closely resembles the Build–Run–Analyze cycle and encompasses the most important parts in the respective abstractions `Builder`, `Runner`, and `Analyzer`. The components encapsulate functionality needed within the respective step of the Build–Run–Analyze cycle. Since different target applications can demand very different handling, e.g., how it is built, the different components invoke user-provided functions that implement the specific details. These functions are referred to as *functors*. The main `PIRA` component orchestrates the previously mentioned components. An overview of the architecture is shown in Figure 5.2.

Analyzer The orchestration-layer `Analyzer` component provides the interface that PIRA invokes to construct the next IC. However, this layer does not include any particular analysis, but generates the required invocations to the Analysis Engine. As part of PIRA, we develop the Profile Guided Instrumentation Selection (PGIS) tool and include it as the default analysis engine. PGIS and its different heuristics are explained in more detail in Section 5.3. The separation between orchestration code and actual analysis implementation enables a subsequent exchange of the engine, for example, should we (or someone else) want to implement and use an analysis engine based on Hatchet [13]. This loose coupling was deliberately chosen to construct PIRA around a "has a" component-based architecture, as compared to a strongly-connected and integrated software system.

Runner The target application gets executed by the orchestration-layer `Runner` component. It processes the argument configuration to create the respective invocation of the target application and manages that the resulting profiles are moved to the specified locations. These steps are particularly relevant when using the Performance Model Heuristic that is explained in Section 5.3.3. The `Runner` also performs time measurements of the target's execution, which are used to compute runtime overheads and allow the analyst an initial assessment of the measurement impact.

Builder The orchestration-layer `Builder` distinguishes between instrumented builds and uninstrumented builds to invoke the corresponding user-provided build functor. Moreover, for instrumented builds, it uses the IC generated and automatically creates the MPI filter. Therefore, it relies on the wrap tool [34] that automatically generates *C* code for the underlying MPI library. This means that, the `Builder` uses the IC to generate wrappers for all MPI functions that should not be measured. These wrappers intercept the MPI call and, instead of going through the Score-P measurement system, immediately call into the MPI implementation. This is particularly relevant for target codes that use asynchronous communication and many calls to, e.g., `MPI_Iprobe`, as this would result in large runtime overheads. Moreover, these overheads can prohibitively influence the recorded characteristics of the target application, i.e., obscuring existing load imbalances. Certain functions, however, are never filtered, e.g., `MPI_Init` or `MPI_Finalize`, as otherwise the Score-P measurement fails.

To finally build the target application, the `Builder` uses the generated IC and passes it to the build-functor. The user-provided functor can then use the arguments to incorporate them into the required build commands. PIRA leaves the final invocation to the user as build systems and their configuration can significantly differ between projects.

5.3. Profile-Guided Instrumentation Selection

PGIS is the default Analysis Engine for PIRA and implements the function selection to construct ICs. Therefore, it implements strategies for initial, i.e., statically determined, ICs and strategies to refine existing ICs using both static and dynamic information. The different selection heuristics are implemented as so-called `EstimatorPhases`. These phases are processing the target application's MetaCG (see Chapter 4), and, potentially, profile information already generated. For most heuristics, taking into account whole-program information via MetaCG is essential. Moreover, the MetaCG's annotation capability allows passing information between different phases, across PIRA iterations, or persisting the information. An IC is output in the Score-P filter format and allows an analyst familiar with Score-P to use it as a starting point for further manual changes and filtering. This enables a more seamless transition from initial assessment to focus measurement.

The initial selection strategy starts from an empty selection and evaluates the heuristic's criterion for every function node in the MetaCG. When performing the refinement, PGIS follows a *filter-and-expand* strategy. This means that it, first, checks and evaluates profile data for a node to determine if the node should be kept with the IC or if it is deemed irrelevant and should be filtered. In a second step, previously uninstrumented child nodes are evaluated whether they should be included in the IC. To decide if nodes should be filtered or included different selection heuristics exist. The subsequent sections present the different selection heuristics in detail, before we evaluate PIRA using the different heuristics in Section 5.4.

5.3.1. Statistical Statement Aggregation Heuristic

The Statistical Statement Aggregation is the initial heuristic applied in most PIRA configurations. It is used to determine the initial IC, when profile information is not yet available and is based on the statement aggregation scheme proposed by Iwainsky and Bischof in [56]. The scheme uses the whole-program CG and computes an aggregated number of statements per function. It is implemented as a depth-first traversal that maintains a visited set to detect cycles. When it detects a cycle, it considers this path as complete and commits the inclusive number of statements to the MetaCG node as meta information.

In addition to the originally proposed technique that used a fixed threshold for filtering, PGIS computes statistical measures of the statement aggregation scheme on the MetaCG. More specifically, it computes the maximum, minimum, and median value of the aggregated number of statements. Table 5.1 shows the different values for the number of statements that PIRA determined from the SPEC CPU codes. The varying complexity of the target codes clearly shows that a single threshold value, as used in the original work,

Benchmark	#Functions	#Reachable	Aggregated Statements		
			Max	Median	Min
403.gcc	— n/a —				
429.mcf	87	37	760	38	0
433.milc	301	219	7,246	86	0
444.namd	1,046	269	6,968	43	0
447.dealII	36,412	11,322	553,919	656	−19
450.soplex	3,265	1,415	38,131	231	−1
453.povray	— n/a —				
456.hmmer	649	291	9,703	104	−1
458.sjeng	208	161	13,188	127	−1
462.libquantum	173	99	4,843	161	−3
464.h264ref	732	612	94,452	993	−2
470.lbm	74	35	350	27	0
473.astar	532	202	2,351	57	0
482.sphinx3	517	351	8,746	96	−3

Table 5.1.: The number of functions and the number of reachable functions together with statistical measures for the aggregated statements for the *C*/*C*++ subset of SPEC CPU 2006. For 403.gcc and 453.povray the CGCollector did not terminate within reasonable time, due to its limitation in resolving function pointers.

is insufficient to perform well across a large set of applications. Note that the negative number of aggregated statements is a result of an implementation detail within PGIS.

For the static instrumentation selection, PGIS, by default, uses the median value as the instrumentation threshold, i.e., functions that exceed this value are added to the IC. Figure 5.3 shows an example for the Statistical Statement Aggregation Heuristic on a simple graph. The original CG in (a) is used to compute the aggregated number of statements per function, shown in (b). Thereafter, the threshold value is computed using the median across all aggregated number of statements (47), and functions exceeding the threshold are added to the IC, while the remaining functions are pruned, as shown in (c). Using the aggregated number of statements is to some extent similar to a compiler's heuristics which functions to inline. An important difference is that PGIS computes the metric across the whole-program CG instead of the TU-local functions.

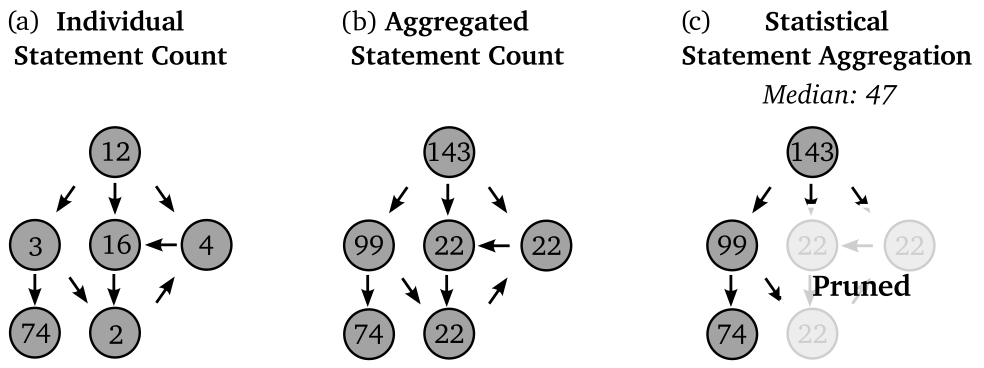

(a) Individual Statement Count
(b) Aggregated Statement Count
(c) Statistical Statement Aggregation
Median: 47
12
3
16
4
74
2
143
99
22
22
74
22
143
99
22
22
Pruned
74
22

Figure 5.3.: The example shows a CG with circles denoting functions and arrows edges. The node labels in (a) are the exclusive number of statements for the respective function. In (b) the labels show the aggregated number of statements for the respective function node. Finally, (c) shows the median value for the example and the nodes not included in the IC as *Pruned* nodes.

5.3.2. Runtime Heuristic

The Runtime Heuristics, or hot-spot heuristics, is the approach that allows PIRA to automatically refine the instrumentation towards functions in the target application that require the largest share of runtime. It follows the filter-and-expand strategy and can be applied as soon as profile information, such as runtime, is available, i.e., starting from iteration one. The runtime considered in the heuristic is the *inclusive runtime*, which means that it considers the sum of a function and its children.

When evaluating the heuristic, PGIS, first, computes a runtime threshold based on the total runtime of the target application. This threshold is used to filter short-running functions that do not contribute significantly to the overall runtime. Figure 5.4 shows an example application of the approach. Hence, for each node in the MetaCG that has profile information attached, PGIS evaluates whether the node crosses the runtime threshold. Should the node not exceed the runtime threshold, it is considered insignificant and not added to the IC. Second, for each node that crosses the threshold, PGIS evaluates whether is has descendant nodes that have already been profiled. Should profile information be available for one or more nodes, PGIS determines the node with the largest runtime and continues its evaluation. Third, arriving at a node that does not have descendants with profile information attached, it applies the Statistical Statement Aggregation with a locally-determined threshold to expand the instrumentation.

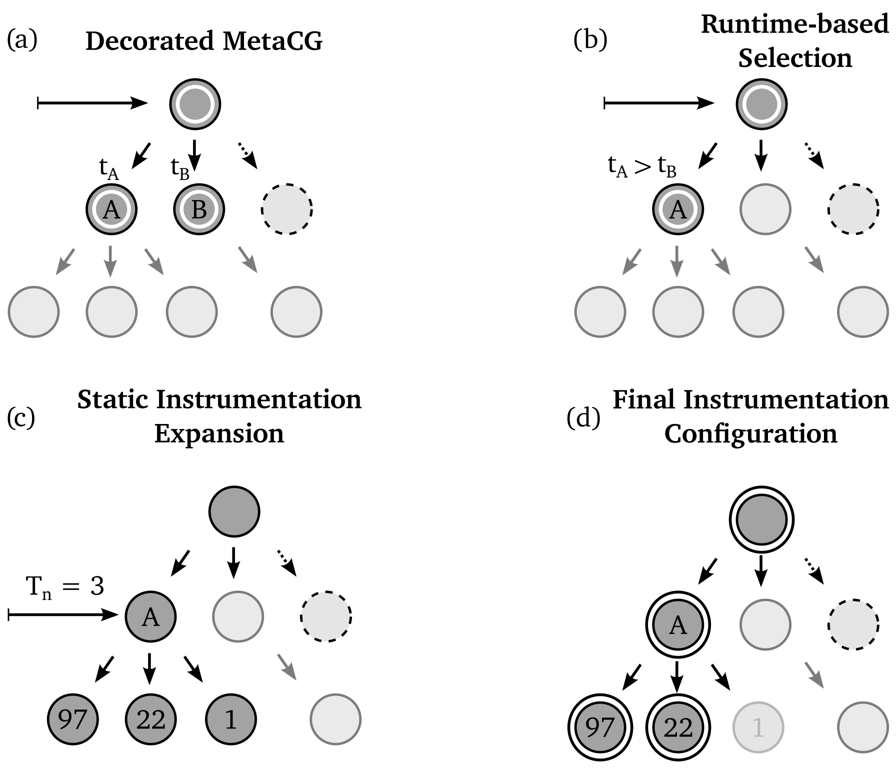


Figure 5.4.: The MetaCG nodes decorated with a white inner circle in (a) have runtime information t_A and t_B attached. PIRA determines the principal node A based on the largest runtime, *cf.* (b). Thereafter, in (c), PIRA applies static instrumentation expansion using a locally determined threshold T_n for the number of aggregated statements. The final IC contains all nodes marked with an additional black circle and shown in (d).

5.3.3. Performance Model Heuristic

While the Runtime Heuristic is useful when analyzing a target software for its current hot-spots, evaluating a software w.r.t. its scaling behavior requires a different approach. To that end, we integrate the possibility to use Extra-P [23] to automatically construct empirical performance models for the functions profiled. These performance models are mathematical functions describing the behavior of a target program's implementation in user-given parameters, e.g., number of MPI processes. This allows PIRA to evaluate the performance model at extrapolated points to compute presumable runtimes of the implementation for these points. Subsequently, PIRA uses this presumable runtime to perform threshold filtering and refine the instrumentation towards significant functions. For a better understanding, we first, briefly, introduce Extra-P and its performance models, before explaining the Performance Model Heuristics in detail.

Extra-P

Extra-P [23] is a state-of-the-art tool that can leverage fine-grained measurements, e.g., provided by Score-P, to generate performance models for multiple parameters [22]. It has been successfully used to detect scalability bottlenecks and evaluate the performance of many libraries and scientific applications [23], [58], [121].

The core concept relies on the fact that the complexity of algorithms implemented in both sequential and parallel applications with respect to most relevant configuration parameters is most commonly polynomial, logarithmic, or some combination thereof. This has led to the introduction of the performance model normal form (PMNF), which expresses the effect of a number of parameters x_i on a metric as a sum of terms consisting of products of polynomial and logarithmic expressions in the parameters x_i. The expression is formalized in Equation 5.1.

$$f(x_1, \ldots, x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \cdot log_2^{j_{kl}}(x_l) \tag{5.1}$$

Given a set of measurements, the performance models are identified in an iterative process. Extra-P first models the effects of each parameter separately, and then tests all possible combinations of the selected single-parameter models to determine the multi-parameter model that fits the measurements best.

An important assumption of the modeling approach is that there is one behavior to the modeled application across the entire parameter range. Should this not be true, for example due to the MPI collective communication algorithm changing with increasing number of processes, then the resulting model may be misleading. A method has been developed that can automatically detect such an occurrence and if necessary suggest additional measurements [50].

Heuristic Implementation

The general approach of the Performance Model Heuristic is similar to the Runtime Heuristic. First, it filters the current set of functions profiled based on the performance models using their extrapolated values and a comparison against a certain threshold, the filter step. Second, it evaluates child nodes of the functions kept for measurements whether they exceed the Statement Aggregation threshold to expand the instrumentation, the expand step. Lastly, it optionally traverses the graph from each selected node upwards until it reaches `main`. While the latter step is optional it can be relevant for an analyst to know the context of individual functions and their behavior within different contexts. It may, however, introduce additional runtime overhead. An example of the Performance Model Heuristic is given in Figure 5.5.

To apply empirical performance modeling, the user needs to provide multiple input data sets that vary the values for the input parameter(s) that should be modeled. Currently, PIRA's implementation supports up to three modeling parameters. It generates the target invocations with the different input parameters, executes the target application, and, stores the generated profiles following the particular Extra-P naming scheme. Since the currently employed Extra-P modeling technique requires all combinations of parameters, together with at least five repetitions, modeling a three parameter model requires at least 125 executions of the target application.

For PIRA's filtering, PGIS computes an extrapolated value for the different model parameters based on the user-provided input values. Therefore, it computes the extrapolated values by (1) averaging the distance between the user-provided input values for the modeling, and, (2) adding this distance to the largest user-provided value. This means that PGIS does not perform extreme extrapolation, for which the error may be significant. However, it also means that it may not identify functions that could become problematic when using much larger data sets then provided to PIRA.

5.3.4. Load Imbalance Heuristic

Load imbalance, i.e., the uneven distribution of work across compute units, is a widespread source for inefficiencies in parallel applications and different approaches have been proposed to identify such regions [17], [29], [126]. In particular, the work in the Scalasca framework [17] is based on *replaying* the target program's execution traces. This facilitates a detailed analysis of its communication patterns and allows for precise classification of

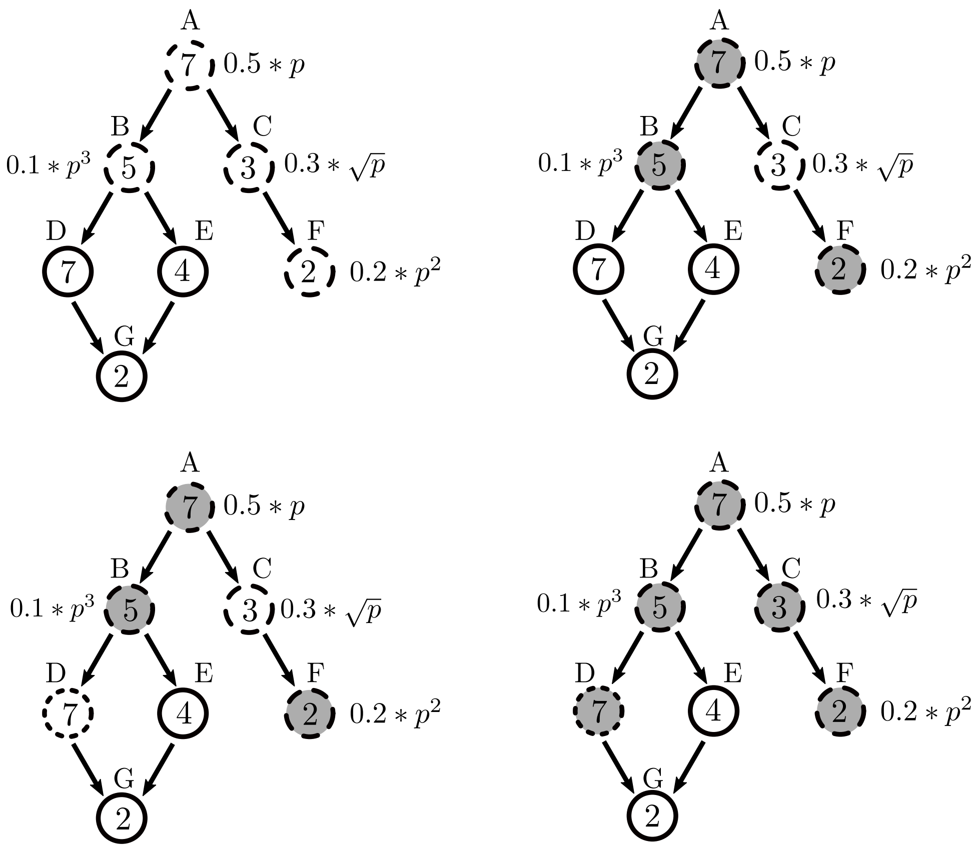
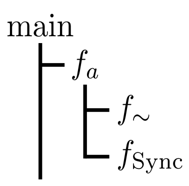
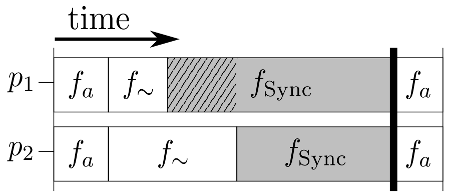


Figure 5.5.: Filter and expand: First, the filter step is applied, marking nodes A, B, and F. Thereafter, starting from those nodes, a static source-code criterion is evaluated on all its child nodes. For node D, the number of statements contained in the function satisfies the selection criterion, hence, D is added to the instrumentation. Node C is added to the instrumentation as it is on a call path from F to the root node.



(a) Example call graph. The `main` function calls a function f_a which invokes an imbalanced function $f_\sim$ and a synchronization routine f_{Sync}, e.g., `MPI_Barrier`.



(b) Example trace of f_a and its children. p_1 reaches f_{Sync} (gray area) first and is forced to wait for p_2. The induced idle time is represented by the hatched area. After f_{Sync} the execution continues in a balanced fashion.

Figure 5.6.: Minimal example for load imbalance with two parallel MPI-processes p_1, p_2. Note that, due to the synchronization in f_{Sync}, f_a is itself balanced although a load imbalance is present in $f_\sim$.

wait states or the exact quantification of negative effects. Since Scalasca relies on Score-P traces, which can be costly to generate, it is of interest to obtain Score-P measurements that already contain relevant candidate regions.

PIRA implements a filter-and-expand strategy to automatically detect such inefficiencies in MPI-parallel applications, called PIRA-LIDe. Therefore, the MetaCG profile information is extended to include the notion of a *location*, e.g., an MPI rank. This information can be conveniently read from the Score-P profiles generated.

Typically, load imbalances are investigated using trace data, due to the precise timing information available. Our approach is different in the way that it relies on an application's profile and considers differences in execution times per function across all locations, e.g., MPI processes. An example for a load imbalance — and the different execution times across functions and MPI locations — is given in Figure 5.6. In the heuristic PGIS uses both types of runtime of a function, i.e., inclusive and exclusive runtime. Inclusive runtime is the runtime of a function *including* all its callees, while exclusive runtime is the runtime of only the particular function *excluding* all its callees. The heuristic uses different thresholds and parameters to determine heuristic cut-offs during the refinement. It uses a runtime relevance threshold to decide which functions to keep, and an imbalance threshold to evaluate if a function is imbalanced. For the latter, we use the Imbalance Percentage (I), *cf.* [29] and defined in Equation 5.2, as the default.

$$I = \frac{T_{\text{max}} - T_{\text{avg}}}{T_{\text{max}}} \times \frac{n}{n-1} \tag{5.2}$$

In addition, the static selection is determined by the two parameters child constant threshold and child fraction. These parameters determine lower bounds on the size of a function either statically, or relative to the actual application size. In our evaluation, we list the particular values that PIRA uses as the default setting. However, these parameters can be adjusted by the user externally.

The load imbalance detection works as follows, and an example application is presented in Figure 5.7. Initially, only the `main` function is instrumented to have a reproducible starting point. The next steps are then iterated for a user-provided number of iterations. The generated IC is applied, and the resulting target is executed to generate the profile information. The profile information is attached to the MetaCG nodes, and the heuristic is evaluated. For every `node` with attached profiling information, PGIS, first, applies an inclusive-runtime threshold filtering that removes short running functions from the IC. If this threshold is not exceeded, the function is marked as irrelevant and excluded from measurement in all subsequent iterations. Since the inclusive runtime includes the runtime of all callees, no important regions are missed as a result of this filtering. Second, it evaluates the imbalance metric, e.g., the Imbalance Percentage, for `node`s that have not been filtered in the previous step. The imbalance metric quantifies the degree of imbalance that occurred within the specific subtree rooted at `node`. Should the `node` not show a sufficient degree of imbalance the `node` itself is removed from the IC. To expand the instrumentation, all children of `node` are then assessed for the Statement Aggregation heuristic, to descend deeper into the CG. As soon as a `node` shows a sufficiently large value for the imbalance metric, it is reported as imbalanced.

Once an imbalanced region is identified, a differentiation along different calling contexts is of interest to an analyst. Hence, PGIS automatically determines all paths from `main` to the identified load imbalance and adds the functions to the IC. This allows a subsequent application of tools, such as Scalasca [36], for an automated analysis of wait states and similar potential root-causes.

5.4. Evaluation

In this section, we evaluate the different heuristics w.r.t. their capability in the respective use case (hot-spot detection, scalability-behavior determination, and load-imbalance detection) and the respective runtime overhead. We run the experiments on exclusively-used compute nodes of the Lichtenberg II HPC system at TU Darmstadt. Each node is equipped with two Intel Xeon Cascade Lake Platinum 9242 Processor with 48 cores and 96 threads each, and 384GB of main memory. The exact number of experiment repetitions is left to the specific subsequent sections, as between experiments the setup is slightly different. Also, the different heuristics are evaluated on different benchmarks due to the availability of input data sets for the various applications.

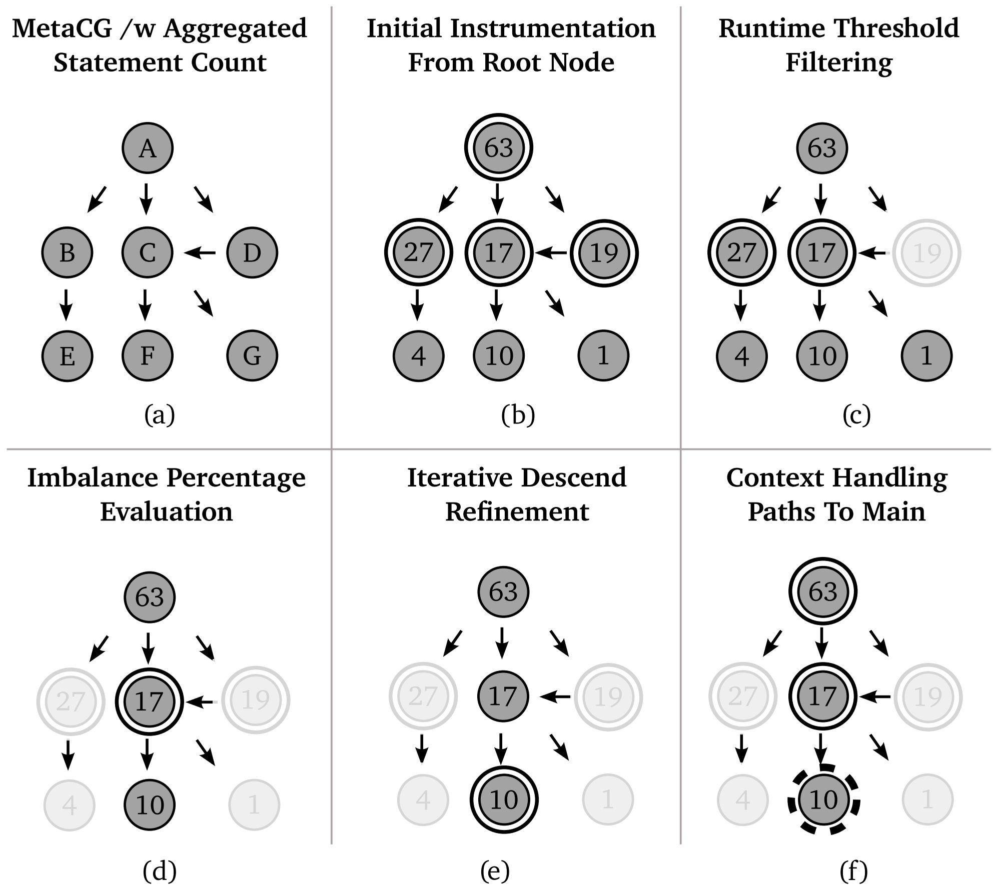

MetaCG /w Aggregated Statement Count
A
B
C
D
E
F
G
(a)
Initial Instrumentation From Root Node
63
27
17
19
4
10
1
(b)
Runtime Threshold Filtering
63
27
17
19
4
10
1
(c)
Imbalance Percentage Evaluation
63
27
17
19
4
10
1
(d)
Iterative Descend Refinement
63
27
17
19
4
10
1
(e)
Context Handling Paths To Main
63
27
17
19
4
10
1
(f)

Figure 5.7.: Example application of PIRA's Load Imbalance Detection Heuristics. Instrumented nodes are marked with an extra circle, and nodes filtered from instrumentation are shown discolored. The initial MetaCG is shown in (a). First, `A` (typically `main`) and its children that exceed the static statement threshold are instrumented as shown in (b). In (c), the node D is filtered due to not reaching the runtime threshold. The CG node B in (d) is balanced, thus filtered, and its child node does not reach the static statement threshold, excluding it from measurement. In (e), node C is itself balanced, but the tree rooted at this node is imbalanced and the instrumentation is refined to its children. Finally, in (f), the imbalanced node F (indicated by the dotted circle) is identified and reported. Optionally, its context is instrumented for subsequent manual inspection of the profile.

In general, we are interested in the runtime overhead imposed by PIRA per iteration, as well as the total runtime. The latter is the time PIRA requires for all its iterations and potentially repetitions, hence, it is the time an analyst would need to wait to obtain a result. PIRA reports the wall-clock runtime of its vanilla baseline measurements as well as each measurement run to the user, and we compute the runtime overhead as $\frac{t_{\text{vanilla}}}{t_{\text{measurement}}}$. Additionally, it is of interest whether PIRA can identify the application's functions relevant for the specific metric, i.e., hot spots, scalability, and load imbalance. Finally, we investigate the number of functions selected compared to the number of functions actually measured at runtime, and, how many regions are contained in the resulting Score-P profiles. A region in Score-P is a function within its calling context, i.e., `foo` called within `bar` is a different region as `foo` called within `main`.

We manually validate PIRA's results w.r.t. the identified regions. Therefore, we compare the results to those identified with other profilers, e.g., Intel vtune, or manually perform an initial performance analysis steps. When not mentioned otherwise, PIRA was able to determine the relevant functions.

Runtime Heuristic

For the Runtime Heuristic, we apply PIRA to the computational fluid dynamics solver SU2 [32] and the SPEC CPU *C/C*++ subset that we used before. To reduce the scattering of timing results, we instruct the Slurm workload manager to disable dynamic frequency scaling of the processor. PIRA executes four iterations, which means that it conducts the vanilla runs first, and then performs an initial, static, instrumentation with three subsequent, dynamic, refinement steps. The runtime filter threshold is set to $0.5 \times t_{\text{main}}$, and $0.25 \times t_{\text{main}}$, where t_{main} is the runtime of the `main` function, respectively. We refer to these settings as PIRA$_H$ and PIRA$_Q$, respectively. In the first case PIRA aggressively filters functions from the measurement, whereas in the latter case, it filters fewer functions. We set PIRA to conduct five repetitions and report the runtime median together with the standard deviation of the five repetitions. Our benchmarks use the reference data set for the respective SPEC CPU target codes.

We construct the MetaCG for each code with CGCollector, and patch potentially missing edges for the data set using CGValidate. CGCollector fails to obtain the MetaCG for 403.gcc within the set time limit of $24\,\text{h}$, hence, we cannot determine PIRA's ability to construct meaningful ICs. For 453.povray, CGCollector requires substantially long to construct the MetaCG, however, Score-P is unable to record a valid profile when using Clang 10. This is likely due to an issue with exceptional control flow that triggers an internal assertion in the Cube library w.r.t. call-path consistency.

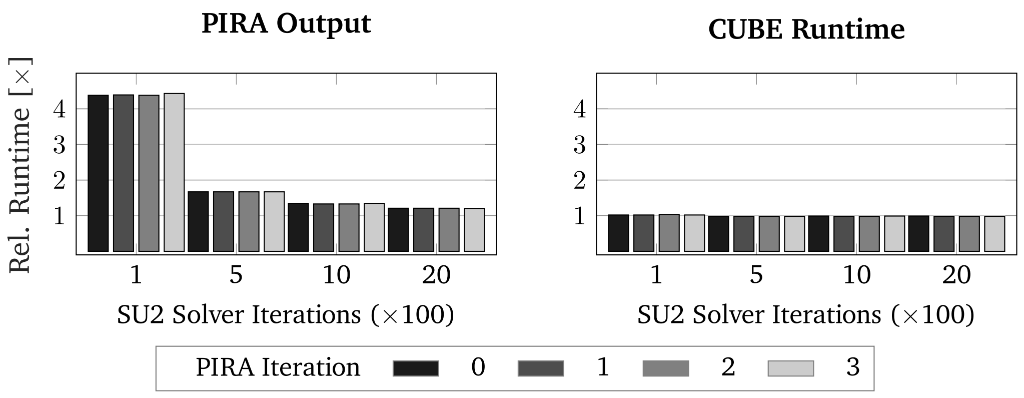


Figure 5.8.: The runtime overhead introduced by PIRA$_H$ over four PIRA iterations and different numbers of solver iterations within SU2: (a) as determined by PIRA, (b) using the runtime of the target as recorded within the Cube profile. The x-axis denotes the number of iterations for the SU2 solver. The y-axis denotes application runtime relative to vanilla runtime with 1 indicating runtime equal to vanilla. The vanilla runtimes for the different number of iterations in SU2 are $\approx$6 s, $\approx$29 s, $\approx$58 s, and $\approx$90 s.

SU2 For SU2, we see that, according to its output, PIRA introduces a slowdown of $\approx 4.38\times$ across the different iterations for the smallest number of iterations, *cf.* the left bar plot in Figure 5.8. The relative runtime overhead decreases with increasing number of iterations. When inspecting the runtime recorded for the application *within* the Cube profile, we find that it is actually much lower, *cf.* the right bar plot in Figure 5.8. We see that even for the smallest number of iterations the runtime overhead introduced is barely noticeable (between $\approx 0.99\times$ and $\approx 1.02\times$). For comparison, an unfiltered Score-P instrumentation results in $\approx 157.5\times$, $\approx 97.7\times$, $\approx 86.5\times$, and $79.1\times$ runtime overhead, and an inline-filtered Score-P instrumentation $\approx 9.5\times$, $\approx 5.7\times$, $\approx 5.2\times$, and $\approx 5.0\times$, across the different numbers of iterations. Across the different number of iterations in SU2, PIRA's final profile records only 10 different functions, which account for a total of 99% of the target's runtime. This suggests that the large relative overhead that PIRA reports can be attributed to either start-up or tear-down time spent by Score-P. Score-P inspects the target binary at start-up time for all function identifiers present in the binary, using `libbfd` for inspection. When we exclude externally defined functions, the SU2 binary contains $42{,}378$ different functions[2] that Score-P has to insert into its region-identifier table. This results in the considerable start-up time.

[2]According to `nm`.

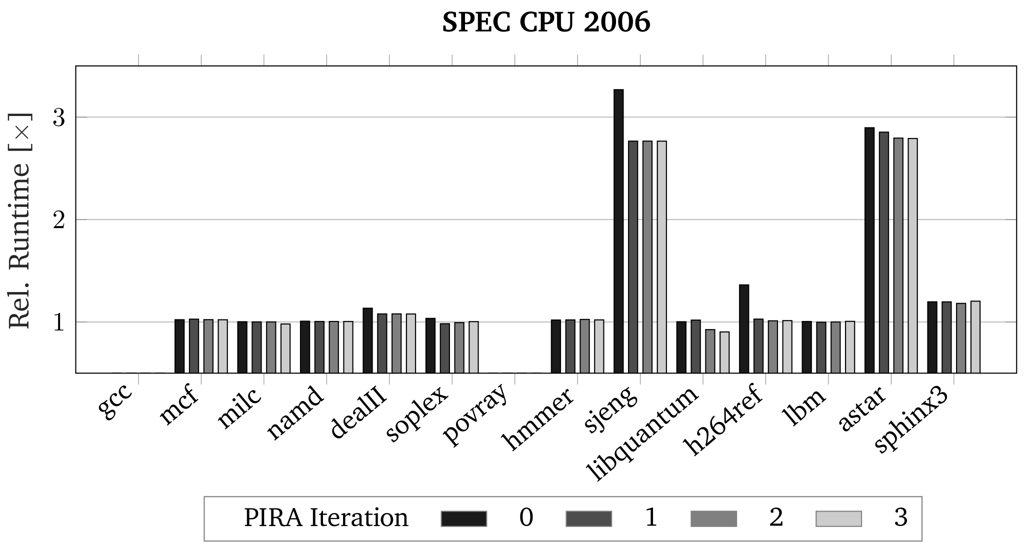


Figure 5.9.: The runtime overhead introduced across PIRA$_H$ iterations for the SPEC CPU 2006 benchmarks (x-axis). The y-axis denotes application runtime relative to vanilla runtime with 1 indicating runtime equal to vanilla.

SPEC CPU 2006 In Figure 5.9 we see the development of the runtime overhead for the SPEC CPU 2006 benchmarks over the course of four PIRA iterations. While the black bar reflects the runtime overhead introduced by the initial static selection heuristic, the remaining three show the development with PIRA's runtime filtering. In most cases, the static instrumentation selection already results in a low-overhead instrumentation. The relative overhead for these cases is between $0\,\%$ and $15\,\%$. 482.sphinx3 shows a slightly higher relative overhead between $16\,\%$ in the final iteration and $25\,\%$ in the initial static instrumentation. However, in the cases 458.sjeng, 464.h264ref, and 473.astar PIRA is unable to statically determine a low-overhead instrumentation. In the case of 464.h264ref, PIRA can reduce the influence of the measurement perturbation with its first Runtime Heuristics filtering. PIRA is unable to satisfactorily reduce the overhead for the cases 458.sjeng and 473.astar, with both showing runtime overheads of $\approx 3\times$. We also see that in none of the cases PIRA adds significant instrumentation in subsequent iterations.

Figure 5.10 shows the runtime overhead for the SPEC CPU 2006 benchmarks for PIRA$_Q$. While we see an uptick of runtime overhead in some benchmarks w.r.t. the overhead for PIRA$_H$, the measurement influence on the execution time still is acceptable in most cases. Both benchmarks that show significant runtime increase for PIRA$_H$ — 458.sjeng and 473.astar — also show a significant increase with PIRA$_Q$.

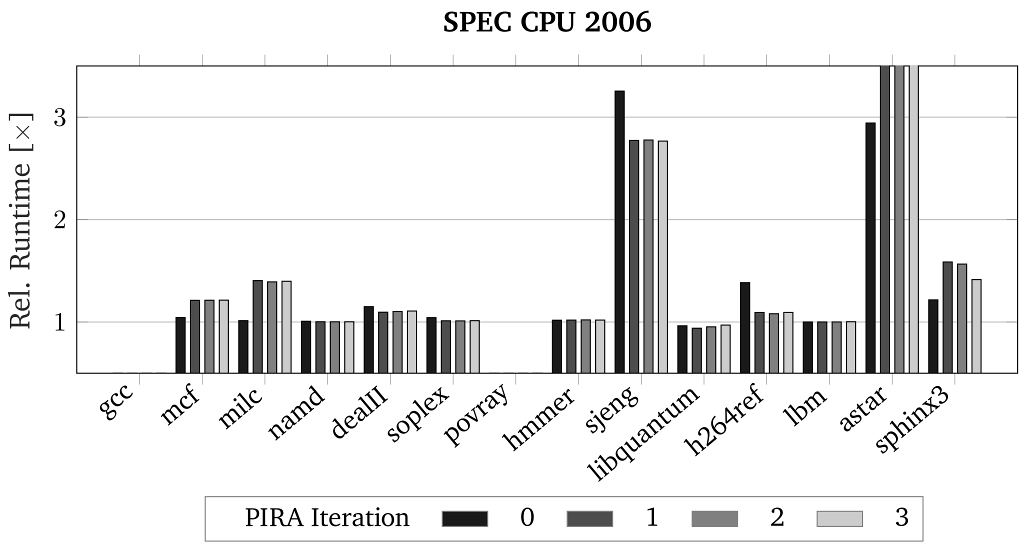


Figure 5.10.: The runtime overhead introduced across PIRA$_Q$ iterations for the SPEC CPU 2006 benchmarks (x-axis). The y-axis denotes application runtime relative to vanilla runtime with 1 indicating runtime equal to vanilla. The overhead of 473.astar is truncated, see Table 5.2.

Benchmark	**Score-P**		**PIRA$_H$**		**PIRA$_Q$**	
	w/ Filter	w/o Filter	Static	Dynamic	Static	Dynamic
403.gcc	13.26	14.42	— n/a —			
429.mcf	1.09	6.61	1.02	1.02	1.04	1.21
433.milc	2.68	2.70	1.00	0.98	1.01	1.40
444.namd	1.00	3.87	1.01	1.01	1.01	1.00
447.dealII	14.55	473.96	1.13	1.08	1.15	1.11
450.soplex	2.93	56.81	1.04	1.00	1.04	1.01
453.povray	— n/a —					
456.hmmer	1.31	1.46	1.02	1.02	1.02	1.02
458.sjeng	7.36	9.22	3.27	2.77	3.25	2.77
462.libquantum	1.35	1.57	1.00	0.90	0.96	1.07
464.h264ref	13.68	14.35	1.36	1.01	1.38	1.09
470.lbm	1.00	1.01	1.01	1.01	1.00	1.00
473.astar	5.40	25.62	2.90	2.79	2.94	7.12
482.sphinx3	2.43	2.87	1.20	1.20	1.22	1.41

Table 5.2.: The runtime overhead introduced by Score-P with and without inline filter for the SPEC CPU 2006 targets. We list the runtime overhead for the initial iteration (Static), and the final PIRA iteration (Dynamic).

With the lower runtime threshold, we see that PIRA maintains more functions in the IC initially and expands on these functions. For 429.mcf, 433.milc, 473.astar and 482.sphinx3, PIRA expands the instrumentation after the static IC, which explains the increase in runtime for the first dynamic iteration. In the case of 482.sphinx3, we also see a subsequent filtering, and a corresponding decrease of runtime overhead, while for 473.astar the runtime overhead is significantly increased for the final PIRA iteration.

For a comparison with an out-of-the-box Score-P system, we list the runtime overhead for measurements of Score-P profiles with and without inline filter in Table 5.2. The table also lists the runtime overhead for PIRA with both runtime thresholds. We see that PIRA can reduce the runtime overhead significantly with both runtime thresholds compared to standard Score-P measurements in almost all cases. For 429.mcf and 470.lbm, Score-P generates a similar runtime overhead as PIRA. In all other cases, the PIRA filtering reduces the runtime filtering significantly. Especially in the case of 447.dealII ($\approx 13.5\times$), 458.sjeng ($\approx 2.7\times$), and 464.h264ref ($\approx 13.5\times$) PIRA reduces the runtime overhead dramatically.

Benchmark	**Runtime (s)**				
	Vanilla	PIRA 0	PIRA 1	PIRA 2	PIRA 3
403.gcc	— n/a —				
429.mcf	202 ± 2	206 ± 3	208 ± 2	207 ± 2	207 ± 3
433.milc	419 ± 0	420 ± 1	420 ± 0	420 ± 0	411 ± 5
444.namd	254 ± 0	256 ± 0	255 ± 0	255 ± 0	255 ± 0
447.dealII	17 ± 0	20 ± 0	19 ± 0	19 ± 0	19 ± 0
450.soplex	77 ± 1	79 ± 1	75 ± 1	76 ± 1	77 ± 1
453.povray	— n/a —				
456.hmmer	167 ± 1	170 ± 1	170 ± 1	171 ± 0	170 ± 0
458.sjeng	337 ± 0	$1{,}100 \pm 9$	931 ± 2	931 ± 5	931 ± 4
462.libquantum	224 ± 19	225 ± 8	228 ± 14	208 ± 17	203 ± 16
464.h264ref	41 ± 0	56 ± 0	42 ± 0	42 ± 0	42 ± 0
470.lbm	202 ± 3	203 ± 0	202 ± 1	202 ± 1	204 ± 0
473.astar	95 ± 1	275 ± 4	271 ± 0	265 ± 0	265 ± 9
482.sphinx3	351 ± 3	420 ± 5	420 ± 8	415 ± 6	423 ± 5

Table 5.3.: Runtime in seconds for vanilla execution and four subsequent iterations with PIRA$_H$. The values denote the median and the standard deviation over five repetitions.

For 473.astar, PIRA reduces the runtime overhead compared to an unfiltered Score-P measurement. When inline filtering is enabled in Score-P, PIRA$_H$ reduces the runtime overhead compared to Score-P, while PIRA$_Q$ leads to a measurement with larger overhead.

Table 5.3 shows the obtained absolute timings for the SPEC CPU suite. In most cases, the runtime does not significantly change between PIRA iterations. For 464.h264ref, we see that the initial IC introduces significant slowdown, which is reduced in the subsequent PIRA iteration. In the case of 458.sjeng, 473.astar and 482.sphinx3 we also see a significant increase in runtime with the initial IC. While the influence is slightly reduced in the case for 458.sjeng, it still introduces a significant slowdown of the application. For both 473.astar and 482.sphinx3 PIRA is unable to significantly reduce the runtime overhead.

In addition to runtime and overhead, the number of functions selected for instrumentation and functions recorded as well as regions in the Cube profile are of interest. As Score-P stores call-path profiles, one function can occur multiple times within an application's profile as a different region. Hence, we list the number of distinct regions in the profile for the final PIRA iteration and Score-P executions with and without **`inline`**-filtering.

Benchmark	Functions			Score-P Regions		
	Selected	Profiled	Score-P	PIRA	w/ Filter	w/o Filter
403.gcc	— n/a —					
429.mcf	12/10	11/9	44	10	45	48
433.milc	56/7	30/7	246	9	294	305
444.namd	32/10	29/10	51	11	52	160
447.dealII	492/10	311/10	2,883	17	4,077	115,174
450.soplex	162/4	95/4	392	10	456	2,767
453.povray	— n/a —					
456.hmmer	69/6	13/5	74	10	81	93
458.sjeng	44/17	23/17	6,362	275	6,516	9,109
462.libquantum	30/6	24/6	267	14	290	337
464.h264ref	159/10	107/10	461	16	473	816
470.lbm	8/2	5/2	11	3	12	15
473.astar	34/5	34/5	99	6	102	513
482.sphinx3	78/8	52/7	156	13	471	704

Table 5.4.: The number of functions **selected** for instrumentation in the initial / final PIRA iteration and the number of functions actually **profiled** at runtime thereof for PIRA$_H$. The number of functions in a filtered Score-P profile for comparison. Score-P regions denote different locations within the call-path profile obtained in the last PIRA iteration, or with Score-P with and without filtering, respectively.

Table 5.4 shows the number of functions selected for instrumentation and the number of functions actually recorded during execution. We see that PIRA reduces the number of distinct functions significantly through its filtering mechanism. This is important, as an analyst can focus on the relevant functions present in the profile, and is not distracted by hundreds of small initialization or accessor methods. The reduction is particularly significant in the case of *C*++ benchmarks, such as 447.dealII. Note that the difference in the number of functions marked for instrumentation and actually profiled is a result of the static selection heuristics that PIRA applies. Moreover, PIRA reduces the number of regions significantly compared to both Score-P with and without filtering. Again, the reduction is particularly significant for *C*++ codes. Even with this large reduction of functions instrumented and regions recorded, some benchmarks exhibit significant overheads, e.g., 473.astar or 458.sjeng. In case of 473.astar, PIRA instruments a small function with a very high call count, and in case of 458.sjeng, PIRA instruments one of the recursively called analysis functions.

Hot Spot Detection We manually checked if PIRA refined the instrumentation towards the hot-spot regions in the target application. Therefore, we performed a few iterations of manual instrumentation refinement using profile analysis and applied the tool Intel vTune in its hot-spot detection configuration.

PIRA is able to refine the instrumentation automatically to the runtime hot-spot of the SU2 solver. Using its predefined thresholds and selection strategies, it arrives at the final instrumentation after the first dynamic filtering iteration, i.e., PIRA 1, and instruments only the kernel and the relevant call path to it. The SU2 solver seems to have a well-suited structure for PIRA's refinement, given its very dominating single kernel and call path.

For the SPEC CPU 2006 benchmark suite, both approaches are able to provide good overview measurements in 8 of 14 cases for PIRA$_H$ and 9 of 14 for PIRA$_Q$. The differences between PIRA$_H$ and PIRA$_Q$ are surprisingly small in most cases. Moreover, a reasonable overview measurement is in many cases already available after the first dynamic refinement step. We subsequently present the results for individual SPEC CPU 2006 benchmarks, and the final PIRA iteration.

For 429.mcf, PIRA$_H$ provides an overview of two main contributors to runtime, while PIRA$_Q$ instruments one function deeper in the CG. This function is a recursive sorting algorithm, which adds significant runtime overhead due to the high call-count. In 433.milc, both PIRA$_H$ and PIRA$_Q$ provide a good overview of where runtime is spent, with PIRA$_Q$ providing a little more detail at the cost of $\approx 40\%$ more runtime overhead. In the case of 444.namd, both settings result in the same final IC and show a good overview of the runtime distribution in the target. Similarly, for 447.deal, PIRA$_Q$ instruments two levels deeper in the CG, thus, provides marginally more insight on where runtime is spent. In the case of 450.soplex, PIRA$_H$ and PIRA$_Q$ refine the IC towards different regions in the target, with their combination providing a good overview on where runtime is spent. However, each profile individually leaves a considerable amount of runtime attributed to high-level functions. PIRA$_H$ and PIRA$_Q$ correctly identify the single hot spot in 456.hmmer. For the target 458.sjeng, both approaches refine the IC identically, which results in large amounts of overhead due to the recursive nature of the search/solve step. In 462.libquantum, PIRA$_Q$ results in a significantly better picture of the target's behavior, compared to PIRA$_H$, without introducing significantly more overhead. For 464.h264ref, both profiles are comparably informative w.r.t. runtime hot-spots of the target. Unsurprisingly, both approaches result in the same IC for 470.lbm and identify the target's hot spot. For 473.astar both approaches draw a similar picture of the application, but leave significant amounts of runtime unattributed, hence, we believe they did not meaningfully detect the application hot-spots. Finally, for 482.sphinx3, PIRA$_H$ gives a good overview of the application's runtime, while PIRA$_Q$ includes one more function that, given the high call count, adds significantly to the overall runtime overhead.

Model Heuristics

We apply PIRA with Model Heuristic to the applications 473.astar and SU2. Extra-P requires five different input data sets for a target, hence, for 473.astar, we vary the number of waypoints in the map to find, and the number of iterations for the SU2 solver. PIRA performs four iterations. Within each iteration, all five data sets are used, and the target is executed five times per data set. PGIS performs the optional call-path instrumentation, i.e., it instruments all paths to the `main` function from a function identified to be kept. We use both vanilla executions of the target and Score-P filtered measurements for comparison.

In previous work, *cf.* [73], we used PIRA with its Model Heuristics and applied it to the MILC solver su3_rmp, which is the MPI-parallel application, and, thus, different from 433.milc from the SPEC CPU 2006 version. Note that this version of PIRA did not include the automatic MPI filtering and experiments were performed on Intel Haswell-based Xeon E5-2680v3 with 64GB main memory. We present the results in the MILC paragraph.

In addition to the evaluation presented in this section, we use the Model Heuristic in Chapter 6 to automatically determine the target's kernels.

SU2 Table 5.5 lists the vanilla runtime required for all data sets and repetitions required for Extra-P modeling. The **total** PIRA time is about 12 times as large, and due to the multiple executions required for PIRA's dynamic filtering. This includes the time for PIRA to analyze the MetaCG and compile the target for every instrumentation adjustment. In

Benchmark	Vanilla	Score-P w/ Filter	Total PIRA	PIRA
SU2	1,378	7,152	19,005	1,865
473.astar	688	4,927	3,439	857

Table 5.5.: Accumulated runtime (seconds) to execute all measurements required for the Extra-P modeling. Vanilla is for reference only. *Score-P w/ filter* is Score-P with inline filter, *Total PIRA* is accumulated over all PIRA iterations, *PIRA* is the time of the final PIRA IC.

Function	Extra-P Model
`CILUPreconditioner::`**`operator`**`()(...)`	$-50.2 + 7.70 * log2(x)$
`CSysSolve::Solve(...)`	$-24.0 + 3.69 * log2(x)$
`CEulerSolver::Centered_Residual(...)`	$-9.5 + 1.46 * log2(x)$
`CSysMatrixVectorProduct::`**`operator`**`()(...)`	$-9.4 + 1.44 * log2(x)$
`CSysSolve::FGMRES_LinSolver(...)`	$-5.7 + 0.88 * log2(x)$
`CMultiGridIntegration::MultiGrid_Cycle(...)`	$-1.5 + 0.22 * log2(x)$

Table 5.6.: Performance models generated by Extra-P for the most relevant functions in SU2 according to PIRA's extrapolation scheme. Model constants are shown with one significant digit and function signatures omitted for brevity.

comparison, the **final** PIRA runtime overhead is $\approx 35\,\%$ compared to the vanilla execution, and about $\approx 3.5\times$ less than the Score-P overhead. When we consider the time measured *within* the Cube profile to factor-out Score-P's start-up overhead, the final PIRA iteration introduces almost no overhead.

Table 5.6 shows the functions in SU2 with the most important runtime models, i.e., the models that evaluate to the largest values in the extrapolation. Compared to the results we achieved with PIRA's Runtime Heuristic, we find that the former hot spot (`FGMRES_LinSolve`) is no longer the most important function. Instead, the application of the preconditioner evaluates to a more important component.

473.astar For 473.astar, we vary the number of waypoints on the map to generate the differently sized input data, and use the values $5k$, $10k$, $25k$, $50k$, and $75k$. Note that this is a different data set compared to the one used in the evaluation of the Runtime Heuristic and overhead numbers cannot be compared reasonably.

Initially, PIRA selects the same 34 functions as with the Runtime Heuristics, since both heuristics use the same initial static selection. Thereof are 29 functions actually executed

Function	Extra-P Model
`wayobj::makebound2(int*, int, int*)`	$-2.3 + 0.0003 * x^{\frac{1}{2}} * log2(x)$
`regwayobj::fill(regobj*, regobj*)`	$10.8 + (-0.04) * log2(x)$
`way2obj::releasepoint(int, int)`	4.11
`way2obj::releasebound()`	3.36

Table 5.7.: Performance models generated by Extra-P for the most relevant functions in 473.astar according to PIRA's extrapolation scheme. Model constants are shown with one significant digit for brevity.

at runtime. For most of the functions, Extra-P finds a constant model which is well below the threshold. Hence, subsequently PIRA instruments 13 functions, which results in an overhead of $1.25\times$ over the vanilla execution for all input data sets.

The total runtime for PIRA with its Model Heuristic is given in Table 5.5, together with time required for all measurements using a vanilla build of the target and a Score-P with enabled inline filtering. Compared to the Score-P version, PIRA's final IC is ≈ 4.7 faster, and limits the instrumentation to relevant regions. This keeps the runtime overhead between $\approx 1.40\times$ and $\approx 1.15\times$, depending on the particular input size used.

Table 5.7 shows the models for the most important functions in the 473.astar benchmark as identified by PIRA's Model Heuristic. We see that three of the four functions identified are of constant runtime w.r.t. the varied input parameter x, i.e., the number of waypoints.

MILC This experiment is a multi-parameter study in which the combination of varied input parameters together with a varied number of MPI processes is investigated. We use fairly small data sets in these experiments to account for the memory footprint with smaller numbers of MPI processes, i.e., the largest grid size and eight MPI processes consumes $\approx$ 4.5GB per process. In the multi parameter case, all combinations of the input parameters need to be run in order to allow Extra-P to generate the performance models (*cf.* Section 5.3.3). Table 5.8 lists the values used to generate the required 25 different combinations. Each is repeated five times for statistical fidelity as recommended by Extra-P, resulting in 125 individual measurements.

The runtimes obtained are listed in the last row of Table 5.9 and show that the average impact of the final PIRA instrumentation is considerably large with an increase of $3.12\times$. Score-P's inline filtering results in a slowdown of $\approx 4.05\times$ compared to a vanilla execution of all required combinations. Compared to the influence of an unfiltered Score-P measurement overhead — which is approximately a $38.56\times$ slowdown — both approaches

Number of MPI Processes	Grid Size
8	16 x 16 x 64 x 16
16	32 x 32 x 64 x 16
32	64 x 64 x 64 x 16
64	128 x 128 x 64 x 16
128	256 x 256 x 64 x 16

Table 5.8.: The values used as input for MILC in the multi-parameter study. All combinations of the parameters are required.

Benchmark	Vanilla	Score-P w/ Filter	Ovh. Score-P	PIRA II Total	Ovh. PIRA II	PIRA II Final	Ovh. PIRA II
MILC	66	333	405%	3,180	4,718%	272	312%

Table 5.9.: Accumulated time (seconds) to execute measurements for Extra-P: no instrumentation (*Vanilla*), Score-P with inline filtering (*Score-P w/ Filter*), PIRA II with all iterations (*PIRA II Total*), and the final PIRA II configuration (*PIRA II Final*).

achieve $10x$ smaller slowdowns. Note that a considerable amount of the overhead stems from the MPI measurements in this scenario.

In the multi parameter setting, the total time PIRA takes to construct the final configuration is larger than the time necessary to conduct all measurements with Score-P. However, the influence of the measurement system in the selectively instrumented binary generated by PIRA should be lower than for the fully instrumented one, decreasing its overall influence on the modeling process. Also, the number of functions presented to the user is reduced by PIRA, although the number of functions in the full profile is not overwhelmingly large.

Load Imbalance Heuristics

In our evaluation, we apply PIRA-LIDe to the LULESH mini-app and the ISSM. The LULESH mini-app allows introducing artificial load imbalance by changing the computational cost of material properties in the evaluation of the equation of state. A load-imbalance parameter b determines the amount of imbalance introduced into the application, with larger values meaning more load imbalance. For ISSM, we use the Greenland model that was used in a recent scalability study in [33]. The model is available in two different resolutions that are denoted by the elements' minimal edge-length. This means that smaller number

Benchmark		Sections	Metric [%]	Total Time [sec.]
LULESH	-b 0	4 ± 1	45	$3{,}532 \pm 29$
	-b 1	3 ± 0	56	$3{,}599 \pm 52$
	-b 2	4 ± 1	61	$3{,}658 \pm 12$
ISSM	G4000/1	4 ± 0	23	$15{,}413 \pm 349$
	G4000/10	4 ± 0	21	$18{,}623 \pm 362$
	G500/1	4 ± 0	13	$22{,}181 \pm 106$

Table 5.10.: PIRA-LIDe's execution result on LULESH and the ISSM for different inputs using 64 MPI processes. *Sections* is the number of sections reported as load imbalanced. *Metric* is the largest imbalance-percentage value in the last PIRA iteration. *Total Time* denotes the total PIRA runtime (including initial build, CG extraction, analysis, profiling, etc.) in seconds. Notation: μ (mean) $\pm$ σ (std. deviation)

mean higher resolution and the coarser model is not viable for the study of physical phenomena. We run LULESH without OpenMP using 8, 27, and 64 MPI processes and the load-imbalance parameter b set to the values 0, 1, and 2. The remaining parameters are left with their default values. For the ISSM, we use 27, 64, and 96 MPI processes to keep the runtime within reasonable limits, and use two different mesh resolutions. We run 1 and 10 timesteps of the coarser mesh (G4000/1, G4000/10), while for the finer mesh we run only 1 timestep (G500/1). PIRA-LIDe uses its default configuration: Imbalance threshold: 5%, relevance threshold: 2%, child constant threshold: 150, child fraction: $3 \cdot 10^{-5}$, PIRA iterations: 20, context handling strategy: `MajorPathsToMain`. However, for trace generation, PIRA-LIDe's `FindSynchronizationPoints` context handling is used.

Table 5.10 shows that PIRA-LIDe identifies four imbalanced sections in the ISSM. For LULESH the number of reported sections varies by one across repetitions. Analyzing PIRA-LIDe's output shows that this is due to a LULESH function with a computed Imbalance Percentage around the threshold used. Hence, depending on measurement variation, the function is included or excluded. Note that the table shows the accumulated runtime for all 20 PIRA iterations and not the time of a single target execution, i.e., the time it takes to conduct the load imbalance detection.

As expected, the imbalance metric increases with LULESH's parameter b for artificial load imbalance. However, even without artificial load imbalance, PIRA-LIDe reports sections as imbalanced. Manual analysis shows that this is correct, as $b = 0$ does not perfectly balance a LULESH execution. Interestingly, we consistently observe negative runtime overhead for LULESH, i.e., the PIRA-LIDe version runs faster than a vanilla version.

Benchmark		Vanilla	PIRA LIDe		Score-P filtered	
		Time [sec.]	Time [sec.]	Ovh. [%]	Time [sec.]	Ovh. [%]
LULESH	-b 0	165 ± 0	163 ± 1	-1.1 ± 0.2	297 ± 3	55 ± 1
	-b 1	171 ± 1	167 ± 1	-1.9 ± 0.4	300 ± 1	56 ± 1
	-b 2	173 ± 1	169 ± 1	-1.9 ± 0.3	307 ± 1	55 ± 1
ISSM	G4000/1	25 ± 1	29 ± 0	14.9 ± 4.9	272 ± 3	900 ± 12
	G4000/10	142 ± 1	152 ± 1	6.4 ± 0.4	$1{,}948 \pm 22$	$1{,}154 \pm 7$
	G500/1	349 ± 9	362 ± 5	3.9 ± 1.8	$2{,}727 \pm 31$	644 ± 11

Table 5.11.: Profiling overhead measured during PIRA-LIDe's execution on different LULESH and ISSM inputs compared to runs with no instrumentation (*Vanilla*) and Score-P-filtered instrumentation respectively. *Time* is the runtime of a single execution in seconds and *Ovh.* the corresponding relative overhead. For PIRA-LIDe, the maximum overhead across PIRA iterations in noted. Notation: μ (mean) $\pm$ σ (std. deviation)

For the ISSM, the maximum value for the imbalance percentage is moderate compared to the one observed for LULESH. Nevertheless, PIRA-LIDe identifies multiple load-imbalanced sections in the ISSM, independent of the model configuration used.

We compare PIRA-LIDe's overhead to Score-P's automatic instrumentation options. *Score-P full* is a full instrumentation of the target application, i.e., every function is instrumented. *Score-P filtered* applies an **`inline`**-filter, i.e., only functions that are not inlined by the compiler are instrumented. Table 5.11 shows the respective runtimes along with vanilla measurements, i.e., a single execution of the target application. For PIRA, we list the largest runtime overhead that occurred across all PIRA iterations. Figure 5.11 and Figure 5.12 depict the relative overhead compared to both of these instrumentation modes for the ISSM and LULESH, respectively. It is visible in Figure 5.13 that the overhead does only increase marginally across the different iteration of iterative descent.

PIRA-LIDe reduces the ISSM runtime overhead by a factor of at least 14 (26) compared to Score-P filtered (Score-P full). Also, it exceeds 20% of runtime overhead in only one case for the ISSM, *cf.* Figure 5.12, G4000/1. Interestingly, this overhead is not noticed for larger models. We found that this is partially caused by constant setup overhead of the measurement system. Such overhead imposes a larger impact for shorter running targets, e.g., smaller models. A typical overhead of less than $\approx 10\%$ is maintained for every other benchmark variant. While Score-P filtered can reduce the excessive overhead of Score-P full drastically in both cases, it never achieves an acceptable runtime overhead

Benchmark		Tracing time [sec.]			Trace size [GB]		
		PIRA LIDe	Score-P filt.	Impr.	PIRA LIDe	Score-P filt.	Impr.
LULESH	-b 0	161	440	$2.7\times$	1.1	927	$843\times$
	-b 1	166	433	$2.6\times$	1.1	927	$843\times$
	-b 2	167	478	$2.9\times$	1.1	927	$843\times$
ISSM	G4000/1	28	461	$16.5\times$	1.3	1,879	$1{,}445\times$
	G4000/10	150	N/A	N/A	11	(14,000)	$(1{,}273\times)$
	G500/1	353	N/A	N/A	12	(20,000)	$(1{,}667\times)$

Table 5.12.: Results of tracing experiments with different ISSM and LULESH inputs using instrumentation generated by PIRA-LIDe and Score-P filtered, respectively. *Tracing time* denotes the runtime of a single tracing run and *Trace size* the size of the resulting trace file. *Impr.* is the relative improvement of PIRA-LIDe instrumentation when compared to Score-P filtered. Values in brackets are estimated by Score-P.

without further manual filtering. Finally, the measurements show that the overhead barely increases with the number of MPI processes.

PIRA-LIDe's overhead reduction is accompanied by a substantial increase in total runtime, *cf.* Table 5.10. In the case of the ISSM (G500/1-model), the total PIRA runtime accumulates to ≈ 6 h (initial build ≈ 2 min, CG collection ≈ 10 min, *Build* ≈ 25 min, *Run* ≈ 2 h, *Analyze* ≈ 3.5 h), which is significantly longer than one run using Score-P's filtered instrumentation. Two major runtime contributors in the analysis step are (1) the graph construction, and, (2) the graph annotation with profile information.

Scalasca Trace Analysis Table 5.12 lists the trace size for LULESH and the ISSM running with 64 MPI processes. Scalasca's analysis requires the trace information to reside in main memory for its analyses. Hence, traces obtained by Score-P full and ISSM traces obtained by Score-P filtered cannot be analyzed with Scalasca as they exceed the 384GB of main memory available in nodes of the Lichtenberg cluster. PIRA-LIDe reduces the trace sizes for LULESH and the ISSM by multiple orders of magnitude, *cf.* Table 5.12, significantly speeding-up or enabling a Scalasca analysis.

The most significant values for Scalasca's *critical-path* and *critical-path imbalance impact* metrics are shown in Table 5.13 for a LULESH execution with 64 MPI processes and PIRA-LIDe's final instrumentation. Higher values reflect more significance, and the functions are listed top to bottom according to the call context, i.e., `LagrangeLeapFrog` calls

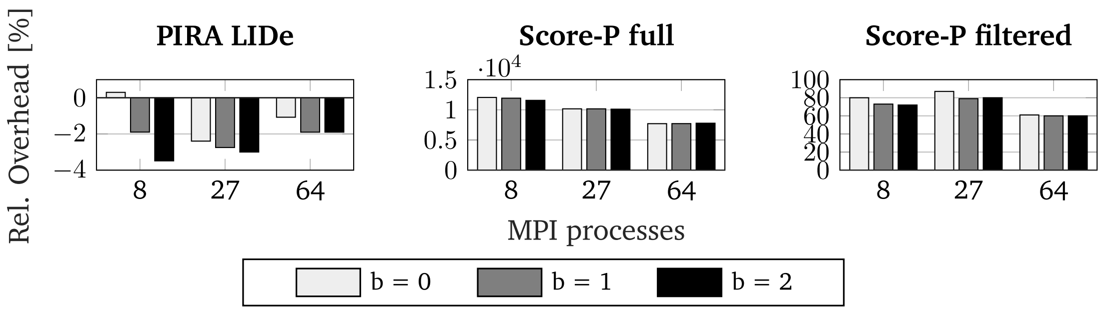


Figure 5.11.: Relative runtime overhead for PIRA-LIDe (one repetition) and Score-P on LULESH.

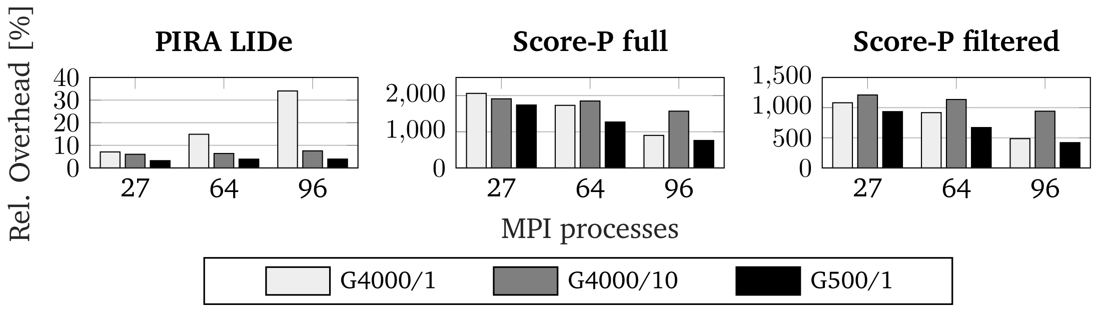


Figure 5.12.: Relative runtime overhead for PIRA-LIDe (one repetition) and Score-P on the ISSM.

`LagrangeElements`, etc. While the critical-path metric changes only slightly between the results for $b = 0$ and $b = 1$, i.e., artificially introduced load imbalance, the imbalance-impact metric changes significantly for two functions. The value for the `LagrangeLeapFrog` increases from 18.8 to 52.5 and the value for `ApplyMaterialPropertiesForElems` from 459.6 to 872, i.e., their imbalance is much more pronounced for $b = 1$.

Table 5.14 lists the respective Scalasca metric values for the sections in the ISSM identified by PIRA-LIDe. The metric values suggest that imbalances with comparable impact exist in code sections of LULESH and the ISSM. More importantly, the load imbalance becomes apparent for the physically relevant model G500/1 in case of the *original* imbalanced version of the ISSM for all sections identified. As part of our work, these load imbalances have been addressed, *cf.* Table 5.14 the two most-right columns, and are discussed in more detail in [9].

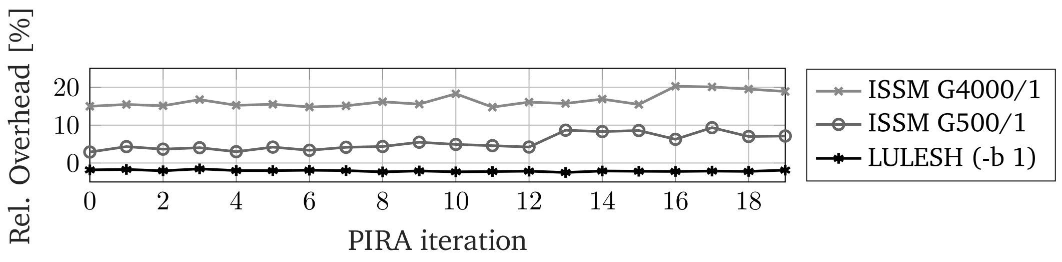


Figure 5.13.: Relative runtime overhead per iteration with 64 MPI processes for a single PIRA-LIDe execution.

LULESH Region	b=0 Critical Path	b=0 Imbalance Impact	b=1 Critical Path	b=1 Imbalance Impact
`LagrangeLeapFrog`	116.5	18.8	118.3	52.5
`LagrangeElements`	13.9	6.5	13.7	2.2
`CalcQForElems`	12.7	55.3	11.1	5.4
`ApplyMaterialProp...`	19.7	459.6	24.7	872.0

Table 5.13.: Scalasca analysis for PIRA-LIDe traces of LULESH for $b = \{0, 1\}$ and 64 MPI processes. Imbalance Impact is Scalasca's Critical-path Imbalance Impact.

5.5. Discussion

In almost all benchmarks, PIRA significantly reduced the runtime overhead for the target application. The reduction is achieved by excluding functions that do not contribute significant amounts of total runtime. As a result, an analyst is presented with a less-cluttered profile to inspect, thus, focusing on the most-important functions in the target application. PIRA, however, struggled to reduce the runtime overhead below 15 % in some cases. In particular, some *C++* targets from the SPEC CPU benchmark suite proved to be challenging for the Runtime Heuristic. In addition, recursive algorithms naturally pose a challenge to function-level instrumentation measurements. Nevertheless, the large reduction of runtime overhead leads to a good starting point for a manual focus analysis.

The PIRA heuristics use threshold filtering to determine which functions to exclude from measurement. For both the Runtime Heuristic and the Load-Imbalance Detection, the thresholds are computed relative to the application runtime. However, in the case of the Performance-Model Heuristic, the threshold currently is a fixed value. This leads to

ISSM	**Imbalanced G4000/1**		**Imbalanced G500/1**		**Improved G500/1**	
Region	Critical Path	Imbal. Impact	Critical Path	Imbal. Impact	Critical Path	Imbal. Impact
`SbA::CreateKMatrixHO`	8.5	108.0	84.1	601.0	77.6	279.9
`SbA::CreatePVector`	1.0	11.2	10.1	72.7	9.9	60.9
`EA::CreatePVector`	1.3	16.2	12.7	89.7	11.8	40.9

Table 5.14.: Scalasca analysis for PIRA-LIDe traces of the ISSM and 64 MPI processes. Imbalance Impact is Scalasca's Critical-path Imbalance Impact.

insufficient adaption for some target codes. Also, the thresholds are computed based on the `main` function. This can prune too many functions from the measurement early on.

The Performance-Model Heuristic is able to filter more functions from the IC compared to the Runtime Heuristic. Since it determines which function's runtime scales with the input parameter, it can exclude large, yet constant functions. Hence, it allows an analyst to obtain a picture of the application's functions that *may become* hot-spots for larger input data, or when scaling to larger machines and more parallelism. This is particularly helpful before a code is used on different machines or larger problem sizes. However, the analysis depends on the parameters set by the user. This means that, for example, should the code be limited in MPI-scalability, but the user chose not to model the code's MPI behavior, PIRA is unable to detect the limitation. It will, thus, focus the instrumentation to other regions, or even filter all instrumentation, since no function crosses the thresholds set.

Finally, the Load-Imbalance Detection was able to identify the load imbalances present in the target applications. In ISSM, the load imbalances were manually validated in a recent performance and scaling analysis, and subsequently addressed. Moreover, PIRA pinpointed the load imbalance in the LULESH mini-app to the particular function that introduces the imbalance. Score-P, on the other hand, filtered this function due to its **`inline`** annotation, hence, an analyst may have been misled by the profile.

Once identified, the important regions can be the target of performance analysis and tuning or performance exploration. Given the vastness of state-of-the-art scientific applications, the analysis and tuning can be significantly challenging. Therefore, the extraction of relevant regions and the construction of representative mini-apps is intriguing, as they allow easier analysis, refactoring, or even re-implementation in a different programming language. In the next chapter, we present an approach that, using PIRA, identifies the most relevant parts of the application, and, subsequently, extracts them into a mini-app.

6. Mini-AppEx: Tool-Supported Kernel and Mini-app Extraction

The chapter is based on the following publications and contains verbatim excerpts of parts which were contributed by the thesis author.

Lehr, Jan-Patrick and Bischof, Christian and Dewald, Florian and Mantel, Heiko and Norouzi, Mohammad and Wolf, Felix. 2021. *Tool-Supported Mini-app Extraction to Facilitate Program Analysis and Parallelization* [72]

Lehr, Jan-Patrick and Hück, Alexander and Fischer, Moritz and Bischof, Christian. 2020. *Compiler-Assisted Type-Safe Checkpointing* [75]

In this chapter, we use the aforementioned tool PIRA to obtain reasonable measurements for kernel identification as a starting point for tool-supported mini-app extraction. We used a similar version of the kernel identification to identify application kernels and subsequently recombine them using the multitier-reactive programming language ScalaLoci [137] to enable hybrid on-premise/cloud execution [124]. However, this work is not discussed in this chapter. The contributions of this chapter are, in particular, the general approach to the tool-supported mini-app extraction, and the novel Clang-based source-to-source translator Mini-AppEx. As part of the extraction process, application data needs to be captured, for which we develop a type-safe CPR abstraction based on TypeART [48].

In the remainder of this chapter, we outline the general approach in more detail. Thereafter, we briefly touch on the extensions to PIRA that were introduced for this use case. We present both the source-to-source tool Mini-AppEx and the type-safe CPR abstraction in more detail. Finally, we apply the approach to the 8.5 million lines of code (astro)physics simulation eos-mbpt [31] to extract a representative mini-app. The mini-app is subsequently used in an evaluation for automatically detecting parallelization opportunities with the DiscoPoP [80] tool.

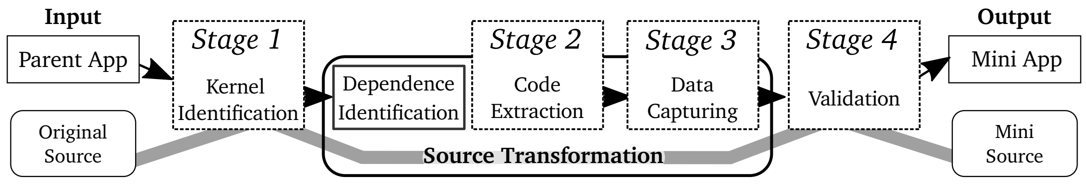


Figure 6.1.: An overview of the mini-app extraction process: (1) Kernels, e.g., functions with considerable and problem-size-dependent runtime, are identified. (2) Kernels are extracted and driver code is generated. (3) Data required in the kernels is identified and captured. (4) Mini-app is validated for correctness of results and execution representativeness.

6.1. Approach

Our approach is composed of four stages, which are shown in Figure 6.1. The stages serve as a general blueprint for the extraction of mini-apps. While we provide tools for each individual stage to address the particular challenge, the general high-level workflow can also be followed with other tools. The subsequent sections present an outline together with technical details on the respective stages in the workflow.

6.1.1. Kernel Identification

The first stage is the identification of the target's kernels, i.e., the most significant parts to be extracted into the mini-app. Determining which kernels are of interest is use-case dependent. Here, the kernels are intended as target of tool-supported parallelization, hence, functions that show increasing runtime for larger input data are of interest.

To identify such kernels, our approach relies on automatic instrumentation refinement provided by PIRA, see Chapter 5. It heuristically creates low-overhead instrumentation configurations that are iteratively adapted by analyzing the runtime profile obtained. To identify kernels, we extend PIRA to sort and output the functions according to their relevance of the extrapolated performance model values. While the performance modeling itself is metric-independent, we use wall-clock time, as a developer can easily relate to it, and the goal of the subsequent parallelization is to reduce execution time.

6.1.2. Source Transformation

Given the list of kernels identified, the next step is to extract the kernels and their respective dependencies to form the mini-app. We consider the mini-app to be a single file that contains the necessary functions, their dependencies and the respective driver code to execute it. Hence, starting from the kernels, all required variables and their types need to be identified. We implemented a novel Clang-based source-to-source translator that (1) computes required dependencies using the compiler's AST, (2) extracts the identified dependencies into a new file, and, (3) implements the generation of CPR code. The next sections present more detail on the implementation of the translator.

Dependency Identification To identify the dependencies, our tool needs to consider *functions*, *variables*, and *types*. It implements a mark-and-sweep approach, meaning that, first, it identifies the required functions starting from the kernel call-site. Processing the functions marked, it, second, identifies and marks the required variables and their types.

Functions: The mark-and-sweep implementation reuses the application's CG constructed from MetaCG, *cf*. Chapter 4, to determine call targets for each function. Our tool reads the serialized MetaCG, and attaches pointers to Clang-AST function definitions present in the program's current TU. Hence, the tool can check whether a specific function definition is present in the current TU and retrieve it, but, more importantly, it can perform whole-program call-dependence analysis while working on a single TU. The kernel call-site, i.e., starting point, is given as source position (`filename:line:column`) that is mapped to the respective Clang-$*$ node by matching the source position. Using the $*$ node and the kernel's function name, the CG analysis, (1) traverses the MetaCG to determine the required callees starting from the kernel, (2) checks for each callee if its function definition is present in the current TU, (3) marks each required function definition for extraction and variable-dependence analysis.

Variables: For each function defined in the current TU, the tool identifies the accessed (global) variables. Locally declared variables are not handled explicitly, as they are extracted as part of individual functions. In addition, the tool identifies all *allocation statements* for global variables, e.g., calls to **`new`** or `malloc`. Therefore, the tool traverses the AST and inspects assignments to the global variables marked previously. Should the assignment to the variable be the result of a call to an allocation function, the respective allocation statement is marked as an allocation dependency of that variable. Allocation statements are then, recursively, inspected for variables they depend on.

Types: Marking and subsequently extracting types is required for user-defined types, such as **`struct`** or **`typedef`** definitions. For each marked variable, the type is inspected via

```
 1 double some_const;// required global variable
 2 float compute_X();// non-system function
 3 void kernel(int i, double *pd, double *r) {
 4   // performs heavy computation with pd
 5   r = some_const * compute_X() * ...;
 6 }
 7 double* compute(int arg_i, double* arg_d) {
 8   double *res = malloc(sizeof(double));
 9   kernel(arg_i, arg_d, res);// kernel call-site
10   return res;
11 }
```

Listing 6.1: Dependencies for the function `kernel`: the `IN` variable `arg_i`; the `OUT` variables `arg_d` and `res`; the global variable `some_const`; and the function `compute_X` and its dependencies.

the Clang-AST. Whenever a user-defined type is encountered, the type's definition is retrieved and marked for extraction. The type's definition is also recursively inspected for potentially other required user-defined types to be marked.

The code generation for allocation dependencies is handled when generating the mini-app's `main` function. Listing 6.1 shows an example, in which the function `kernel` is identified as the function to extract. Note that the tool also considers the variables passed into the kernel at the respective call-site, such as the variable `res` in the example.

Code Extraction The entry point for the code extraction is a designated call-site to the kernel function in the original application, *cf.* line 9 in Listing 6.1. From the call-site, the tool uses the dependencies identified in the *dependency identification* step to copy the required source code into a new file. The copy mechanism uses Clang's built-in mechanism to pretty-print all marked *C/C++* constructs — functions, variables, and types. External and system library dependencies are maintained, i.e., the respective include statements are added in the mini-app for external or standard library functions. If the translator is unable to determine the correct `include` statement, it outputs a hint to the developer listing the function in question along the original full include-file path.

Along the extracted source code, the translator generates (1) a `main` function to execute the final mini-app, and, (2) a kernel-wrapper to capture application data. The previously captured allocation statements for global variables are output as part of the `main` function. This is important, as otherwise the CPR mechanism cannot restore memory content into such allocations. For multi-dimensional arrays, the respective loops are generated, of which the loop bounds may depend on values provided by the CPR mechanism. Hence, dependencies are sorted and output into the mini-app that for a given statement `stmt(a)` the variable it depends on (`a`) is read from the checkpoint before the dependent statement. To subsequently fill such allocations with the application data, calls to our CPR library are generated to read the data from a checkpoint file. The kernel wrapper is generated as drop-in replacement for the original kernel call-site, i.e., it has the same type signature as the original function. It can be placed at the location of the original call-site and registers and stores the required application data into a checkpoint file. Note that the CPR mechanism used applies a two-step approach, by first marking memory regions that should be handled, and, second, calling either checkpoint or restore. Hence, most of the CPR code is shared between the kernel wrapper and the `main` function differing only in whether it should record or restore.

Data Capturing To enable the execution of the mini-app, the application data is captured from the original application and subsequently read by the mini-app. As mentioned in the previous section, the source-to-source translator generates a kernel-wrapper to be placed into the original application and automatically capture the variables' contents. Our approach relies on our type-safe CPR interface TyCart [75] and enables both a lightweight sequential, and an MPI-parallel data capture using the VeloC [103] or the Fault Tolerance Interface [11] checkpoint library. TyCart uses the type tracking and sanitizer tool TypeART to provide its type-checking, *cf.* Figure 6.2.

We extended the available approach to automatically capture memory allocations without the need to specify their size explicitly. This simplifies the generation of the checkpoint statements, as it defers the computation of non-static allocation bounds to the runtime. Moreover, we added lightweight introspection mechanisms to retrieve the size of an allocation through an API. This simplifies the recording of data for multidimensional arrays in a similar fashion as it simplifies the generation of allocation statements explained in the previous section.

6.1.3. Validation

We capture different runtime metrics of the target application to compare it to the original application as proposed by Aaziz et al. [1]. Specifically, we capture hardware performance counter measurements — using Caliper [16] for JSON output on top of PAPI [18] — for the kernel regions, as well as the dynamic instruction mix (using PIN [85]) of the target's execution. The hardware performance counter measurements are fed into a bottom-up hierarchical clustering algorithm (HCA), i.e., each feature vector starts as its own cluster

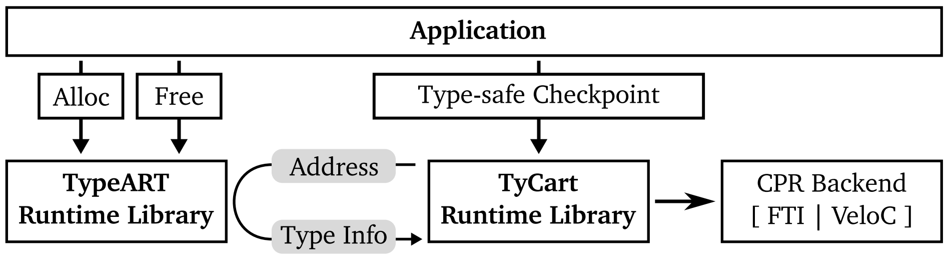


Figure 6.2.: Instead of direct calls to a CPR library, the equivalent type-safe API of TyCart is used. For each invocation, the type information of the memory pointer, provided by TypeART, is compared to the user-specified one of the type assert. A successful check passes the call to the respective CPR backend.

and the clusters are successively merged. To increase the number of feature vectors available to the HCA, we also capture hardware performance measurements for each function individually. The final hierarchy, hence, reflects the similarity between the respective feature vectors. Given that the runtime measurements, particularly the hardware performance counter measurements, can be noisy we use Ward's linkage criterion [134]. This linkage criterion constructs clusters of pairs of two, which makes it easier for a human analyst to perceive the cluster members.

6.2. Evaluation

We apply the extraction approach to the eos-mbpt astrophysics application [31]. To rate the quality of the mini-app, we consider both the reduction in code complexity by creating the mini-app, as well as its representativeness. eos-mbpt computes the equation of state for nuclear matter by calculating energy diagrams in a perturbative expansion. In this work, one of those diagrams, the so-called Hartree-Fock (HF) energy, is considered. The eos-mbpt code base consists of ≈ 8.5 million lines of code and the main contributor to runtime is the computation of (high-dimensional) integrals using a Monte-Carlo integration. To drive the integration, eos-mbpt uses the Cuba library [41]. For the required spline interpolations, it relies on the GNU scientific library (GSL) [38] that we use with the OpenBLAS [133] package. While the Cuba library offers parallelization using `fork()` at the Monte Carlo point level, we execute the application sequentially, to fulfill DiscoPoP's requirement of a sequential input application. Finally, we apply and evaluate DiscoPoP's ability to identify parallelization opportunities and their speed-up.

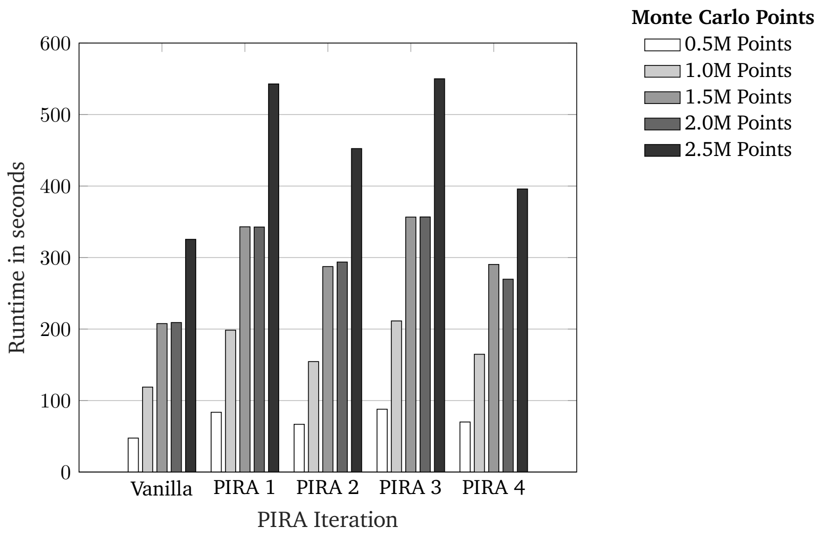


Figure 6.3.: Runtime of eos-mbpt with different numbers of Monte-Carlo points (in million) for vanilla (w/o instrumentation) and four PIRA iterations (w/ instrumentation). Runtime for standard Score-P measurements for $0.5, 1.0, 1.5, 2.0$, and $2.5M$ points: $695\,s, 1{,}981\,s, 3{,}461\,s, 3{,}169\,s, 5{,}126\,s$.

6.2.1. Extraction Approach

We apply PIRA for the automatic kernel identification using performance models. The number of Monte-Carlo points serves as the modeling parameter. PIRA is applied running the default four iterations, i.e., one initial and three refinement iterations, with a final runtime overhead of $\approx 75\,\%$. From Figure 6.3, we can see the runtime impact of the iterative refinement. It reduces the profile to a total of 16 regions, compared to 584 in a default Score-P [64] profile. Table 6.1 lists the constructed performance models for the eos-mbpt kernels after the fourth PIRA iteration, while Table 6.2 presents an excerpt of the final iteration's profile and a breakdown of the runtime share of the different functions. We see that the identified kernels consume the largest share of the runtime, with `f_NN_HF` being the most important one. `Int_NN_HF` is an important wrapper function on the call path from `main` to `f_NN_HF`, which is why it is listed despite its very short runtime.

Given the identified kernels and the final runtime profile, we determine a single kernel call-site as the starting point of the extraction process. Starting from this call-site, the

Function	Performance Model
`f_NN_HF`	$-4.5 + 4.2^{-5} \times N$
`get_qNN`	$-2.3 + 2.3^{-5} \times N$
`SphericalY`	$-4.8 + 8.5^{-5} \times (N^{\frac{3}{4}}) \times log2(N)$

Table 6.1.: The identified kernels and their constructed performance models as a function of the number of Monte-Carlo points N. C_i denote constants.

Function	**Runtime (s)**		**% Total**
	Inclusive	Exclusive	
`main`	490.0	1.2	0.3 %
`Int_NN_HF`	488.9	1.2	0.3 %
`f_NN_HF`	488.1	359.9	73.5 %
`get_qNN`	128.2	26.3	5.4 %
`SphericalY`	101.9	101.9	20.8 %

Table 6.2.: Excerpt of a flat profile and breakdown of runtime share for iteration PIRA 4 and $1.0M$ Monte-Carlo points. Sum of percentages may exceed 100 % due to rounding.

translator identifies and marks 15 functions, 21 (global) variables, and 18 types — including built-ins — to extract. Three of the functions come from two third-party libraries, for which **`#include`** statements are required. The translator is able to generate the correct include statement for the system includes required, while for third party libraries it outputs hints to the developer which files are needed. In addition to the global variables, 17 local variables are required for the kernel call-site. For all variables the respective CPR calls are generated, and for four of the variables determining the allocation size is deferred to application runtime.

Table 6.3 shows the significant reduction that the tool-supported extraction approach achieves compared to the original application. *Mini-app* is a fully automatically extracted mini-app that maintains all calls to third-party libraries. *MCS Mini-app* is a manually augmented version of Mini-app that includes all code required for the Monte-Carlo integration. We see that our approach is able to reduce the number of lines of code (LOC) drastically ($\approx 7{,}500\times$) compared to the original application. Also, the number of functions, reachable functions, variables and types is significantly reduced in the mini-app as well. We count types, e.g., **`int`**, and pointers to such types as the same type, but count **`typedef`** types as their own type, i.e., **`int`** `*` is not a new type whereas **`typedef int`**`* IntP` is. In the original

Metric	Original	Mini-app	MCS Mini-app
Lines of Code	8,571,815	422	1,117
Functions	31,196	46	82
– Reachable	445	33	52
Variables	1,206	183	356
Types	462	18	94

Table 6.3.: Reduction in size and complexity achieved with our tool-supported extraction approach (Mini-app) and manual addition of third-party library functions (MCS Mini-app).

application, matrices of function pointers are stored, thus the relatively high number of functions (31,196). However, our configuration does not enable paths to these matrices, thus, the much smaller number of reachable functions (445). In particular, the automated extraction results in 46 functions of which 33 are reachable, and only 183 variables within the program. Manually adding the Monte-Carlo integration source code increases this complexity again to 82 functions of which 52 are reachable and a total of 356 variables. This increase is due to additional functions, previously hidden in the Cuba library implementation, and now accessible in the mini-app. The integration of the Monte-Carlo integration source code removes the previously maintained dependency to the third party Cuba library and leaves the GSL and OpenBLAS libraries as only dependencies. We use this final version of the mini-app for our subsequent performance improvement experiments.

6.2.2. Mini-app Quality

The mini-app should reduce the size of the original application, while maintaining key performance characteristics of its execution. Hence, we evaluate the quality of the mini-app extracted by comparing its complexity and execution characteristics after compilation with GCC and optimization level `-O3`.

Hierarchical Clustering The final stage of our approach considers the analysis of representativeness. Figure 6.4 shows the resulting dendrogram from our hierarchical clustering analysis for the kernel region as well as the individual functions using hardware performance counter measurements. The x-axis shows the Ward distance, i.e., the similarity measure, of which lower numbers mean higher similarity and are better. The y-axis shows the specific function measured, prefixed with `eos` for the original application or `miniEOS` for the mini-app, while the number in parentheses denotes multiples of the same regions.

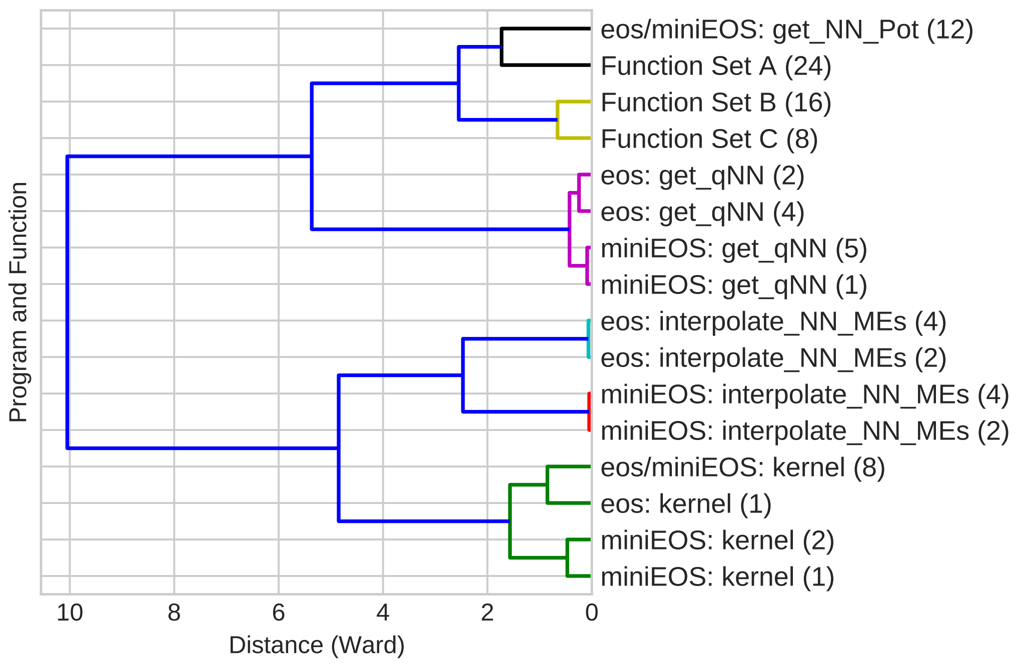


Figure 6.4.: Dendrogram resulting from representativeness validation using hardware performance counter measurements on the eos-mbpt original and mini-app.

This multiplicity arises from multiple executions of the hardware performance counter measurements and very low Ward distances. `Function Set A`, `B`, `C` are introduced for brevity and refer to the functions listed in Table 6.4.

From Figure 6.4 we can see that (1) the kernel region, i.e., the kernel and its dependencies, of the original and the mini-app are similar to each other and form one cluster (bottom four), (2) the same function from one application is usually its own cluster, and, (3) the same function from the original and the mini-app are typically clustered next. This means that we consider the observable behavior of the kernel region similar, hence, representative. However, the Ward distance is a relative measure of similarity, thus, the clusters may be misleading Consequently, the following paragraphs present a more detailed evaluation of the representativeness and go beyond the hierarchical clustering, while using the same measurement data.

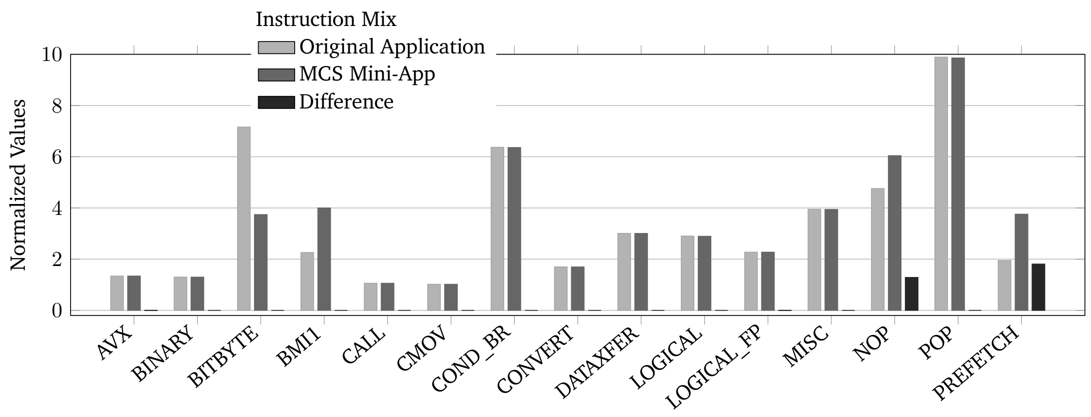
Function Set A	Function Set B	Function Set C
`eos:Int_NN_HF`	`miniEOS:Int_NN_HF`	`miniEOS:Int_NN_HF`
`eos:getCartCoords`	`miniEOS:getCartCoords`	`miniEOS:getCartCoords`
`eos:getSpherCoords`	`miniEOS:getSpherCoords`	`miniEOS:getSpherCoords`
`eos:f_NN_HF`	`miniEOS:f_NN_HF`	`miniEOS:f_NN_HF`

Table 6.4.: Function names for Function Set A, B, and C used in the dendrogram in Figure 6.4.

Dynamic Instruction Mix Since the mini-app is much smaller, the compiler may take different decisions when creating the binary. We capture the dynamic instruction mix using a PIN [85] tool to compare how similar the actually executed machine instructions are. Please note that we capture the instruction mix for the full execution of both original and mini-app, thus, we expect some differences. Figure 6.5 depicts the occurrences of instructions of different categories at execution of both original and mini-app. From left to right, the plot shows the different categories and the y-axis shows the occurrence within the different apps, and their difference. The values are normalized by the respective event's order of magnitude to eliminate the large discrepancies in total numbers, e.g., an instruction of the `AVX` category is executed $1{,}340{,}130{,}423{,}838$ times, whereas a `NOP` is executed only $604{,}217{,}093$ times. The difference values are normalized by the order of magnitude obtained from the mini-app measurement. We see that for a few categories, i.e., `NOP`, `PREFETCH`, `STRINGOP`, `SYSCALL`, `SYSTEM`, and `XSAVE`, the difference is significant. For example, the original app performs more file-IO, which explains the difference in the `SYSCALL` and `SYSTEM` categories. However, most categories behave very similar, meaning that the compiler indeed produces similar — if not identical — code for the mini-app.

Execution Performance From Table 6.5 we see that the mini-app can be compiled much faster, while the runtime for the kernel region is almost identical to the original application. Although the compile time of the original application is not dramatically larger, the reduction is significant. The compile-time speedup is due to the mini-app including significantly fewer *C*/*C*++ header files compared to the original application.

The runtime in both original and mini-app is almost identical. This is expected, as the kernel code-path is the same for original and mini-app. We determine the maximum resident set size using the `rusage` utility, i.e., the largest amount of memory occupied at any point of the execution. Table 6.5 shows that the application's memory footprint is in general relatively small. The CPR mechanism in the mini-app increases this demand, due to its internal data buffering.



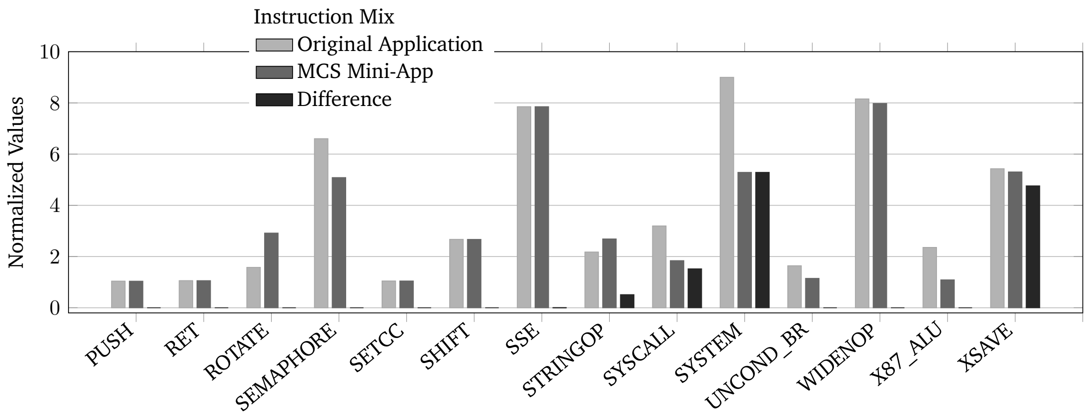


Figure 6.5.: Visualization of the categorized dynamic instruction mix for the original application (light gray), the mini-app (gray), and the difference of both (dark gray). The categories are normalized to the respective order of magnitude to help plotting. The bars for *Original Application* and *MCS Mini-App* should be equal; for *Difference* lower is better.

Metric	Original	MCS Mini-app
Compile Time (s)	152 ± 2	1 ± 0
Runtime (s)	$1{,}458 \pm 11.1$	$1{,}438 \pm 10.0$
Memory (KB)	49,636	112,808
Instructions Executed	9,205,467	9,153,898
Conditional Branches	635,980	635,790
Branches Mispredicted	10,684	10,724
L2D Accesses	45,550	45,467
L2D Misses	18,453	18,624
L2D Reads	39,744	39,462
L2D Writes	1,074	1,069
Stores	599,855	598,899
Loads	2,419,059	2,369,460
Misprediction Rate	1.68	1.69
L2D Miss Rate	40.51	40.96

Table 6.5.: Different metrics relevant in (performance) experimentation for original and mini-app. All hardware performance counter values are in millions of events and give the median over ten repetitions (standard deviation: $< 8\,\%$). L2D refers to the level two data cache.

Also, the hardware performance counter values obtained for the kernel region in both original and mini-app are almost identical. Since the extracted code for the kernel code path is identical between the original and the mini-app, this is not surprising. More importantly, however, is that it indicates that our approach did not simplify the mini-app to a degree that allows the compiler to optimize much more aggressively. We measured level 2 data cache (L2D) accesses and misses, together with data cache reads and writes, as these are often used to investigate memory-subsystem utilization. In addition, we measured load and store instructions, as well as conditional branches executed and branches mispredicted. From Table 6.5, we can see that the measured events are very similar between original and mini-app. As a result, also commonly used derived metrics, such as L2D miss rate (40.51 compared to 40.96) and the branch misprediction rate (1.68 compared to 1.69), are very similar. Hence, important insights that are gained by analyzing the mini-app can be transferred back to the original application.

6.2.3. Tool-supported Parallelization

DiscoPoP [80] is a parallelism discovery tool. It first decomposes a sequential application into so-called Computational Units (CU). The CUs are the basic blocks for the parallelization, i.e., a CU is not well-suited for thread-level parallelization. Then, DiscoPoP extracts data dependencies in the application using a its data-dependence profiler [94], [104]. The profiler instruments memory-access instructions in the code with functions that are invoked during runtime to compute and record data dependencies. To obtain dependencies in different execution paths of a program, users need to employ a representative set of inputs [105]. Using the data dependencies recorded, DiscoPoP identifies parallelization opportunities following parallel design patterns, e.g., DoAll or Pipeline. Once the patterns are found, DiscoPoP points the developer to source-code lines which exhibit the patterns and makes suggestions for the implementation of the patterns in the form of OpenMP constructs [105]. Moreover, it suggests how to classify the data sharing of variables inside the specific regions.

Identification of Patterns We apply DiscoPoP to the final mini-app and identify thread-level parallelization opportunities. To reduce the execution time of the dependence profiling, we reduce the number of Monte-Carlo points to $500{,}000$, whereas the original application performs up to $16{,}250{,}000$. The vanilla runtime of the mini-app using the reduced number of Monte-Carlo points is 45.2 seconds with a standard deviation of less than one percent. The DiscoPoP profiler introduces a slowdown of $289.7\times$, and identifies 42 unique patterns. For the evaluation of the suggested patterns, we focus on suggested DoAll patterns, see Table 6.6.

Pattern	Containing Function	Kernel
350	`get_NN_Pot`	N
442	`get_qNN`	Y
474	`f_NN_HF`	Y
482	`f_NN_HF`	Y
490	`f_NN_HF`	Y

Table 6.6.: Evaluated DoAll pattern suggestions with the containing function and if the function was identified as kernel during the mini-app extraction.

Evaluation of Parallelization Suggestions We implement the different patterns separately and evaluate the speed-up using the original data set size, i.e., $16{,}250{,}000$ Monte-Carlo points. The experiments are run on 2.5GHz Intel Xeon E5-2670v3 processors with frequency scaling and HyperThreading disabled. The runtime results denote the median of 10 consecutive runs on the same processor, and a standard deviation of $\approx 3\,\%$. We varied the number of threads used in the parallel region and varied the thread binding. Figure 6.6 shows the speed-ups achieved with the different patterns. The x-axis shows the different pattern IDs, and the y-axis the speed-up achieved over vanilla execution. We find that the largest speed-up is achieved from using pattern 442 and four threads spread across the four NUMA domains of the processor. This results in a speed-up over the serial execution of $\approx 32\,\%$. Generally, pattern 442 leads to the largest speed-ups when run with four threads. Additionally, pattern 482 achieves a speed-up of up to $\approx 20\,\%$, when run with two threads and compact NUMA distribution.

Transfer to Original Application Given that the relevant regions, i.e., the kernel and its dependencies, are copied from the original application, the transfer of the parallelization opportunities to the original application is straight forward. The user can search for the function in the original application and copy the added OpenMP pragmas to the respective loops. Running the original application with the transferred pattern 442 results in a speed-up of $28\,\%$ for four threads and spread NUMA binding.

6.3. Discussion

Our approach allowed us to extract a representative mini-app from a large-scale simulation code. Increasing the interoperability between PIRA and the source extractor, however, would further reduce the amount of manual work required. Currently, an analyst needs to

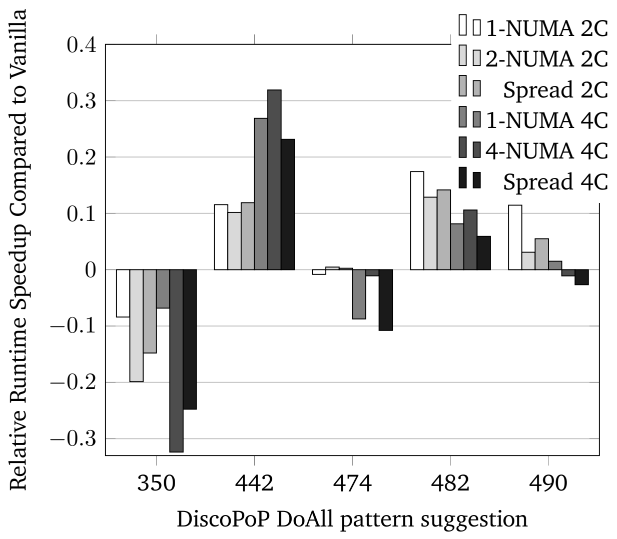


Figure 6.6.: Speed-up achieved with different DiscoPoP-suggested parallelization patterns using 2 and 4 threads and different thread binding: n-NUMA refers to manual pinning of the threads to n NUMA domain(s), whereas *Spread* refers to `OMP_PROC_BIND=SPREAD`.

inspect the generated profiles and match the identified kernels to their respective call sites, which are subsequently provided to the extractor. Furthermore, our evaluation showed that, while PIRA correctly identified the application's kernel, it also generated a significant amount of runtime overhead. Nevertheless, the linear performance model constructed follows our expectations, due to the Monte-Carlo algorithm used.

The source-to-source translator, currently, identifies dependencies in a context- and path-insensitive manner. This becomes apparent in one of the identified dependencies, as the function contains large parts of dead code. However, the extractor, considering the full function body, identified additional dependencies that were not required. We addressed this with preprocessor macros, as the dead code was part of another version of the algorithm. Hence, a user now selects at compile time which algorithm to use.

The approach presented is focused on sequential applications. However, within the workflow, all tools except the source extractor are capable of handling MPI-parallel applications. As the source extractor operates at the function level, we expect that it can handle such codes partially. In its current version, however, the extractor is unaware whether it needs to generate additional MPI-related setup code, e.g., a call to `MPI_Init`, or, potentially more elaborate preparation steps, such as the (re)creation of custom MPI communicators. While we believe that it would extract valid subregions from an MPI application, careful validation of both runtime behavior and computational result is required.

7. Summary

We presented a study on the influence of two state-of-the-art performance measurement tools, with a focus on instrumentation-based measurement. It shows that current instrumentation-based tools provide insufficient automatic filtering, leading to impractically large runtime overhead and measurement perturbation. Our tool PIRA uses whole-program call graph (CG) analysis and automatically refines instrumentation-based measurements towards regions of interest, reducing the runtime overhead and presenting only relevant data to an analyst. We developed MetaCG to provide a flexible whole-program CG infrastructure for compiler-based tools, such as PIRA. Finally, we presented a tool-supported approach to semi-automatically extract mini-apps from existing code-bases.

Measurement Perturbation Study Our introduction pointed out the lack of a study on the influence of state-of-the-art performance measurement tool. To fill the gap, we investigated the influence of instrumentation-based performance measurements with the state-of-the-art tool Score-P. In addition, we obtained data for such measurements using the sampling-based tool HPCToolkit. Particularly for *C*++ applications, the runtime overhead of instrumentation-based measurements can be substantial. While this is not surprising, we also found that, despite the large slowdown, even some finicky-to-obtain hardware performance counter (HWPC) values can be obtained without being perturbed heavily. Even derived metrics, i.e., metrics that take into account multiple low-level HWPC measurements, can be obtained almost non-perturbed. Nevertheless, the large slowdowns in some cases can lead to prohibitively long instrumentation-based measurements.

PIRA Our tool PIRA addresses the currently required manual filter creation for Score-P automatically (1) identifying an initial low-overhead instrumentation, and, (2) refining the instrumentation towards application hot-spots. This frees a performance analyst from tedious manual filter creation and repetitive refinement cycles. PIRA itself is constructed from exchangeable components to allow for flexible extension and uses the widespread Score-P measurement system as profiling backend. Hence, resulting profiles and traces can be viewed and analyzed with the Score-P ecosystem's tools, such as Vampir or Scalasca.

PIRA's analyzer implements different heuristics for instrumentation-filter generation and refinement. For instance, PIRA provides filtering based on empirically-constructed performance models, or the automatic identification of load imbalances in MPI-parallel applications. In our evaluation, PIRA is able to generate instrumentation-based measurements with overheads typically below $5\,\%$ for hot-spot detection, or, $15\,\%$ for load imbalance detection. Since it outputs regular Score-P filter files, an analyst can continue to work from the PIRA-generated filter list for further investigation of the target application.

MetaCG As part of the PIRA development, we implement MetaCG — a lightweight whole-program CG library focusing on *C*++. MetaCG allows to freely annotate function nodes with user-defined metadata, and the subsequent serialization of the CG to exchange data between tools. For the CG construction, we implemented a Clang-based tool that constructs translation unit-local CG and subsequently merges them into the whole-program CG. PIRA facilitates the metadata mechanism for its static instrumentation heuristics that are based on source-code features, but are, conceptually, not tied to a particular compiler.

Mini-AppEx We developed an approach to use the kernels identified with PIRA to extract mini-apps from existing applications. To that end, we implemented a novel Clang-based source-to-source translator, and a type-safe checkpoint/restart abstraction.

The source-to-source translator identifies, starting from a kernel call-site, in a mark-and-sweep fashion all required dependencies, i.e., functions, variables, and types. It reuses the application's MetaCG for whole-program reachability analysis while processing the current translation unit. Its dependency analysis operates at function granularity, making it less expensive than traditional slicing at the statement level. The almost identical source code of the mini-app leads to almost identical execution behavior as the original application.

We applied our approach to extract a mini-app with $\approx 1{,}100$ lines of code from a simulation of ≈ 8.5 million lines of code. Thereafter, we used the extracted mini-app to perform a brief performance analysis and subsequently applied the tool DiscoPoP to identify potential parallelism. We implemented a promising subset of the shared-memory parallelization opportunities proposed by DiscoPoP and arrived at a speed-up of $\approx 35\,\%$ for the mini-app. We successfully transferred the parallelization back to the original application and achieved $\approx 30\,\%$ speedup. The transfer was straight-forward as the kernel source-code was the same between the mini-app and the original application.

Altogether, our work leads to a toolchain that automates important relevant parts of performance analysis and exploration, thereby enabling broader applicability of this efficiency-enhancing work with the limited human resources at hand.

8. Future Work

Around the different aspects of this thesis several avenues for future research opened up. Some are of fundamental nature, others more technical.

MetaCG While we worked on MetaCG, we realized that the systematic evaluation of different CG construction tools is almost non-existent. This is particularly true for the languages *C* and *C*++, despite their broad usage. Hence, it is of interest to investigate high-level and language-agnostic description possibilities for CGs and implement an integrated evaluation and benchmark suite for CG construction. The evaluation should include criteria for completeness, e.g., *missed edges*, and precision, e.g., *over approximation*, but could extend to practical aspects, such as multi-TU support. From high-level descriptions the actual test cases can be generated for different languages, significantly broadening the targeted community. Together with a well-defined evaluation methodology, similar to [78], [83], this allows for an insightful comparison of different CG construction tools and approaches. Of course, language-specific challenges, such as reflection in Java, need to be included and reasonable abstractions and mechanisms need to be found.

For MetaCG itself, including the explicit modeling of edges to distinguish between virtual call edges and direct call edges is of high interest. Including this distinction eliminates certain cases of over approximation w.r.t. the potential call targets and improves the precision. Similarly, the alias computation and, thus, the function pointer handling, in particular across TU boundaries, could be improved.

PIRA Currently, PIRA uses simple static heuristics to determine which functions to instrument. Hence, it is of interest to develop and evaluate more elaborate static selection heuristics, such as a global loop depth. Such additional heuristics could potentially improve the number of functions mistakenly selected, or reduce the number of functions that would be subsequently filtered. Improvements, such as recursion detection, and the subsequent use of call-site instead of function-level instrumentation may significantly improve measurement overheads for the respective situations. Similarly, hybrid analysis and instrumentation mechanisms, i.e., recompilation of the target application only every

i-th PIRA iteration, can reduce the total PIRA time required. This is of particular interest when the compilation of the target application takes considerably longer than the execution, as it was the case in the ISSM load-imbalance detection experiment.

The combination of the already existing (and newly developed heuristics), into a combined cost-model also looks promising. Such a cost assessment would allow an analyst to run a single PIRA experiment and receive multiple insights about the target application's behavior. Consequently, the analyst is not required to run multiple independent PIRA experiments, meaning that less waiting time and fewer compute resources are needed. It is also worth exploring whether data-analytics and machine-learning approaches can be used to improve the heuristics, whenever past performance data is already available.

Paradigm- and library-specific heuristics are further interesting areas of research. For example, OpenMP constructs can be used to guide hot-spot and scalability heuristics, given that parallelization typically is used in areas where considerable work is performed or expected. However, also awareness of calls to scientific libraries, such as PETSc [10], may help the analyzer to determine where to place instrumentation.

Mini-AppEx In this thesis, we applied our mini-app extraction approach to only one use case. While this showed the potential of the approach, it is of great interest to use it with more target applications. Of particular interest are applications that already include parallelism, for example using MPI or OpenMP, as in these cases, the extraction approach needs to maintain the communication behavior of the original application. Otherwise, performance analysis insight may not be transferrable from the mini-app to the original application. In addition, the validation of the application's results is of significant interest.

Similarly, a fully-automatic approach to the validation of the mini-app's representativeness is worth pursuing. While the validation step presented in this thesis used data-analytic techniques for data clustering and visualization, the final interpretation was left to the human analyst. Determining reasonable and relevant measures to application and execution similarity as well as their automatic interpretation and rating need to be developed.

Improvements to the source-to-source translator might include a better handling of multi-translation-unit programs for extraction, as well as significant support for *C*++ and its modern standards. As part thereof, additional meta information in MetaCG seem appropriate, e.g., export of the call sites to a particular function as metadata to improve subsequent analysis precision.

A. The Influence of Measurement

The tables present the data captured for the different hardware performance counter measurements. We list only HWPC events mentioned in the main matter for brevity. All raw data can be provided upon request, and will be made publicly available once the final version of the thesis is published.

Benchmark	Flavor	Runtime	BR_MSP	BR_CN
403.gcc	finstr	1.57	3.34	1.00
	scorep	14.04	3.75	13.08
	hpct500	1.02	1.02	1.01
	hpct1000	1.02	1.01	1.00
429.mcf	finstr	1.26	1.01	1.03
	scorep	5.83	0.99	10.04
	hpct500	1.01	1.00	1.00
	hpct1000	1.01	1.00	1.00
433.milc	finstr	1.05	3.17	1.00
	scorep	3.25	1.56	11.89
	hpct500	1.02	1.37	1.01
	hpct1000	1.01	1.19	1.01
444.namd	finstr	1.05	0.99	1.00
	scorep	3.43	1.00	7.61
	hpct500	1.01	1.00	1.01
	hpct1000	1.00	1.00	1.00
447.dealII	finstr	14.24	41.06	1.12
	scorep	367.28	33.97	468.25
	hpct500	1.02	1.02	1.00
	hpct1000	1.01	1.01	1.00
450.soplex	finstr	3.11	3.77	1.00
	scorep	57.34	1.69	80.39
	hpct500	1.00	1.00	1.00
	hpct1000	1.00	1.00	1.00
453.povray	finstr	4.09	14.52	1.02
	scorep	50.80	6.98	68.91
	hpct500	1.02	1.01	1.01
	hpct1000	1.01	1.01	1.00

Table A.1.: (1/2) Perturbation on runtime and branching-behavior related HWPC for the different measurement techniques on Sandy Bridge processors using GCC 4.9.4 and Score-P 3.0, as well as HPCToolkit with 500 and 1000 samples per second, respectively.

Benchmark	Flavor	Runtime	BR_MSP	BR_CN
456.hmmer	finstr	1.01	1.00	1.00
	scorep	1.20	1.00	1.77
	hpct500	1.01	1.01	1.00
	hpct1000	1.00	1.00	1.00
458.sjeng	finstr	1.56	2.05	1.00
	scorep	8.25	1.70	7.03
	hpct500	1.01	1.01	1.01
	hpct1000	1.01	1.00	1.00
462.libquantum	finstr	1.01	1.17	1.00
	scorep	1.47	1.02	1.30
	hpct500	1.01	1.01	1.00
	hpct1000	1.01	1.00	1.00
464.h264ref	finstr	1.51	4.41	1.00
	scorep	12.76	2.59	16.44
	hpct500	1.02	1.02	1.01
	hpct1000	1.01	1.01	1.00
470.lbm	finstr	1.00	1.02	1.00
	scorep	1.00	1.15	1.00
	hpct500	1.01	1.16	1.03
	hpct1000	1.00	1.09	1.01
473.astar	finstr	1.91	1.21	0.94
	scorep	24.68	1.45	44.18
	hpct500	1.01	1.00	1.00
	hpct1000	1.01	1.00	1.00
482.sphinx3	finstr	1.05	1.11	1.00
	scorep	2.57	1.05	2.91
	hpct500	1.01	1.01	1.00
	hpct1000	1.01	1.00	1.00

Table A.2.: (2/2) Perturbation on runtime and branching-behavior related HWPC for the different measurement techniques on Sandy Bridge processors using GCC 4.9.4 and Score-P 3.0, as well as HPCToolkit with 500 and 1000 samples per second, respectively.

Benchmark	Flavor	Runtime	TOT_INS	TOT_CYC	BR_MSP	BR_CN
403.gcc	finstr	1.41	1.32	1.42	1.10	1.00
	scorep-no-filter	13.25	19.96	12.62	4.29	12.82
429.mcf	finstr	1.17	1.69	1.17	0.97	1.03
	scorep-no-filter	4.94	20.18	4.96	1.01	10.04
433.milc	finstr	1.04	1.11	1.04	1.05	1.00
	scorep-no-filter	3.02	5.77	3.03	1.92	11.79
444.namd	finstr	1.06	1.11	1.06	0.99	1.00
	scorep-no-filter	3.35	4.04	3.36	1.01	7.60
447.dealII	finstr	12.41	5.95	12.53	4.93	1.12
	scorep-no-filter	396.61	342.41	400.07	36.54	466.61
450.soplex	finstr	2.92	2.71	2.93	1.22	1.00
	scorep-no-filter	54.69	106.87	55.02	1.81	80.59
453.povray	finstr	3.32	1.92	3.41	2.62	1.02
	scorep-no-filter	54.53	60.73	56.43	15.05	70.19
456.hmmer	finstr	1.02	1.01	1.02	1.00	1.00
	scorep-no-filter	1.16	1.38	1.16	0.99	1.78
458.sjeng	finstr	1.30	1.18	1.30	0.95	1.00
	scorep-no-filter	7.98	11.45	7.96	1.82	7.03
462.libquantum	finstr	1.01	1.01	1.01	1.01	1.00
	scorep-no-filter	1.41	1.51	1.42	0.96	1.30
464.h264ref	finstr	1.35	1.26	1.35	1.20	1.00
	scorep-no-filter	11.82	11.16	11.84	2.34	16.87
470.lbm	finstr	1.00	1.00	1.00	1.00	1.00
	scorep-no-filter	1.00	1.00	1.00	1.00	1.00
473.astar	finstr	1.91	2.06	1.94	1.05	0.94
	scorep-no-filter	22.10	60.50	22.19	1.61	44.33
482.sphinx3	finstr	1.05	1.04	1.05	0.96	1.00
	scorep-no-filter	2.53	2.79	2.53	0.96	2.88

Table A.3.: The influence of GCC 4.9.4 and Score-P 3.0 w/o **inline** filtering on runtime, total instructions (TOT_INS), total cycles (TOT_CYC), mispredicted branches (BR_MSP), and conditional branches (BR_CN).

Benchmark	Flavor	Runtime	TOT_INS	TOT_CYC	BR_MSP	BR_CN
403.gcc	scorep-no-filter	13.72	20.73	13.20	4.41	12.48
	scorep	11.26	17.94	10.87	4.01	9.93
429.mcf	scorep-no-filter	5.36	21.52	5.33	1.77	10.21
	scorep	4.84	20.24	4.81	1.67	9.62
433.milc	scorep-no-filter	3.07	6.10	3.07	1.46	11.86
	scorep	3.02	5.77	3.02	1.56	11.05
444.namd	scorep-no-filter	3.43	4.10	3.42	1.04	7.19
	scorep	1.00	1.00	1.00	1.00	1.00
447.dealII	scorep-no-filter	424.01	354.30	436.00	38.20	483.47
	scorep	12.01	12.10	12.16	3.93	11.93
450.soplex	scorep-no-filter	59.06	115.74	59.22	1.69	76.14
	scorep	2.88	4.50	2.87	1.21	3.19
453.povray	scorep-no-filter	59.91	65.84	62.16	17.92	68.21
	scorep	0.00	0.00	0.00	0.00	0.00
456.hmmer	scorep-no-filter	1.11	1.37	1.12	0.99	1.67
	scorep	1.29	1.35	1.29	0.99	1.63
458.sjeng	scorep-no-filter	8.21	11.44	8.19	1.82	6.76
	scorep	7.86	10.81	7.89	1.78	6.42
462.libquantum	scorep-no-filter	1.43	1.52	1.44	1.45	1.28
	scorep	1.25	1.30	1.25	1.73	1.16
464.h264ref	scorep-no-filter	12.49	11.25	12.55	2.94	15.99
	scorep	11.94	10.61	12.01	3.06	15.00
470.lbm	scorep-no-filter	1.00	1.00	1.00	1.00	1.00
	scorep	1.00	1.00	1.00	1.00	1.00
473.astar	scorep-no-filter	23.01	62.35	22.99	1.58	42.32
	scorep	8.23	20.02	8.23	1.57	13.45
482.sphinx3	scorep-no-filter	2.65	2.78	2.66	1.03	2.84
	scorep	2.53	2.65	2.53	1.05	2.66

Table A.4.: The influence of GCC 9.1.0 and Score-P 6.0 w/ and w/o inline filter on runtime, total instructions (TOT_INS), total cycles (TOT_CYC), mispredicted branches (BR_MSP), and conditional branches (BR_CN).

Benchmark	Flavor	Runtime	TOT_INS	TOT_CYC	BR_MSP	BR_CN
403.gcc	scorep-no-filter	13.69	20.78	13.38	4.27	13.15
	scorep	12.09	15.06	9.54	3.80	9.41
429.mcf	scorep-no-filter	5.14	19.12	5.16	1.07	9.39
	scorep	1.19	1.43	1.19	1.01	1.20
433.milc	scorep-no-filter	3.10	7.12	3.12	1.19	32.78
	scorep	3.13	7.09	3.15	1.18	32.54
444.namd	scorep-no-filter	3.65	4.28	3.65	1.02	7.32
	scorep	1.00	1.00	1.00	1.00	1.00
447.dealII	scorep-no-filter	424.63	445.75	419.32	41.09	685.26
	scorep	13.33	15.79	13.41	2.44	18.88
450.soplex	scorep-no-filter	58.98	124.28	59.34	2.18	91.52
	scorep	2.96	5.14	2.96	1.00	4.03
453.povray	scorep-no-filter	0.00	0.00	0.00	0.00	0.00
	scorep	0.00	0.00	0.00	0.00	0.00
456.hmmer	scorep-no-filter	1.26	1.42	1.25	0.97	1.46
	scorep	1.12	1.27	1.11	0.98	1.29
458.sjeng	scorep-no-filter	8.92	12.81	8.96	1.87	7.59
	scorep	7.19	10.09	7.15	1.81	6.06
462.libquantum	scorep-no-filter	1.41	1.51	1.41	0.68	1.29
	scorep	1.26	1.32	1.27	1.09	1.18
464.h264ref	scorep-no-filter	13.11	13.06	13.11	2.72	20.82
	scorep	12.86	12.73	13.18	2.63	20.69
470.lbm	scorep-no-filter	1.00	1.00	1.00	1.00	1.00
	scorep	1.00	1.00	1.00	0.99	1.00
473.astar	scorep-no-filter	23.97	67.55	23.99	1.45	46.39
	scorep	5.08	12.34	5.08	1.14	8.55
482.sphinx3	scorep-no-filter	2.91	3.00	2.92	1.04	3.05
	scorep	2.45	2.48	2.46	1.05	2.52

Table A.5.: The influence for Clang 10.0 with Score-P 6.0 w/ and w/o inline-filter on runtime, total instructions (TOT_INS), total cycles (TOT_CYC), mispredicted branches (BR_MSP), and conditional branches (BR_CN).

Benchmark	Flavor	Runtime	TOT_CYC	TOT_INS
403.gcc	scorep	12.09	9.54	15.06
	scorep-no-filter	13.69	13.38	20.78
429.mcf	scorep	1.19	1.19	1.43
	scorep-no-filter	5.14	5.16	19.12
433.milc	scorep	3.13	3.15	7.09
	scorep-no-filter	3.10	3.12	7.12
444.namd	scorep	1.00	1.00	1.00
	scorep-no-filter	3.65	3.65	4.28
447.dealII	scorep	13.33	13.41	15.79
	scorep-no-filter	424.63	419.32	445.75
450.soplex	scorep	2.96	2.96	5.14
	scorep-no-filter	58.98	59.34	124.28
453.povray	scorep	0.00	0.00	0.00
	scorep-no-filter	0.00	0.00	0.00
456.hmmer	scorep	1.12	1.11	1.27
	scorep-no-filter	1.26	1.25	1.42
458.sjeng	scorep	7.19	7.15	10.09
	scorep-no-filter	8.92	8.96	12.81
462.libquantum	scorep	1.26	1.27	1.32
	scorep-no-filter	1.41	1.41	1.51
464.h264ref	scorep	12.86	13.18	12.73
	scorep-no-filter	13.11	13.11	13.06
470.lbm	scorep	1.00	1.00	1.00
	scorep-no-filter	1.00	1.00	1.00
473.astar	scorep	5.08	5.08	12.34
	scorep-no-filter	23.97	23.99	67.55
482.sphinx3	scorep	2.45	2.46	2.48
	scorep-no-filter	2.91	2.92	3.00

Table A.6.: The influence for Clang 10.0 with Score-P 6.0 w/ and w/o inline-filter on runtime, and basic events for derived metric Instructions per Cycle.

Benchmark	Flavor	Runtime	L2_TCM	L2_TCA	L2_DCM	LST_INS
403.gcc	scorep	12.09	1.42	3.22	1.30	20.32
	scorep-no-filter	13.69	1.59	4.12	1.43	28.27
429.mcf	scorep	1.19	1.04	1.13	1.03	1.57
	scorep-no-filter	5.14	1.15	1.14	1.14	24.90
433.milc	scorep	3.13	0.92	1.06	0.92	9.11
	scorep-no-filter	3.10	0.95	1.06	0.95	9.12
444.namd	scorep	1.00	1.01	0.99	0.95	1.00
	scorep-no-filter	3.65	1.25	1.34	1.16	8.92
447.dealII	scorep	13.33	1.44	1.62	1.28	17.51
	scorep-no-filter	424.63	28.22	266.21	24.86	477.07
450.soplex	scorep	2.96	0.96	1.11	0.96	5.92
	scorep-no-filter	58.98	0.70	2.38	0.68	148.83
453.povray	scorep	0.00	0.00	0.00	0.00	0.00
	scorep-no-filter	0.00	0.00	0.00	0.00	0.00
456.hmmer	scorep	1.12	1.36	0.91	1.39	1.24
	scorep-no-filter	1.26	1.42	0.87	1.36	1.37
458.sjeng	scorep	7.19	2.85	14.48	2.68	14.95
	scorep-no-filter	8.92	3.91	18.13	3.47	19.13
462.libquantum	scorep	1.26	1.00	1.02	1.00	1.68
	scorep-no-filter	1.41	1.01	0.95	1.00	2.10
464.h264ref	scorep	12.86	1.10	1.77	0.97	13.28
	scorep-no-filter	13.11	1.12	2.00	1.01	13.65
470.lbm	scorep	1.00	1.01	1.00	1.01	1.00
	scorep-no-filter	1.00	1.00	1.00	1.00	1.00
473.astar	scorep	5.08	0.91	1.05	0.90	14.82
	scorep-no-filter	23.97	0.83	1.09	0.83	81.14
482.sphinx3	scorep	2.45	1.08	1.10	1.04	3.58
	scorep-no-filter	2.91	1.10	1.16	1.06	4.49

Table A.7.: The influence for Clang 10.0 with Score-P 6.0 w/ and w/o inline-filter on runtime, level two total cache misses (L2_TCM), level two total cache accesses (L2_TCA), level two data cache misses (L2_DCM), and lost/store instruction (LST_INS).

Benchmark	Flavor	Runtime	L3_TCM	L3_TCA	L2_DCA
403.gcc	scorep	12.09	1.02	1.42	3.31
	scorep-no-filter	13.69	1.01	1.58	4.35
429.mcf	scorep	1.19	1.02	1.03	1.13
	scorep-no-filter	5.14	1.09	1.14	1.13
433.milc	scorep	3.13	0.99	0.92	1.06
	scorep-no-filter	3.10	1.03	0.95	1.06
444.namd	scorep	1.00	1.01	0.99	0.99
	scorep-no-filter	3.65	1.05	1.23	1.33
447.dealII	scorep	13.33	1.13	1.39	1.55
	scorep-no-filter	424.63	1.42	26.70	269.09
450.soplex	scorep	2.96	1.03	0.96	1.10
	scorep-no-filter	58.98	1.05	0.70	2.33
453.povray	scorep	0.00	1.35	0.00	0.00
	scorep-no-filter	0.00	1.46	0.00	0.00
456.hmmer	scorep	1.12	38.06	1.39	0.90
	scorep-no-filter	1.26	37.37	1.39	0.86
458.sjeng	scorep	7.19	1.01	2.91	13.71
	scorep-no-filter	8.92	1.00	3.82	17.42
462.libquantum	scorep	1.26	0.99	1.00	1.02
	scorep-no-filter	1.41	1.01	1.00	0.94
464.h264ref	scorep	12.86	4.17	1.07	1.63
	scorep-no-filter	13.11	3.74	1.11	1.88
470.lbm	scorep	1.00	1.00	1.01	1.00
	scorep-no-filter	1.00	1.00	1.00	1.00
473.astar	scorep	5.08	0.62	0.91	1.05
	scorep-no-filter	23.97	0.53	0.83	1.08
482.sphinx3	scorep	2.45	1.12	1.05	1.09
	scorep-no-filter	2.91	1.27	1.07	1.15

Table A.8.: The influence for Clang 10.0 with Score-P 6.0 w/ and w/o inline-filter on runtime, level three total cache misses (L3_TCM), level three total cache accesses (L3_TCA), and level two data cache accesses (L2_DCA).

List of Figures

List of Tables

Listings

Acronyms

API application programming interface. 7, 11, 13, 15–17, 22, 24, 27, 32, 33, 43, 49, 93

AST abstract syntax tree. 5, 7, 8, 14, 40, 43, 45, 46, 91, 92, 123

CFG control-flow graph. 7, 10, 11, 119

CG call graph. 5, 7, 9, 10, 16, 19, 26, 40, 41, 43–45, 47–49, 61–63, 69, 70, 78, 79, 91, 106–108, 119

CPR checkpoint/restart. 22, 89, 91–94, 96, 99, 107

HPC high-performance computing. 1–5, 11, 19, 20, 69

HWPC hardware performance counter. 2, 3, 11, 15, 18–20, 24, 27–30, 32, 33, 36, 38, 106, 110–112, 121, 122

IC instrumentation configuration. 57, 58, 60–64, 69, 71, 75, 76, 78, 79, 81, 88

IPC Instructions per Cycle. 34, 35, 116

IR intermediate representation. 4, 7, 14, 17, 18

ISSM Ice-sheet and Sea-level System Model. 50, 51, 55, 88, 109

JIT Just in Time. 17, 18

MPI Message Passing Interface. 22, 49–51, 53, 55–57, 60, 65, 68, 79, 81–83, 85, 87, 88, 93, 105, 107, 109, 121

PGIS Profile Guided Instrumentation Selection. 6, 48, 51, 59–63, 66, 68, 69, 79

TU translation unit. 41, 44–47, 49, 55, 62, 91, 107, 108

Bibliography

[1] O. Aaziz, J. Cook, J. Cook, *et al.*, "A methodology for characterizing the correspondence between real and proxy applications", in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2018, pp. 190–200.

[2] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Pearson Education, 2004.

[3] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems", in *2011 33rd Intl. Conference on Software Engineering (ICSE)*, ACM, 2011, pp. 746–755. DOI: 10.1145/1985793.1985898.

[4] J. Adam and S. Kell, "Type Checking beyond Type Checkers, via Slice & Run", in *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis*, ser. TAPAS 2020, Virtual, USA: Association for Computing Machinery, 2020, pp. 23–29, ISBN: 9781450381895. DOI: 10.1145/3427764.3428324.

[5] L. Adhianto, S. Banerjee, M. Fagan, *et al.*, "Hpctoolkit: Tools for performance analysis of optimized parallel programs", *Concurrency and Computation: Practice and Experience.*, vol. 22, no. 6, pp. 685–701, 2010.

[6] F. E. Allen, "Control flow analysis", *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs", in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88, San Diego, California, USA: Association for Computing Machinery, 1988, pp. 1–11, ISBN: 0897912527. DOI: 10.1145/73560.73561.

[8] G. Antoniol, F. Calzolari, and P. Tonella, "Impact of function pointers on the call graph", in *Third European Conference on Software Maintenance and Reengineering*, 1999, pp. 51–59. DOI: 10.1109/CSMR.1999.756682.

[9] P. Arzt, Y. Fischler, J.-P. Lehr, and C. Bischof, “Automatic Low-Overhead Load-Imbalance Detection in MPI Applications”, in *Euro-Par 2021: Parallel Processing*, L. Sousa, N. Roma, and P. Tomás, Eds., Cham: Springer International Publishing, 2021, pp. 19–34, ISBN: 978-3-030-85665-6.

[10] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, “Efficient management of parallelism in object oriented numerical software libraries”, in *Modern Software Tools in Scientific Computing*, Birkhäuser Press, 1997, pp. 163–202. DOI: 10.1007/978-1-4612-1986-6_8.

[11] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, *et al.*, “FTI: High performance fault tolerance interface for hybrid systems”, in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11, Seattle, Washington: ACM, 2011, ISBN: 9781450307710. DOI: 10.1145/2063384.2063427.

[12] D. M. Berris, A. Veitch, N. Heintze, E. Anderson, and N. Wang, “Xray: A function call tracing system”, 2016.

[13] A. Bhatele, S. Brink, and T. Gamblin, “Hatchet: Pruning the overgrowth in parallel profiles”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290. DOI: 10.1145/3295500.3356219.

[14] A. Bhattacharyya and T. Hoefler, “Pemogen: Automatic adaptive performance modeling during program runtime”, in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT ’14, Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 393–404, ISBN: 9781450328098. DOI: 10.1145/2628071.2628100.

[15] C. Bischof, D. an Mey, and C. Iwainsky, “Brainware for green hpc”, *Computer Science - Research and Development*, vol. 27, no. 4, pp. 227–233, Nov. 2012, ISSN: 1865-2042. DOI: 10.1007/s00450-011-0198-5.

[16] D. Boehme, T. Gamblin, D. Beckingsale, *et al.*, “Caliper: Performance introspection for HPC software stacks”, in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2016. DOI: 10.1109/sc.2016.46.

[17] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf, “Identifying the Root Causes of Wait States in Large-Scale Parallel Applications”, *ACM Transactions on Parallel Computing*, vol. 3, no. 2, 11:1–11:24, 2016, ISSN: 2329-4949. DOI: 10.1145/2934661.

[18] S. Browne, “A portable programming interface for performance evaluation on modern processors”, *Intl. Journal of High Performance Computing Applications.*, vol. 14, no. 3, pp. 189–204, 2000. DOI: 10.1177/109434200001400303.

[19] D. Bruening and S. Amarasinghe, “Efficient, transparent, and comprehensive runtime code manipulation”, Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[20] B. R. Buck, “An API for runtime code patching”, *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000. DOI: 10.1177/109434200001400404.

[21] M. Burger, G. N. Nguyen, and C. Bischof, “Developing models for the runtime of programs with exponential runtime behavior”, in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, S. A. Wright, S. A. Jarvis, and S. D. Hammond, Eds., 2020, pp. 109–125. DOI: 10.1109/PMBS51919.2020.00015.

[22] A. Calotoiu, D. Beckinsale, C. W. Earl, *et al.*, “Fast multi-parameter performance modeling”, in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 172–181.

[23] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, “Using automated performance modeling to find scalability bugs in complex codes”, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13, Denver, Colorado: ACM, 2013, 45:1–45:12, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503277.

[24] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems”, in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’04, Boston, MA: USENIX Association, 2004, p. 2.

[25] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, “CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization”, *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, 6:1–6:24, Apr. 2015, ISSN: 1544-3566. DOI: 10.1145/2724717.

[26] C. Cifuentes and D. Simon, “Procedure abstraction recovery from binary code”, in *Proc. of the 4th Europ. Software Maintenance and Reengineering.*, 2000, pp. 55–64. DOI: `10.1109/CSMR.2000.827306`.

[27] M. Copik, A. Calotoiu, T. Grosser, *et al.*, “Extracting clean performance models from tainted programs”, in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 403–417, ISBN: 9781450382946. DOI: `10.1145/3437801.3441613`.

[28] A. Daniel, *Cloc github repository*, 2006–2020. [Online]. Available: `https://github.com/AlDanial/cloc`.

[29] L. DeRose, B. Homer, and D. Johnson, “Detecting Application Load Imbalance on High End Massively Parallel Systems”, in *Euro-Par 2007 Parallel Processing*, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 150–159, ISBN: 978-3-540-74466-5. DOI: `10.1007/978-3-540-74466-5\_17`.

[30] L. Djoudi, D. Barthou, P. Carribault, *et al.*, “Maqao: Modular assembler quality analyzer and optimizer for itanium 2”, in *The 4th Workshop on EPIC architectures and compiler technology.*, vol. 200, 2005.

[31] C. Drischler, K. Hebeler, and A. Schwenk, “Chiral interactions up to next-to-next-to-next-to-leading order and nuclear saturation”, *Physical Review Letters*, vol. 122, p. 042 501, 4 Jan. 2019. DOI: `10.1103/PhysRevLett.122.042501`.

[32] T. D. Economon, F. Palacios, S. R. Copeland, T. W. Lukaczyk, and J. J. Alonso, “Su2: An open-source suite for multiphysics simulation and design”, *Aiaa Journal*, vol. 54, no. 3, pp. 828–846, 2015.

[33] Y. Fischler, M. Rückamp, C. Bischof, *et al.*, “A scalability study of the ice-sheet and sea-level system model (issm, version 4.18)”, *Geoscientific Model Development Discussions*, vol. 2021, pp. 1–33, 2021. DOI: `10.5194/gmd-2021-265`. [Online]. Available: `https://gmd.copernicus.org/preprints/gmd-2021-265/`.

[34] T. Gamblin, *wrap.py – A PMPI Wrapper*. [Online]. Available: `https://github.com/LLNL/wrap`.

[35] M. Geimer, B. Kuhlmann, F. Pulatova, F. Wolf, and B. J. N. Wylie, “Scalable collation and presentation of call-path profile data with CUBE”, in *Proc. of the Conference on Parallel Computing (ParCo), Aachen/Jülich, Germany, Minisymposium Scalability and Usability of HPC Programming Tools*, 2007, pp. 645–652.

[36] M. Geimer, F. Wolf, B. J. N. Wylie, *et al.*, “The Scalasca performance toolset architecture”, *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010. DOI: 10.1002/cpe.1556.

[37] Z. Gong, Z. Chen, J. Szaday, *et al.*, “An empirical study of the effect of source-level loop transformations on compiler stability”, *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276496.

[38] B. Gough, *GNU Scientific Library Reference Manual - Third Edition*, 3rd. Network Theory Ltd., 2009, ISBN: 0954612078.

[39] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof: A call graph execution profiler”, *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982, ISSN: 0362-1340. DOI: 10.1145/872726.806987.

[40] A. Griewank and A. Walther, *Evaluating Derivatives*, Second. SIAM, 2008. DOI: 10.1137/1.9780898717761.

[41] T. Hahn, “Cuba – a library for multidimensional numerical integration”, *Computer Physics Communications*, vol. 168, no. 2, pp. 78–95, 2005, ISSN: 0010-4655.

[42] J. L. Henning, “SPEC CPU2006 benchmark descriptions”, *ACM SIGARCH Computer Architecture News.*, vol. 34, no. 4, pp. 1–17, 2006. DOI: 10.1145/1186736.1186737.

[43] T. Hoefler and R. Belli, “Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results”, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’15, Austin, Texas: Association for Computing Machinery, 2015, ISBN: 9781450337236. DOI: 10.1145/2807591.2807644.

[44] J. K. Hollingsworth and B. P. Miller, “Dynamic control of performance monitoring on large scale parallel systems”, in *Proceedings of the 7th international conference on Supercomputing*, ACM, 1993, pp. 185–194.

[45] J. Hubicka, *The gcc call graph module: A framework for inter-procedural optimization*, 2004.

[46] A. Hück, “Compiler support for operator overloading and algorithmic differentiation in c++”, Ph.D. dissertation, Technische Universität, Darmstadt, 2020.

[47] A. Hück, C. Bischof, and J. Utke, “Checking C++ Codes for Compatibility with Operator Overloading”, in *15th IEEE International Working Conference on Source Code Analysis and Manipulation*, vol. 15, IEEE, 2015, pp. 91–100. DOI: 10.1109/SCAM.2015.7335405.

[48] A. Hück, J.-P. Lehr, S. Kreutzer, *et al.*, "Compiler-aided Type Tracking for Correctness Checking of MPI Applications", in *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, Nov. 2018, pp. 51–58. DOI: `10.1109/Correctness.2018.00011`.

[49] A. Hück, J. Protze, J.-P. Lehr, *et al.*, "Compiler-aided type tracking for correctness checking of adjoint MPI applications", in *2020 IEEE/ACM 4th Intl. Workshop on Software Correctness for HPC Applications (Correctness)*, IEEE, Nov. 2020, ISBN: 978-0-7381-1045-5. DOI: `10.1109/Correctness51934.2020.00010`.

[50] K. Ilyas, A. Calotoiu, and F. Wolf, "Off-road performance modeling – how to deal with segmented data", in *Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain*, ser. Lecture Notes in Computer Science, Springer, Aug. 2017, pp. 1–12.

[51] Intel, *Intel 64 and ia-32 architectures optimization reference manual*, Jun. 2021. [Online]. Available: `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html`.

[52] Intel, *Intel vTune website*, 2021. [Online]. Available: `http://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html`.

[53] C. Iwainsky, *InstRO – a framework for building tailored compiler instrumentation*, 2015. [Online]. Available: `https://github.com/InstRO/InstRO`.

[54] C. Iwainsky, "InstRO: A component-based toolbox for performance instrumentation", Technische Universität Darmstadt, 2015, Ph.D. dissertation, Technical University of Darmstadt, Aachen, 2016, ISBN: 978-3-8440-4562-8.

[55] C. Iwainsky, R. Altenfeld, D. an Mey, and C. Bischof, "Enhancing brainware productivity through a performance tuning workflow", in *Euro-Par 2011: Parallel Processing Workshops.*, Springer, 2011, pp. 198–207.

[56] C. Iwainsky and C. Bischof, "Call tree controlled instrumentation for low-overhead survey measurements", in *2016 IEEE Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW 2016)*, 2016.

[57] C. Iwainsky, J.-P. Lehr, and C. Bischof, "Compiler Supported Sampling through Minimalistic Instrumentation", in *2014 43rd Intl. Conf. on Parallel Processing Workshops*, Institute of Electrical & Electronics Engineers (IEEE), 2014. DOI: `10.1109/icppw.2014.33`.

[58] C. Iwainsky, S. Shudler, A. Calotoiu, *et al.*, “How many threads will be too many? on the scalability of openmp implementations”, in *European Conference on Parallel Processing*, Springer, 2015, pp. 451–463.

[59] T. Jammer, C. Iwainsky, and C. Bischof, “Automatic detection of mpi assertions”, in *High Performance Computing*, H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, Eds., Cham: Springer International Publishing, 2020, pp. 34–42, ISBN: 978-3-030-59851-8.

[60] I. Karlin, J. Keasler, and R. Neely, “Lulesh 2.0 updates and changes”, Lawrence Livermore National Lab, Livermore, CA, Tech. Rep. LLNL-TR-641973, Aug. 2013, pp. 1–9. [Online]. Available: `https://asc.llnl.gov/sites/asc/files/2021-01/lulesh2.0_changes1.pdf`.

[61] G. Kiczales, J. Lamping, A. Mendhekar, *et al.*, “Aspect-oriented programming”, in *ECOOP’97 — Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242, ISBN: 978-3-540-69127-3.

[62] Y. Kim, J. Dennis, C. Kerr, *et al.*, “Kgen: A python tool for automated fortran kernel generation and verification”, *Procedia Computer Science*, vol. 80, pp. 1450–1460, 2016, International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2016.05.466`.

[63] J. Kinder and H. Veith, “Precise static analysis of untrusted driver binaries”, in *Formal Methods in Computer-Aided Design (FMCAD), 2010*, 2010, pp. 43–50.

[64] A. Knüpfer, C. Rössel, D. a. Mey, *et al.*, “Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir”, in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds., Berlin, Heidelberg: Springer, 2012, pp. 79–91, ISBN: 978-3-642-31476-6. DOI: `10.1007/978-3-642-31476-6_7`.

[65] P. Kogge and J. Shalf, “Exascale computing trends: Adjusting to the ”new normal”’ for computer architecture”, *Computing in Science Engineering*, vol. 15, no. 6, pp. 16–26, 2013. DOI: `10.1109/MCSE.2013.95`.

[66] W. Korn, P. J. Teller, and G. Castillo, “Just how accurate are performance counters?”, in *IEEE Intl. Conf. on Performance, Computing, and Communications, 2001.*, 2001, pp. 303–310. DOI: `10.1109/IPCCC.2001.918667`.

[67] V. Kutscher, S. Ruland, P. Müller, *et al.*, “Towards a circular economy of industrial software”, *Procedia CIRP*, vol. 90, pp. 37–42, 2020, ISSN: 2212-8271. DOI: 10.1016/j.procir.2020.01.133.

[68] E. Larour, H. Seroussi, M. Morlighem, and E. Rignot, “Continental scale, high order, high spatial resolution, ice sheet modeling using the Ice Sheet System Model (ISSM)”, *Journal of Geophysical Research: Earth Surface*, vol. 117, no. F1, 2012. DOI: 10.1029/2011JF002140.

[69] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, 2004.

[70] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, “Pebil: Efficient static binary instrumentation for linux”, in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 175–183. DOI: 10.1109/ISPASS.2010.5452024.

[71] J.-P. Lehr, “Counting Performance: Hardware Performance Counter and Compiler Instrumentation”, in *INFORMATIK 2016. Lecture Notes in Informatics (LNI)*, Erscheinungstermin: 26.09.2016, Gesellschaft für Informatik (GI), vol. P-259, Bonn: Heinrich C. Mayr, Martin Pinzger, 2016.

[72] J.-P. Lehr, C. Bischof, F. Dewald, *et al.*, “Tool-Supported Mini-App Extraction to Facilitate Program Analysis and Parallelization”, in *50th International Conference on Parallel Processing*. New York, NY, USA: Association for Computing Machinery, 2021, ISBN: 9781450390682. DOI: 10.1145/3472456.3472521.

[73] J.-P. Lehr, A. Calotoiu, C. Bischof, and F. Wolf, “Automatic Instrumentation Refinement for Empirical Performance Modeling”, in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, IEEE, 2019, pp. 40–47. DOI: 10.1109/ProTools49597.2019.00011.

[74] J.-P. Lehr, A. Hück, and C. Bischof, “PIRA: Performance Instrumentation Refinement Automation”, in *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, ser. AI-SEPS 2018, Boston, MA, USA: ACM, 2018, pp. 1–10, ISBN: 978-1-4503-6067-8. DOI: 10.1145/3281070.3281071.

[75] J.-P. Lehr, A. Hück, M. Fischer, and C. Bischof, “Compiler-assisted type-safe checkpointing”, in *ISC High Performance 2020 Workshops*, Springer International Publishing, 2020, ISBN: 978-3-030-59851-8. DOI: 10.1007/978-3-030-59851-8_1.

[76] J.-P. Lehr, A. Hück, Y. Fischler, and C. Bischof, “MetaCG: Annotated Call-Graphs to Facilitate Whole-Program Analysis”, in *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis*. New York, NY, USA: ACM, 2020, pp. 3–9, ISBN: 9781450381895.

[77] J.-P. Lehr, C. Iwainsky, and C. Bischof, “The Influence of HPCToolkit and Score-p on Hardware Performance Counters”, in *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems*, ser. SEPS 2017, Vancouver, BC, Canada: ACM, 2017, pp. 21–30, ISBN: 978-1-4503-5517-9. DOI: 10.1145/3141865.3141869.

[78] J.-P. Lehr, T. Jammer, and C. Bischof, “MPI-CorrBench: Towards an MPI Correctness Benchmark Suite”, in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’21, Virtual Event, Sweden: ACM, 2021, pp. 69–80, ISBN: 9781450382175. DOI: 10.1145/3431379.3460652.

[79] O. Lhoták, “Comparing call graphs”, in *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE ’07, ACM, 2007, pp. 37–42, ISBN: 9781595935953. DOI: 10.1145/1251535.1251542.

[80] Z. Li, R. Atre, Z. U. Huda, A. Jannesari, and F. Wolf, “Unveiling parallelization opportunities in sequential programs”, *Journal of Systems and Software*, vol. 117, pp. 282–295, Jul. 2016. DOI: 10.1016/j.jss.2016.03.045.

[81] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, “Effective source-to-source outlining to support whole program empirical optimization”, in *Languages and Compilers for Parallel Computing*, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 308–322, ISBN: 978-3-642-13374-9.

[82] P. T. Lin, M. A. Heroux, R. F. Barrett, and A. B. Williams, “Assessing a mini-application as a performance proxy for a finite element method engineering application”, *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5374–5389, 2015. DOI: 10.1002/cpe.3587.

[83] P.-H. Lin, C. Liao, M. Schordan, and I. Karlin, “Exploring Regression of Data Race Detection Tools Using DataRaceBench”, in *IEEE/ACM 3rd Intl. Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 11–18. DOI: 10.1109/Correctness49594.2019.00007.

[84] K. A. Lindlan, J. Cuny, A. D. Malony, *et al.*, “A tool framework for static and dynamic analysis of object-oriented software with templates”, in *Proc. ACM/IEEE 2000 Conf. Supercomputing*, Nov. 2000, p. 49. DOI: 10.1109/SC.2000.10052.

[85] C.-K. Luk, R. Cohn, R. Muth, *et al.*, “Pin: Building customized program analysis tools with dynamic instrumentation”, *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, 2005, ISSN: 0362-1340. DOI: 10.1145/1064978.1065034.

[86] J. R. Madsen, M. G. Awan, H. Brunie, *et al.*, “Timemory: Modular performance analysis for hpc”, in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., Cham: Springer International Publishing, 2020, pp. 434–452, ISBN: 978-3-030-50743-5.

[87] A. D. Malony, “Event-based performance perturbation: A case study”, in *Proc. of the 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming.*, ser. PPOPP ’91, Williamsburg, Virginia, USA: ACM, 1991, pp. 201–212, ISBN: 0-89791-390-6. DOI: 10.1145/109625.109646.

[88] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, “Performance measurement intrusion and perturbation analysis”, *IEEE Transactions on Parallel and Distributed Systems.*, vol. 3, no. 4, pp. 433–450, 1992, ISSN: 1045-9219. DOI: 10.1109/71.149962.

[89] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, “On the performance portability of structured grid codes on many-core computer architectures”, in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds., Cham: Springer International Publishing, 2014, pp. 53–75, ISBN: 978-3-319-07518-1.

[90] S. McIntosh-Smith, M. Martineau, T. Deakin, *et al.*, “Tealeaf: A mini-application to enable design-space explorations for iterative sparse linear solvers”, in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 842–849.

[91] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, “A performance analysis of the first generation of hpc-optimized arm processors”, *Concurrency and Computation: Practice and Experience*, vol. 31, no. 16, e5110, 2019, e5110 cpe.5110. DOI: 10.1002/cpe.5110.

[92] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, 2015. [Online]. Available: www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[93] B. P. Miller, M. D. Callaghan, J. M. Cargille, *et al.*, “The paradyn parallel performance measurement tool”, *Computer*, vol. 28, no. 11, pp. 37–46, Nov. 1995, ISSN: 0018-9162. DOI: 10.1109/2.471178.

[94] N. Morew, M. Norouzi, A. Jannesari, and F. Wolf, "Skipping non-essential instructions makes data-dependence profiling faster", in *Euro-Par 2020: Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 12247, Springer, 2020, ISBN: 978-3-030-57674-5. DOI: 10.1007/978-3-030-57675-2_1.

[95] A. Morris, A. D. Malony, S. Shende, and K. Huck, "Design and implementation of a hybrid parallel performance measurement system", in *39th Intl. Conf. on Parallel Processing.*, Institute of Electrical & Electronics Engineers (IEEE), 2010. DOI: 10.1109/icpp.2010.57.

[96] D. Mosberger *et al.*, *The libunwind project*, 2011.

[97] T. Moseley, D. Grunwald, and R. Peri, "Chainsaw: Using binary matching for relative instruction mix comparison", in *18th Intl. Conf. on Parallel Architectures and Compilation Techniques, PACT.*, 2009, pp. 125–135. DOI: 10.1109/PACT.2009.12.

[98] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 158–191, 1998. DOI: 10.1145/279310.279314.

[99] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis", in *Euro-Par 2011 Parallel Processing*, Springer, Jan. 1, 2011, ISBN: 978-3-642-23399-9. DOI: 10.1007/978-3-642-23400-2_7.

[100] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!", in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, Washington, DC, USA: ACM, 2009, pp. 265–276, ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508275.

[101] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources", *Supercomputer*, vol. 12, no. 63, pp. 69–80, Jan. 1996.

[102] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation", in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100, ISBN: 9781595936332. DOI: 10.1145/1250734.1250746.

[103] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, “VeloC: Towards high performance adaptive asynchronous checkpointing at large scale”, in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 911–920. DOI: 10.1109/IPDPS.2019.00099.

[104] M. Norouzi, Q. Ilias, A. Jannesari, and F. Wolf, “Accelerating data-dependence profiling with static hints”, in *Euro-Par 2019: Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 11725, Springer, 2019, pp. 17–28, ISBN: 978-3-030-29399-4. DOI: 10.1007/978-3-030-29400-7_2.

[105] M. Norouzi, F. Wolf, and A. Jannesari, “Automatic construct selection and variable classification in OpenMP”, in *Proc. of the Intl. Conference on Supercomputing (ICS), Phoenix, AZ, USA*, ACM, 2019, pp. 330–341, ISBN: 978-1-4503-6079-1. DOI: 10.1145/3330345.3330375.

[106] R. T. Prosser, “Applications of boolean matrices to the analysis of flow diagrams”, in *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, ser. IRE-AIEE-ACM ’59 (Eastern), Boston, Massachusetts: Association for Computing Machinery, 1959, pp. 133–138, ISBN: 9781450378680. DOI: 10.1145/1460299.1460314.

[107] D. Quinlan, “ROSE: Compiler Support for Object-oriented Frameworks”, *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000, ISSN: 1793-642X. DOI: 10.1142/S0129626400000214.

[108] M. Reif, F. Kübler, M. Eichberg, and M. Mezini, “Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java”, in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA’18, Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 107–112, ISBN: 9781450359399. DOI: 10.1145/3236454.3236503.

[109] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, “Detecting code clones in binary executables”, in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA ’09, Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 117–128, ISBN: 9781605583389. DOI: 10.1145/1572272.1572287.

[110] M. Sagebaum, T. Albring, and N. R. Gauger, “High-performance derivative computations using CoDiPack”, *ACM Trans. Math. Softw.*, vol. 45, no. 4, 2019. DOI: 10.1145/3356900.

[111] M. Sagebaum and N. R. Gauger, *MeDiPack – message differentiation package*, 2020. [Online]. Available: www.github.com/scicompkl/medipack.

[112] E. Saillard, P. Carribault, and D. Barthou, “PARCOACH: Combining static and dynamic validation of MPI collective communications”, *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 425–434, 2014.

[113] D. Schmidl, C. Iwainsky, C. Terboven, C. Bischof, and M. S. Müller, “Towards a performance engineering workflow for openmp 4.0”, in *Parallel Computing: Accelerating Computational Science and Engineering (CSE)* (Advances in Parallel Computing), Advances in Parallel Computing. 2014, vol. 25. DOI: 10.3233/978-1-61499-381-0-823.

[114] D. Schmidl, P. Philippen, D. Lorenz, *et al.*, “Performance analysis techniques for task-based OpenMP applications”, in *OpenMP in a Heterogeneous World*, Springer Science + Business Media, 2012, pp. 196–209. DOI: 10.1007/978-3-642-30961-8_15.

[115] P. D. Schubert, B. Hermann, and E. Bodden, “Phasar: An inter-procedural static analysis framework for c/c++”, in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 393–410, ISBN: 978-3-030-17465-1. DOI: 10.1007/978-3-030-17465-1_22.

[116] K. Serebryany and D. Vyukov, *Threadsanitizer, a data race detector for c/c++ and go*.

[117] J. Seward and N. Nethercote, “Using valgrind to detect undefined value errors with bit-precision.”, in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[118] J. Seyster, K. Dixit, X. Huang, *et al.*, “Aspect-oriented instrumentation with gcc”, in *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, *et al.*, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 405–420, ISBN: 978-3-642-16612-9.

[119] J. Shalf, D. Quinlan, and C. Janssen, “Rethinking hardware-software codesign for exascale systems”, *Computer*, vol. 44, no. 11, pp. 22–30, 2011.

[120] S. S. Shende, “The tau parallel performance system”, *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006. DOI: 10.1177/1094342006064482.

[121] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf, “Exascaling your library: Will your implementation meet your expectations?”, in *Proc. of the International Conference on Supercomputing (ICS), Newport Beach, CA, USA*, ACM, Jun. 2015, pp. 165–175.

[122] J. Slabý, J. Strejček, and M. Trtík, "Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution", in *Formal Methods for Industrial Critical Systems*, Springer, 2012, pp. 207–221, ISBN: 978-3-642-32469-7. DOI: `10.1007/978-3-642-32469-7_14`.

[123] J. Slabý, J. Strejček, and M. Trtík, "Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution", in *Formal Methods for Industrial Critical Systems*, M. Stoelinga and R. Pinger, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 207–221, ISBN: 978-3-642-32469-7.

[124] D. Sokolowski, J.-P. Lehr, C. Bischof, and G. Salvaneschi, "Leveraging Hybrid Cloud HPC with Multitier Reactive Programming", in *2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud)*, IEEE, Nov. 2020, pp. 27–32, ISBN: 978-0-7381-1055-4. DOI: `10.1109/SuperCompCloud51944.2020.00010`.

[125] B. Steensgaard, "Points-to analysis in almost linear time", in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96, St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 32–41, ISBN: 0897917693. DOI: `10.1145/237721.237727`.

[126] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles", in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ISSN: 2167-4337, 2010, pp. 1–11. DOI: `10.1109/SC.2010.47`.

[127] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary analysis for measurement and attribution of program performance", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 441–452, ISBN: 978-1-60558-392-1. DOI: `10.1145/1542476.1542526`.

[128] *The gnu compiler collection*. [Online]. Available: `https://gcc.gnu.org`.

[129] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms", *SIGPLAN Not.*, vol. 35, no. 10, pp. 281–293, Oct. 2000, ISSN: 0362-1340. DOI: `10.1145/354222.353190`.

[130] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments", in *39th Intl. Conf on Parallel Processing Workshops.*, 2010, pp. 207–216. DOI: `10.1109/ICPPW.2010.38`.

[131] R. Tschüter, J. Ziegenbalg, B. Wesarg, *et al.*, "An llvm instrumentation plug-in for score-p", in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC'17, Denver, CO, USA: Association for Computing Machinery, 2017, ISBN: 9781450355650. DOI: 10.1145/3148173.3148187.

[132] J. S. Vetter, S. Lee, D. Li, *et al.*, "Quantifying architectural requirements of contemporary extreme-scale scientific applications", in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds., Cham: Springer International Publishing, 2014, pp. 3–24, ISBN: 978-3-319-10214-6. DOI: 10.1007/978-3-319-10214-6_1.

[133] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus", in *Proc. of the Intl. Conference on High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013, pp. 1–12.

[134] J. H. Ward Jr., "Hierarchical Grouping to Optimize an Objective Function", *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236–244, 1963. DOI: 10.1080/01621459.1963.10500845.

[135] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?", in *IEEE Intl. Symp. on Workload Characterization, 2008. IISWC 2008.*, 2008, pp. 141–150. DOI: 10.1109/IISWC.2008.4636099.

[136] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior", in *Computational Science - ICCS 2004*, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 440–447, ISBN: 978-3-540-24688-6.

[137] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with scalaloci", *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276499. [Online]. Available: https://doi.org/10.1145/3276499.

[138] M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984. DOI: 10.1109/TSE.1984.5010248.

[139] J. J. Wilke, J. P. Kenny, S. Knight, and S. Rumley, "Compiler-assisted source-to-source skeletonization of application models for system simulation", in *High Performance Computing*, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, Eds., Cham: Springer International Publishing, 2018, pp. 123–143, ISBN: 978-3-319-92040-5.