Master's Thesis

Configuration-Aware Performance Analysis of Compile-Time Configurable Systems

Approaches and Challenges for the HPC Domain

Lukas Abelt

January 2, 2024

Advisor:
Florian Sattler   Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel   Chair of Software Engineering
Prof. Dr. Sebastian Hack   Compiler Design Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ____________________________

(Datum/Date) (Unterschrift/Signature)

Abstract

Detecting the cause of performance regressions in complex real-world software systems is a non-trivial task. Even more so, if the system at hand is highly configurable, such as the High-Performance Computing (HPC) frameworks Dune or HyTeG. Applications built on top of these frameworks often run on large computing clusters. Hence, detecting performance regressions early is important, as even small performance regressions can produce significant execution overhead and, by that, entail high costs.

Currently, an open issue when it comes to regression detection is tracking down the cause of the performance regression in these compile-time configurable systems. The problem is that, in these systems, performance issues can arise only for certain features or specific combinations of features. While current black-box profiling techniques allow to identify the occurrence of a regression, they often lack the proper capabilities to track down the cause of a performance regression. White-box performance profilers on the other hand can provide much more insights into the system internals, enabling them to detect the cause of regressions or identify performance bottlenecks. While techniques exist to make current state-of-the-art profilers feature-aware, there currently is no clear picture on the exact capabilities and limitations of such approaches.

In this thesis, we investigate the capabilities of various white-box profilers to detect feature-specific performance regressions. We evaluate, how severe a regression needs to be, such that it is detected by a profiler, how capable configuration-aware white-box profilers can attribute regressions to specific features and how accurate different profilers measure regressions. We evaluate these metrics by injecting synthetic regressions into several synthetic and real-world subject systems and perform measurements with three state-of-the-art profilers that we enhance with feature-specific information.

Our results indicate that white-box profilers achieve comparable or better results than black-box profilers in detecting feature-specific regressions and their measuring accuracy. When attributing regressions to specific features, our results show that there is a notable difference between both different profiling approaches as well as between different kinds of subject systems. Overall our results show promising first insights into the capabilities of configuration-aware profilers while also revealing shortcomings for real-world subject systems that highlight the need for additional investigation.

O glücklich, wer noch hoffen kann,
Aus diesem Meer des Irrtums aufzutauchen!
Was man nicht weiß, das eben bräuchte man,
und was man weiß kann man nicht brauchen.

— Faust [20]

Acknowledgments

Writing a thesis is always a challenging journey. A journey riddled by challenges, disappointments, enlightening moments and opportunities. I want to use this opportunity to thank everyone who has enabled me this journey that has not always been easy, but in the end was an enjoyable experience that I am most grateful for.

First and foremost I would like to thank my advisor Florian Sattler who, not only for the past six months, but also long before that, has been a great guidance and inspiration to me. Florian always, willingly or unwillingly, presented me with new challenges requiring me to test my limits, for which I am grateful. His workflow is also a piece of art which I can only hope to come close to replicate at some point in the future. I hope that he will continue to strive in his upcoming new position outside of academia.

Second, I want to thank Prof. Dr. Sven Apel not only for being my first examiner of this thesis, but most importantly for opening my eyes to the realm of configurable software systems. When starting my journey here at Saarland University in late 2020, I always assumed that it will just be a short, temporary, section of my life to simply obtain a Masters degree. I assumed that I will turn back to my life as a software engineer and had no serious interest in pursuing a further academic career. It was the introduction to configurable software systems that kindled the flame in my mind to pursue knowledge. The whole domain did, and still does, fascinate me for many reasons. I often see so many parallels to the challenges in my past jobs, that it made me realize that research and practice do not have to be mutually exclusive. For this realization, that came from multiple discussions with Sven and all the various colleagues at the software engineering chair, I am much grateful. I am even more grateful that I have the opportunity to further dive into this with a PhD position after my masters.

In a more general notion, I want to thank everyone who provided feedback on my thesis and provided me with valuable inputs regarding type-setting, figures and general improvements on my thesis structure. In detail, I want to thank my colleague Sebastian Böhm for providing various helpful LaTeXsnippets and a comprehensive and nice C++ code style for the listings. Furthermore, thanks to my proof readers Nils Alznauer and Dr. Andreas Buchheit who have provided my with feedback and corrections at various stages of the thesis. I also want to thank Prof. Dr. Sebatian Hack for agreeing to be the second examiner for this thesis.

And finally, I want to say a wholehearted *Thank You!* to all my friends and family who have always supported me throughout my life. A special and personal mention goes to my fellow students of the Computer Science Students' Representative Council: Being part of the students council has been a major part of my studies here at Saarland University and has been one of the best decisions I have made. Coming to a new town is difficult enough as it is, even more so in the middle of COVID. The work within the students' council has introduced me to a great community of wonderful people, most of them dedicated to making the best out of the students life at our university. I am grateful. Grateful, for the many events that we organized together. Grateful, for the many hours of discussion we had in our weekly meetings. Grateful, for the numerous complaints by me that none of you seem to know how to operate a dishwasher. Grateful, for all the friends I have made along the way. Organizing events for and with the students' council has become a great passion of mine and an integral part of my identity. I sincerely hope that I can stay part of this great community, even though my time might be much more limited in the future.

Contents

List of Figures

List of Tables

Listings

Acronyms

PIM	Performance Influence Model
SUT	System-under-test
HPC	High-Performance Computing
CRTP	Curiously Recurring Template Pattern
API	Application Programming Interface
TEF	Trace Event Format

XML	Extended Markup Language
IR	Intermediate Representation
SDT	Statically Defined Tracing

Introduction

Most modern software systems offer the user a wide range of configuration options to make them configurable. This gives a developers and users alike the flexibility to only enable the functionality that is relevant for their specific use-case. This reduces the risk of bugs being introduced [26] to an installation and also decreases the overall size of the installed system [2]. One prominent example of such a configurable software system is the Linux Kernel, which provides over 10 000 independently selectable features, also referred to as *configuration options* [62]. A *variant* of a software is referred to as the specific instance of a software with a specific configuration. In this context, the *configuration* refers to the subset of features that is enabled.

However, the flexibility of configurable software systems also introduces its own unique challenges. Given that the amount of possible software configurations grows exponentially with the number of configuration options, it becomes non-trivial to analyse such software systems with respect to their qualitative and quantitative properties, e. g., performance or correctness. The problem is that certain bugs or performance regressions may only occur in specific configurations and interactions of features, meaning, that these effects only occur when a certain subset of features is enabled [26]. Due to this high complexity of configurable software systems, identifying and fixing performance regressions is a non-trivial task. Due to the exponential growth of possible configurations, testing all configurations is infeasible practically. Therefore, configuration-aware performance analysis approaches for such systems are imperative.

A prominent use-case for configurable software systems are application in the HPC domain. In this domain configurability is used to tailor frameworks to a multitude of problems and optimize performance. Examples for that are the HPC frameworks Dune and HyTeG. For such HPC applications, performance analysis and regression detection becomes especially relevant as even small performance regression can aggregate quickly when run on large parallel compute clusters, leading to significant overhead in required resources and costs. As the individual parts of these HPC applications also tend to be highly specialised and domain specific, a regression detection that can directly identify the affected feature is a helpful guideline for debugging and mitigating the cause of a regression.

In current research two strategies have emerged for analysing the performance of configurable software systems: black-box techniques and white-box techniques. In black-box analysis, different variants of the software are created and their performance is measured independently from each other by a black-box profiler. From the data measurements, combined with information about the selected features, black-box performance analysis builds performance influence models [48], through which a black-box approach can approximate the performance influence of individual features and even predict the performance of new configurations without explicitly measuring them. However, since not only single features but also the interaction of multiple features can have an influence on performance, building an accurate model requires measuring a large set of configurations. The reason for this is that the potential number of feature interactions grows super-exponentially with the number of individual features. In practice different sampling techniques arise that aim to select a suitable small set of configurations following specific heuristics, such as code coverage, or coverage of specific features or interactions [24, 30, 51]. Ultimately, a black-box performance model only has access to performance data of a subset of all configurations. For features and interactions that did not arose in the initial sample, a model can only approximate and interpolate the performance, thus the resulting prediction may be inaccurate.

The second strategy to performance analysis are white-box approaches, which is a more informed approach to configuration-aware analyses. Using white-box approaches it is possible to relate performance data to specific parts of the inner workings of a configurable software system. In a most naïve approach one could manually investigate the source code of a system to identify potentially performance relevant code sections. However, this approach is tedious, error prone, and also not suitable for real-world software systems that consist of hundreds of thousands of lines. Therefore, different profiling approaches have emerged that can be used to extract such information in a more automated manner. Special tooling, such as compiler extensions, use different approaches to collect data specific to the inner workings of a software system. This additional information enables an informed and detailed analysis of the system at hand. Furthermore, they enable to enrich the program with additional feature-specific data that can later be used in composition with other tooling such as profilers, to enable a configuration-aware performance analysis. However at the moment there is no clear picture about the exact capabilities and limitations of configuration-aware white-box performance analysis.

Another challenge that arises when analysing configurable software systems is the variety of implementation techniques that realise the configurability in code. Different patterns, idioms, and technical frameworks are used to implement the selection and deselection of features at run-time, load-time, or compile time. Depending on the technique a profound understanding of the underlying programming language is required to fully understand the implementation. One example for this are *templates* in the programming language C++. By embedding configurability information into the type system of C++ different functionality can be selected at compile time. By embedding the configuration into the type system, this allows to already select and tailor functionality at compile-time which can lead to improvements at runtime as e. g., virtual calls can be eliminated which, in a high-performance environment, can lead to performance gains. While templates are a powerful mechanism to implement variability, its complexity highlights the need for automatic tooling to relate performance-data with configurability information even more. With this work we aim to address this by analysing several synthetic and real-world compile time configurable systems and evaluate the capabilities of configuration-aware white-box profilers.

1.1 Goal of the Thesis

The main goal of this thesis is to explore the capabilities and limitations of configuration-aware white-box profilers with regards to detecting performance regressions. We investigate and compare the differences both between black-box and white-box profilers and between different white-box profilers. We focus on a specific family of compile-time-configurable software systems that use C++ templates as a mechanism to implement variability. One of the reasons is that C++ is often used for high-performance computing applications. Examples for software that use templates are the numerical simulation frameworks Dune or HyTeG. Since these systems usually run on large computing clusters, detecting and investigating small performance regressions is relevant as these can quickly lead to an substantial overhead in CPU time spent and, as an effect, increased cost and energy consumption. Both Dune and HyTeG employ C++ templates to select specific implementations at compile-time. Since templates are complex to develop and understand [34, 52], this work aims to improve the understanding of performance regressions identification to benefit both users and developers of such frameworks.

1.2 Contributions

With this thesis we aim to make provide a better understanding of the capabilities and limitations of configuration-aware white-box profilers for compile-time configurable systems. For this we provide a unified experiment pipeline that is applicable to detect performance regressions with different black-box and white-box profiling approaches. Our work shows that configuration-aware white-box profilers detect feature-specific regressions with at least the same sensitivity and accuracy as a black-box compiler for simple subject systems. However, our results also show that complex real-world systems exhibit challenges when it comes to feature detection and hence, also detecting feature-specific regressions. Furthermore, our results reveal that the analyses capabilities of different white-box profilers can differ due to different technical limitations of the profilers or due to the implementation techniques used in the evaluated software systems. Lastly, we report on a shortcoming in our experiment pipeline which serves as a basis for an improved experiment setup for future work. To run our experiments we provide a selection of synthetic subject systems that implement simple compile-time configurability systems using different implementation techniques. Lastly, we present a vast set of synthetic regressions for all of our subject systems and in addition also two real-world systems Dune and HyTeG.

All our implementations and code we use for the evaluation are publicly available on GitHub as part of the VaRA-Tool-Suite project[1]. This allows others to reproduce and verify our results. Our additional digital appendix includes the raw results of our experiments and instructions how to reproduce most of the tables and plots that are in this thesis.

1 https://github.com/se-sic/VaRA-Tool-Suite/tree/MA-Abelt

1.3 Thesis Structure

This thesis is structured as follows. Chapter 2 gives an overview of the core concepts that are required to follow this thesis including configurable systems, implementation patterns for them and the basis of performance profiling and analyses, followed by a detailed description of our methodology in Chapter 3. The methodology chapter provides details about our exact research questions, as well as our subject systems, the profilers we evaluate and the regressions we use for our evaluation. Chapter 4 describes how we structure our experiments to evaluate these research questions. In addition we define our operationalization and expectations for each research question. In Chapter 5 we report our experiments results. We also perform a discussion of our findings with regard to our expectations and provide an overview of the potential threats to validity. Chapter 6 gives an overview of other works that discuss similar questions as this thesis and provides an insight into the state of the art. Finally, Chapter 7 provides a conclusion to our work and an outlook on future work and extensions to the experiments and evaluations of this thesis.

2 Background

In this chapter, we give an overview of the core concepts required to follow this thesis.

2.1 Configurable Software Systems

This chapter gives an overview of Configurable Software Systems. This includes an overview of the general concept, the core terminologies that we also use throughout this thesis and a short summary of implementation techniques that realize configurability.

2.1.1 Terminology

As a configurable software system we understand a system in which users can decide to include or exclude certain functionalities. The selection of these functionalities results in an end-product which offers a specific subset of functionality to the end user or simply different behavioral characteristics such as performance. We refer to single functionalities that can be (de-)selected as *features* or *configuration options* of the configurable software system.

A configurable software system may consist of many different features. We refer to the set of selected and deselected features as the *configuration*. A *variant* is a configurable software system that uses a specific configuration. Usually, there are restrictions and dependencies between the individual features that restrict the amount of valid configurations of a configurable software system. Take for example a software system which offers different encryption methods. While different encryption methods may be supported, it is not possible to use multiple encryptions at the same time. Some encryption methods may only be usable in conjunction with other features, such as specific signing algorithms. In general, to prevent misconfiguration of the system, we need to encode such constraints and check the validity of a configuration.

In practice, *feature models* encode the restrictions between the individual features of a configurable systems. A feature model is a boolean formula which describes the valid configurations of a configurable software system. These models encompass the relation and restriction between the individual features and therefore describe which configurations of a software system are valid. Feature models can also be visualized as trees in *feature diagrams*. This visual representation has the benefit that they are also interpretable by humans. In its boolean representation, feature models are used to check whether a given configuration is valid for the software system [6].

Figure 2.1 shows an example for a simple feature diagram of a database application. The feature model allows to represent different dependencies between individual features. For

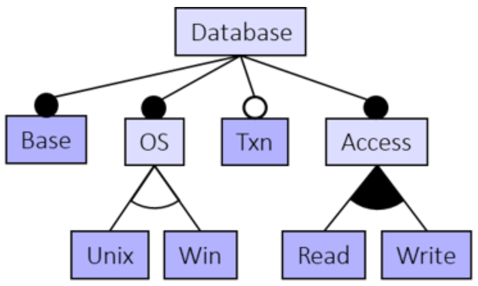


Figure 2.1: Example for a feature diagram of a simple database application

example features can be mandatory or optional child features. In addition one can use the different group types such as alternative groups to e. g., model mutual exclusion between features. However in real world software the dependencies of features may be more complex than to adhere to the strict hierarchical structure that can be represented by a tree. Therefore additional constraints may be added as boolean formulas below the feature diagram. We refer to those as *cross-tree constraints*.

2.1.2 Binding Times

In the context of configurable software systems the term *binding time* refers to the time at which a program decides to include or exclude a feature. Depending on the binding time different implementation techniques are appropriate to implement this kind of binding. Generally, we differ between three binding times: run-time, load-time and compile time [6].

For run-time binding, all feature code is always shipped with the program. However, due to different control paths or decision made during run-time the program runs only the feature specific code that is currently active. Therefore it is also possible to reconfigure a configurable software system while it is running. One example of a run-time configuration option is the selection of a theme while using a web-browser or the installation of certain extensions.

With load-time binding the feature selection is made during the startup of a program and then do not change throughout the run of a program. Examples for load-time binding are command-line options or configuration files that are read during startup. Other examples include certain software architectures where e. g., a certain set of plugins to be used for a program can only be altered during startup of the program [6, 45].

For compile-time bound features, the decision which features are included in the final software is already made during the compilation of a program variant. That means, as opposed to run- and load-time configurability, the compiled binary only contains the functionality that has been selected. Deselected features are not accessible during execution. One benefit of this approach is that undesired functionality is not delivered, which can lead to smaller binary sizes, memory footprint and increased security[11, 60, 63].

2.1.3 Implementation Techniques

Variability can be implemented with a variety of different implementation techniques. Choosing the appropriate technique depends on the exact requirements and use-case of a scenario. For this thesis however, we focus software systems from the HPC domain. As these software systems are tuned to maximizing performance, run- or load-time variability is not always an optimal choice as these can introduce small overheads at run-time. For this reason, such systems often use compile-time techniques to realize variability.

Due to our focus on HPC systems, we limit this section to an overview of implementation techniques for compile-time variability. For an extensive overview of implementation techniques for run- or load-time variability, we refer to the corresponding literature[6].

One of the most prominent implementation techniques for compile-time variability is the use of the C preprocessor in combination with **#ifdef** directives. With this approach, certain code blocks are completely removed from the source code before the actual compilation process starts [43]. Because this approach can also just remove single lines of code from the source, it offers a very fine-granularity of variability. On the one hand, whole functions or parts thereof could be added or removed based on the configuration but at the same time, the C preprocessor can also be used to add arguments to a function signature for specific configurations. Listing 2.1 shows a small code snippet in which the C preprocessor alters a functions signature and behavior based on the specific configuration.

Listing 2.1: Example how the C preprocessor can en- and disable functionality based on configuration. Here, the USE_COMPRESSION macro to alter the function signature of foo and alter its' behavior

```
// Un-/Comment following line to en-/disable compression
#define USE_COMPRESSION

void sendData(MessageData Message
#ifdef USE_COMPRESSION
              ,int compressionLevel
#endif
             )
{
#if USE_COMPRESSION
  Message = compress(Message, compressionLevel);
#endif
  send(Message);
}
```

The C preprocessor is widely used in practice and various research has been published around the analysis and specific challenges of this implementation technique[17, 43, 46, 61]. Due to the challenges of C-Prepocessors, there is also different work into building dedicated tooling support[18, 32, 40] and how to lift pre-processor variability to other forms of variability[28, 55, 58].

For the C++ programming language there exists another prominent approach that can be used to selectively compile functionality into the final program. *Templates* allow a wide range of implementation options to provide configurable software. The simplest usages of templates are generic data containers and algorithms as the *Standard Template Library* of the

C++ language provides them. During the compilation process the compiler will generate a separate version of a templated class or function for each different type that it is required for. Popular examples for templates are types like `std::vector` and `std::map`. Listing 2.2 displays some simple examples of implementations for generic data structures and algorithms.

Listing 2.2: Example of a generic data pair class and a generic print function

```cpp
template <typename StoredType>
struct ValuePair {
  StoredType First;
  StoredType Second;
}

template<typename ValueType>
void print(ValueType Value) {
  std::cout << Value << std::endl;
}

// Later use in code:
ValuePair<int> IPair{13, 37};
ValuePair<string> SPair{"Hello", "World"};

print(SPair.First);
print(SPair.Second);
print(IPair.First);
print(IPair.Second);
```

However, by using more advanced implementation techniques, one can also encode the variability into the type system of a C++ program. We will present a selection of a few commonly used implementation techniques that employ templates to realize variability. We base our overview of the implementation techniques on the in-depth descriptions of Vandevoorde[64], Alexandrescu[4] and Czarnecki[15].

Template Specialization With this technique it is possible to provide a specialized implementation for a templated class for function when used with specific template arguments. A template specialization can be useful for multiple scenarios: Some classes may not offer the interface that a templated function uses. When this only occurs for a few selected instances, providing a specialized version might be easier to implement than using other techniques to provide a unified interface. Consider for example Listing 2.3 in which we provide a specialized variant of the `print` function from Listing 2.2, to print `bool` types in a more readable manner.

Listing 2.3: Example for a template specialization for the `print` function

```cpp
template<>
void print<bool>(bool Value) {
  if (Value) {
    std::cout << "true" << std::endl;
  } else {
    std::cout << "false" << std::endl;
  }
}
```

Another scenario where a specialized version of a template function might be preferred even if it offers a common interface are instances where classes may provide a more specific interface that is more efficient than the general interface. Take, for example, a generic library solving differential equations such as Dune or HyTeG: In this, some of the underlying grid types might provide more efficient access methods that should be used instead of a more generic element-wise access. In this case, template specialization provides a way to use the grid-specific interface on these occurrences.

For template classes the specialization can occur for either the whole class, or even only for parts of a class. Depending on the use-case this allows to either only modify the behavior of specific member functions of a class or adapt the interface of a class altogether. A common example for a specialized class in C++ is the `std::vector<bool>` class, which provides a more space efficient implementation for the `std::vector` container type, when storing boolean values in it. In this specialization the boolean elements of a vector may be stored in a dynamic bitset such as that each element only occupies one bit in memory[1].

Traits Template parameters allow for flexible design of classes and functions. However it may not be desirable to introduce separate template parameters for all possible customization points. While this allows to tailor the functionality to every possible use-case scenario this becomes overly tedious to use in client code as one has to specify all template parameters. For a lot of cases however, most of these possible additional template parameters may have reasonable default values or can be specified dependent on some "main" parameters. *Traits* are a technique to express this kind of flexibility using templates in C++.

Facilitating traits is possible in a multitude of ways, e. g., by representing default values or return types or simply providing a traits class to represent the capabilities of a specific class. As a motivating example, consider the template function `accumulate` as depicted in Listing 2.4. While for most cases it seems reasonable that the return type is the same as the input type, there are scenarios in which that assumption leads to errors. A straightforward case for this is when one accumulates values of small integer types, such as `char`. Here using the small type as a result type can quickly lead to overflows.

Listing 2.4: A simple template function that accumulates values in a given range

```cpp
template<typename T>
T accumulate (T const* beg, T const* end)
{
  T total{};
  while (beg != end) {
    total += *beg;
    ++beg;
  }
  return total;
}
```

To avoid specifying the result type as an additional template type, we can define a simple type trait that encapsulates the result type. From an implementation perspective there are different ways to implement traits. One of them is to use template specialization. That is, a trait class is a template class that takes as a template argument the type for which it contains

1 If and how this specialization is provided may differ between different compilers and is implementation-defined

the specific traits. The general template class may provide reasonable default values or no implementation at all. While the former makes it easier to use traits with new classes the latter enforces the user to provide a reasonable specialization themselves for their custom types. Listing 2.5 shows an example of a traits template class that encapsulates the result type for our `accumulate` function. Listing 2.6 now shows our new version of `accumulate` that makes use of this trait template.

Listing 2.5: Trait templates for our accumulate function

```
template<typename T>
struct AccTraits;

template<>
struct AccTraits<char> {
  using ResT = int;
};

template<>
struct AccTraits<short> {
  using ResT = int;
};

template<>
struct AccTraits<int> {
  using ResT = long;
};
```

Listing 2.6: Example of our `accumulate` function using traits

```
template<typename T>
auto accumulate (T const* beg, T const* end)
{
  using ResT = typename AccTraits<T>::ResT;

  ResT total{};
  while (beg != end) {
    total += *beg;
    ++beg;
  }
  return total;
}
```

The above example is just one example usage of traits. There are a variety of other trait techniques for example to transform, compare or classify types. For a more detailed description of trait types and implementation techniques we refer the reader to additional literature [64].

Since the C++11 standard, traits are already incorporated into the language standard. The most prominent examples that C++ developers will get in touch with are the `<type_traits>` and `<iterator_traits>` headers. While the former provides various functionality to convert, transform and compare types, the latter provides common traits that are useful for implementing generic algorithms that make use of iterators. In modern language standards traits, in composition with the `constexpr` keyword can even be used for conditional compilation.

Policies Another example to modify the behavior of parametrized classes are *Policies*, which share some commonalities with traits and can also benefit from one another. However while the focus of traits is more on types, policies focus on making behavior configurable. Take, for example, again the *accumulate* function in Listing 2.4. Here another application scenario might be that one wants to find the minimum or maximum of a sequence. In the given implementation we can achieve this by simply changing line 5 to the corresponding `min(...)` or `max(...)` operation.

Listing 2.7: Simple examples for different accumulation policies.

```cpp
struct SumPolicy {
  template<typename T>
  static void accumulate (T& curRes, T const& nValue) {
    curRes += nValue;
  }

  template<typename T>
  static T initial() {
    return T{};
  }
};


struct MaxPolicy {
  template<typename T>
  static void accumulate (T& curRes, T const& nValue) {
    curRes = max(curRes, nValue);
  }

  template<typename T>
  static T initial() {
    return std::numeric_limits<T>::lowest();
  }
};

struct MinPolicy {
  template<typename T>
  static void accumulate (T& curRes, T const& nValue) {
    min(curRes, nValue);
  }

  template<typename T>
  static T initial() {
    return std::numeric_limits<T>::max();
  }
};
```

Such scenarios are examples where a policy class comes to play. A specific policy now defines a specific class (template) interface. This interface can have all the same properties as regular classes, such as member variables and functions or inner type definition. For our accumulate example, the policy would be a simple class with a static `accumulate` function that takes the current result and the next value as its' input and returns the result of the appropriate operation. In addition our policy class defines an additional `initial` function, that provides a reasonable initial value for the policy. Listing 2.7 shows an example for three different policies that calculate the sum, minimum and maximum respectively.

We can now use these policy classes to modify the behavior of our `accumulate` function. Listing 2.7 shows a version of the function where the user can specify the desired policy via a template parameter. One important note is that our implementation still misses some implementation details, e. g., we can easily run into the same issue as described in the *Traits* section where the accumulate function uses an inappropriate return type, which could be avoided by combining traits and policies.

Listing 2.8: Example of a generic `accumulate` function using policy classes. The user can modify the behavior of accumulate by providing an appropriate policy class as a template parameter.

```cpp
template<typename T,
       typename Policy = SumPolicy,
T accumulate (T const* beg, T const* end)
{
  T res = Policy::initial();
  while (beg != end) {
    Policy::accumulate(res, *beg);
    ++beg;
  }
  return res;
}
```

***Curiously Recurring Template Pattern* (*CRTP*)** This implementation technique describes a scenario for which a class is derived from a template class. However, a class specifically passes itself as a template parameter to the superclass. Listing 2.9 shows a simple usage of this pattern. While this pattern has various usages, for our scope we want to mainly focus on *static polymorphism* applications.

Listing 2.9: A general example of an implementation of the CRTP

```cpp
template<typename Derived>
class CRTP {
  ...
}

class Concrete : public CRTP<Concrete> {
  ...
}
```

The use of CRTP allows the derived classes to implement their own behavior while still providing a common interface through a common base class. However, as opposed to ”standard” inheritance that achieves this through dynamic polymorphism and function overriding, CRTP can enable the same functionality while avoiding virtual function calls. For most common day software the overhead of virtual function resolution will most likely go unnoticed by the user, which is not always the case in the HPC environment. For HPC applications such as Dune or HyTeG even such a slight performance improvement can lead to reduction in the necessary computing resources or execution time when run on massive compute clusters, which is one of the main reasons we consider this implementation pattern in our evaluation.

Listing 2.10 shows how CRTP achieves this in practice: As in CRTP the base class will always be instantiated with its respective derived class as a template parameter this can eliminate an indirection that is usually solved with a virtual function call. During compilation, the instantiated template of our CRTP-base class already knows the specific subclass it is instantiated for. Therefore, one can use a `static_cast` to the derived class, which allows the compiler to already identify the exact function to be called during compile-time, eliminating the need for any virtual calls during execution.

Listing 2.10: An example how CRTP uses `static_cast` to eliminate be a virtual function call

```
template<typename Derived>
struct CRTPBase {
  void foo(){
    Derived &Underlying = static_cast<Derived &>(*this);
    Underlying.foo();
  }
}

struct Concrete : public CRTPBase<Concrete> {
  void foo(){
    std::cout << "Called from Concrete"<< std::endl;
  }
}
```

2.2 Performance Analysis

Complex and compute intensive operations, such as one finds them in HPC applications, often run on large cluster systems to get an output in a reasonable time frame. However it is not always possible or feasible to just speed up a computation by providing more hardware to solve the problem. The issues with this naïve idea are multifold: For once in the real-world one always has to deal with limited budget and, consequently, resources. Thus just ”adding more resources” is usually not a satisfying solution. Second, one cannot rely on ”automatic” hardware improvements anymore as current hardware processor and transistor designs are reaching the limits of what is physically possible[16].

For this reason one must choose another angle to approach the challenge at hand. If hardware improvements are getting harder and harder, the other driver for performance gains are software improvements[41]. However to improve software one first needs to understand the current state of a software and what influences its performance. This is where performance analysis comes to play. In short, one goal of performance analysis is to figure out the performance critical sections of code, usually ones were a programs spends much time, to further analyse and improve the code. In the following we will provide a brief overview of the general characteristics of performance analysis and how to conduct it.

Performance analysis refers to a broad term with various interpretations. Throughout this thesis we will follow a loose definition were we refer to performance analysis as the process of identifying the performance characteristics of a software system. As software systems evolves and changes over time [13, 49, 53], it becomes obvious that performance analysis is not a one-time effort. Otherwise a change in the software might degrade and hence cause a *performance regression*. To avoid such regressions, performance analysis is a continuous task that monitors the performance of a software system throughout its' evolution.

To conduct a reasonable performance analysis, one first needs to identify and further quantify the properties of interest in a subject system. For software systems, properties of interest might be cache misses, memory usage, I/O operations or time measurements. Through correct interpretation of such measurements, one may reason about potential memory bottlenecks or identify where time is spent during program execution. For some of these measurements, operating systems provide in-built counters and appropriate Application Programming Interfaces (APIs) to query those. External tools and libraries such as XRay, eBPF or Likwid can provide access to additional information. While performance regressions can be related to any of these performance characteristics, for the scope of this thesis we only consider performance measurements that related to the execution time of a program and time spent in specific parts of a program.

To specify how to collect and interpret performance characteristics, Kounev et al. [39] define the terms *measurement* and *metrics*. On a conceptual level, a measurement describes the raw value of a property of interest, while a metric is more concerned with a high-level interpretation or aggregation of multiple measurements. As a simple example, imagine one wants to analyse the performance of a compression tool. To evaluate this one could use the tool to compress a folder containing multiple files. As a *measurement*, one could now consider the total time required to compress the file or the memory usage during compression. While these measurements themselves can already serve as metrics, they also allow further interpretation. For example, one could build a relation between the number of files compressed and the total execution time to build a *metric* how long the compression tool took on average to compress a single file in the folder. There is of course some speculation if such a metric would be useful or even accurate as a lot of other factors beside the number of files (such as files sizes or file types) influence the compression speed.

Different measurement techniques are available when collecting performance measurements. The differences between these techniques lay in the amount of information about the System-under-test (SUT) that is available to the profiler. One the one hand, a *black-box* approach, considers a SUT solely from an outside perspective treating it as a "black-box", hence the name. While black-box profilers do not cannot use any system-specific information for their analyses, they exhibit several advantages. As they do not require any modification of the SUT, they are easy to set-up and usually do not require a large overhead [10, 22]. A fundamentally different approach is the use of *white-box* profilers. In this approach, the profilers include information about the SUT, such as the source code, into their analyses. Furthermore, these profilers may modify the system to enhance their analyses. White-box performance profilers, for example, might inject additional measurement code into a system during compilation. This measurement code is then executed during evaluation of the SUT and allows a profiler to collect additional information. Naturally, there is no strict division between white-box and black-box profilers. Some techniques may position themselves in the middle-ground between the two approaches, e. g., by only making subtle changes to the SUT. These systems are usually referred to as *gray-box* profilers.

When collecting performance measurements, profilers can use different techniques to decide *how* and *when* this happens. Kounev et al. divide these into three fundamental strategies: *Sampling*, *Event-Driven* and *Tracing* strategies. While the strategies collect the measurements differently, they all revolve around the common mechanism of reacting to specific triggers, also referred to as *events*. Sources for these triggers can be a variety of occurrences, such as the operating system exposing access to specific hard- and software events such as cache-misses or I/O operations. Additionally there are tools and libraries available to create custom tracepoints in a program that, when reached during execution, trigger an event. In the following paragraphs we will give a brief summary of the three profiling strategies.

With the *sampling* strategy, a profiler periodically records a set of measurements in a pre-defined interval. One example for such a measurement could be the programs stack trace. The trigger in this case is an internal timer that causes the profiler to record the measurements with a specific sampling frequency, usually specified in measurements per second (Hz). Sampling has the benefit that the overhead is independent of the number of events, and only driven by the chosen sampling frequency. However at the same time this means that sampling may miss some events if they occur between two measurements. Therefore, using sampling as a profiling strategy incurs a trade-off between overhead and precision.

As the name suggests, *event-driven* profilers built up on the occurrence of events. That is, they record measurements on the occurrence of specified events, e. g., to count the number of branch-misses. For general events that the operating system exposes, this approach usually does not require modification of the SUT. Thus, a black-box profiler could count the number of branch misses without any internal knowledge of the program under test. However, if one wants to record specific information about the SUT, such as the number of call to a specific API, it might be necessary to instrument the program with a white-box profiler.

Tracing strategies can, in a sense, be seen as an extension to event-driven profiling strategies. Similarly, they also collect their measurements after the occurrence of events. However, tracing strategies enable a profiler to collect additional data about the events that occur. For example when measuring the calls to a specific API, tracing allows to e. g., also collect the arguments passed to the API whereas an event-driven strategy only counts the number of calls. This allows to create more in-depth analyses about the SUT.[39]

In the end, choosing an appropriate profiling technique is always a trade-off between precision and necessary overhead. Here overhead refers to both the required overhead to integrate a specific profiling technique into a SUT and the overhead that the technique itself may induce to the SUT. As a profiler may inject specific instrumentation code into the SUT, this can always have unexpected impacts on the code execution due to compiler optimizations or e. g., branch predictions. On the other hand, sampling strategies may have a low-overhead and little influence on program execution, but might yield less precise measurements. In the end, choosing the appropriate strategy and estimating their influences to a SUT is very specific to the application scenario and requires individual consideration.

2.3 Profiling Frameworks

In this section we present the fundamentals of LLVM XRay and the eBPF framework which we use in our evaluation for this thesis.

2.3.1 LLVM XRay

One profiler we consider is the XRay function call tracing system. It allows for low-overhead performance measurements that can be dynamically enabled and disabled. It is part of the LLVM framework which allows for flexible extensions to its functionality. We give a brief overview of the general approach of XRay. Later in Section 3.3, we describe how we utilize the XRay framework to enable configuration-aware profiling.

The XRay framework allows to dynamically trace the function calls during a program execution. It achieves this by embedding so called "no-op sleds" into a binary at compilation time. A "no-op sled" is essentially a specific code sequence that don't do anything. When tracing is disabled these operations do not perform anything and thus only incur minimal runtime overhead. However, once tracing is enabled the XRay runtime dynamically replaces these operation sleds with calls to specific instrumentation code. This instrumentation code then logs information about the specific instrumentation point into trace files. Trace files produced by XRay are stored as Trace Event Format (TEF) files, which is a specific JSON file format storing a list of events[2]. Each event contains information about the event name, its type (e. g., begin or end of a region), a time stamp along with some additional metadata. A TEF file contains sufficient information to reconstruct a program run with specific analyses tools. Tools like Perfetto[3] also visualize traces stored in TEF files which can be useful for manual inspection. Figure 2.2 shows a conceptual example of such a visualized trace.

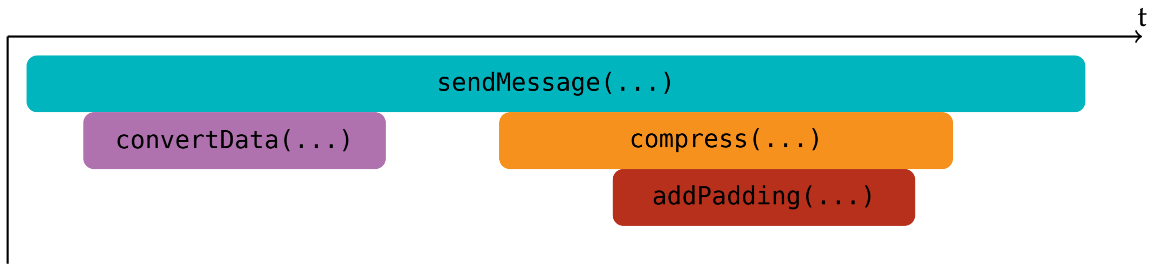


Figure 2.2: Conceptual visualization of traces that XRay generates. The trace visualizes the function stack at any point in time and is therefore useful to see how functions nest.

To reduce and fine-tune the overhead that XRay induces, it employs two heuristics to decide whether it instruments a function. The first heuristic is based on the number of instructions in a function. The user can specify a threshold of X instructions that a function at least needs to contain in order to be instrumented. If the user does not specify a threshold, the default minimum of instructions for XRay instrumentation is 200. The second heuristic selects functions that contain non-trivial loops. Functions with non-trivial loops may not reach the defined instruction threshold but still introduce a non-trivial amount of runtime, that is of interest for analysis. Therefore XRay also instruments all functions containing these kinds of loops[8].

2 https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/edit?usp=sharing
3 https://ui.perfetto.dev/

2.3.2 eBPF

Another profiling framework we consider in our thesis is the eBPF infrastructure[4]. eBPF is a part of the linux kernel that allows to run sandboxed programs in privileged contexts. One example for such a privileged context is the operating system kernel. This capability makes eBPF a perfect fit for performance profiling, as on a kernel level one can keep the overhead lower and also have access to more system insights. At the same time, running code in a privileged environment exposes many risks towards a systems stability and security.

eBPF by itself follows an event-driven strategy to enable profiling. eBPF attaches itself to certain hooks that the kernel or an application passes during execution. These hook points can either be from a set of pre-defined hooks or can be custom hooks that users create for their specific needs. Some examples for pre-defined hooks are function entries or exits, I/O operations or system calls. When a user requires a more specialized hook than is available with the pre-defined hooks, eBPF can also attach to custom kernel probes[5] or user probes[6]. In Section 3.3 we give an overview how we use these capabilities to create a feature-aware profiler using the eBPF ecosystem.

To alleviate the security and stability concerns of running code in a privileged context, eBPF comes with their own ecosystem. This ecosystem ensure that eBPF programs that run in a privileged context adhere to certain guarantees, such as the program does not crash and always terminates. This is realized by eBPFs own verifier that runs on any eBPF program before it runs. eBPF programs themselves are provided in bytecode and only support a specific instruction set. To make eBPF programs more approachable by programmers, projects like Cilium[7], bcc[8] or bpftrace[9] provide a higher level abstraction to develop safe eBPF programs[56].

2.4 Configuration-Aware Performance Analysis

When considering configurable software systems for performance evaluation, some unique challenges arise. Current research reveals that most of the performance issues in configurable software systems arise from specific feature combinations or misconfigurations[1, 26, 62]. Therefore there is a specific need to understand the performance of software systems on a configuration-, or even, feature-level.

To enable a configuration-aware modeling of the performance of a configurable software systems, Siegmund et al. introduce Performance Influence Models (PIMs)[59]. In essence, a PIM is a combination of linear term that assign a coefficient to each feature and feature interactions. This model then allows to predict the performance for any configuration of a subject system. As an example, consider a configurable system that consists only of the two configuration options *Encryption*(🔐) and *Compression*(📦). By measuring all configurations, we can create a performance influence model (Π):

4 https://ebpf.io/what-is-ebpf/
5 https://docs.kernel.org/trace/kprobes.html
6 https://docs.kernel.org/trace/uprobetracer.html
7 https://cilium.io/
8 https://github.com/iovisor/bcc
9 https://bpftrace.org/

$$\Pi(\text{🔐}, \text{📦}) = 5\,\mathrm{s} + 4\,\mathrm{s} \cdot \text{🔐} + 7\,\mathrm{s} \cdot \text{📦} - 2.5\,\mathrm{s} \cdot \text{🔐} \cdot \text{📦}$$

For this notation, 🔐 and 📦 are assigned with either 0 or 1, depending on whether the respective configuration option is selected or not. So, with the PIM given above, a variant with both encryption and compression enabled, will result in a performance of 13.5 s, whereas one that only selects encryption, only 9 s.

The main challenge now lies in building accurate PIMs. As the number of configurations grows exponentially with the number of configuration options, measuring all of them is infeasible in practice. Therefore, one can use different black- and white-box profiling approaches to build such models without measuring all configurations. With a black-box approach the main idea is to build a performance influence model by only measuring a small subset of configurations. The measurements of these configurations, along with the configurability information, then serve as a basis to build a linear model using simple linear regression or more advanced machine learning approaches[24, 25, 49, 51].

The limitation of the black-box approach however, lies in the selection of configurations. As some performance characteristics may only occur in certain configurations, when one does not select these configurations for evaluation, the model cannot represent those. To approach this issue, current research provides different sampling techniques that aim to represent a specific subset of the configuration space[24, 30, 31, 48]. However, in the end a black-box model is only a statistical model that cannot perfectly represent all aspects of the performance of a software system. Specifically configurations that were not part of the initial sample may have different performance characteristics than the model predicts due to feature interactions that did not arise in the original sample.

White-box techniques, on the other hand try to use information about the inner workings, such as the source code, to analyse the behavior of a configurable software system. By having access to, e. g., the source code of a system more detailed data- and control-flow analyses can be performed for various purposes. For example checking for type errors in configurable software systems, finding configurations with higher performance or building more accurate performance models [3, 35, 65, 66].

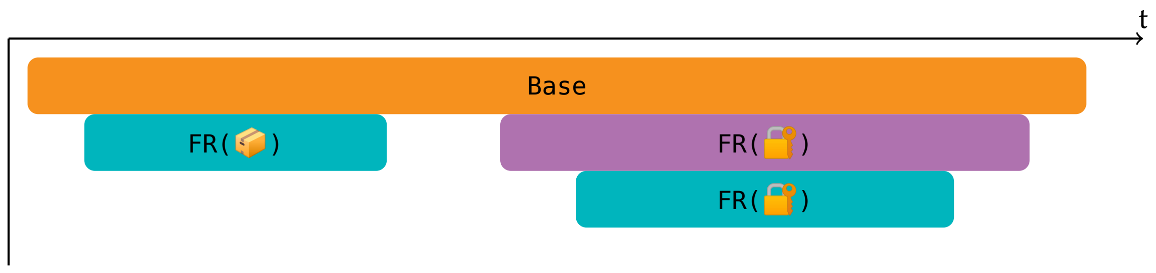


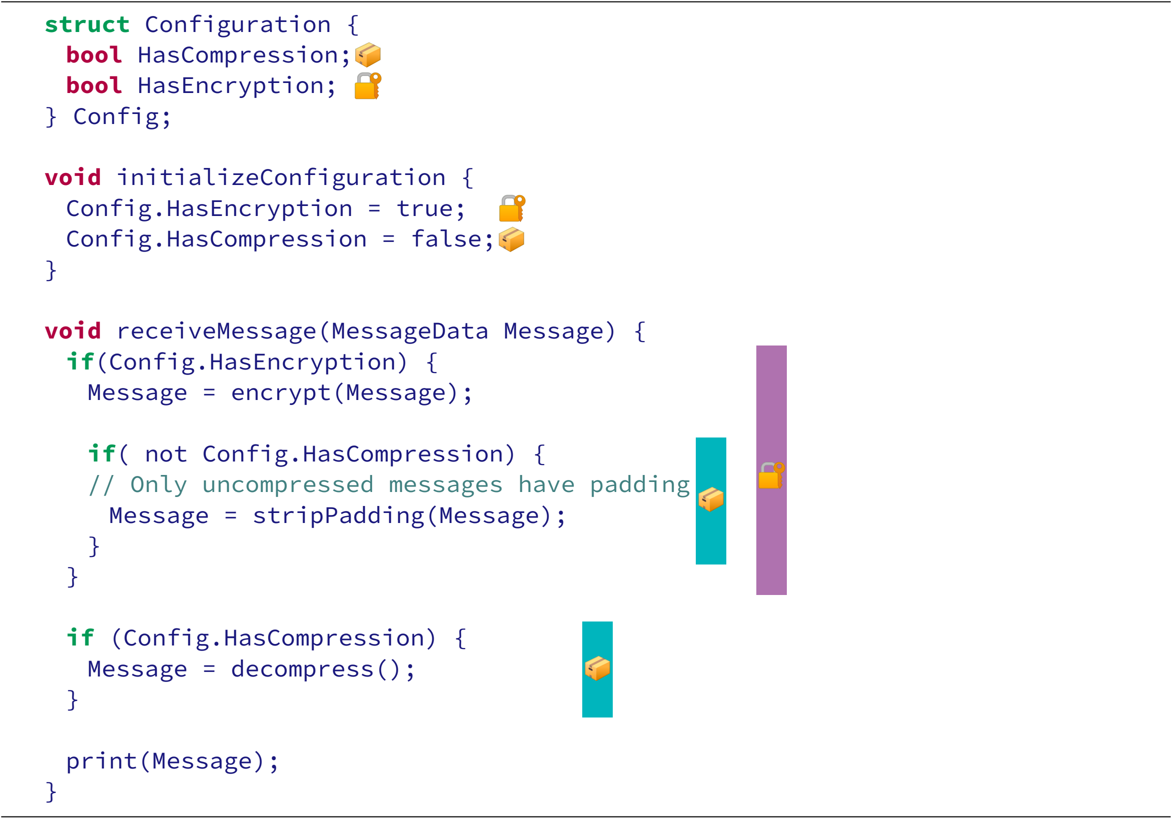
Figure 2.3: Conceptual visualization of a feature trace. The example program contains the feature regions (`FR(...)`) for the features *compression*(📦) and *encryption*(🔐).

The access to the system internals also allow white-box models to build more accurate PIMs. By identifying the exact regions in the code that belong to a feature, profilers can insert profiling instructions which allow to build a *feature trace* when instrumenting a program. Figure 2.3 shows a visualization for such a trace. Using the feature trace one can reconstruct the time spent in individual features and interactions. VaRA is a framework that can produce such feature traces. Section 2.5 introduces VaRA and its capabilities in more details. While black-box profilers rely on measuring multiple system variants to create a PIM, a white-box approach can, in theory, build a simple PIM with just one pass of a program. However, of course, also this model cannot guarantee to represent the performance behavior of a configurable software system entirely correct, as the overall performance of a system often also depends on factors besides the configuration, such as the specific workload as it can have impact on the control-flow [38, 50]. In addition, a white-box approach also introduces an additional overhead, both through the potential additional instrumentation code, and in the effort required to provide a mapping between features and code.

In summary, white-box approaches for configuration-aware performance profiling might not be the best fit for an overall performance model of a configurable system. They can, however, be a useful tool to compare two individual runs of a program, that may differ in their version or workload, to identify the specific performance differences between the two runs on a feature level.

2.5 VaRA

Listing 2.11: Visualization of feature regions for a simple code snippet



```
struct Configuration {
  bool HasCompression;
  bool HasEncryption;
} Config;

void initializeConfiguration {
  Config.HasEncryption = true;
  Config.HasCompression = false;
}

void receiveMessage(MessageData Message) {
  if(Config.HasEncryption) {
    Message = encrypt(Message);

    if( not Config.HasCompression) {
    // Only uncompressed messages have padding
      Message = stripPadding(Message);
    }
  }

  if (Config.HasCompression) {
    Message = decompress();
  }

  print(Message);
}
```

This section gives an overview of the VaRA framework that enables a region-aware analysis of source code. We start with a general overview of the concept of code regions, followed by a more technical overview of how VaRA uses code regions to make programs configuration aware. We conclude the VaRA specific aspects with an overview of the challenges and limitations of the framework. Lastly we provide a brief overview of the VaRA-Tool-Suite, which provides an additional tool suite to manage and run experiments and analyses based on VaRA.

VaRA (*Variability-aware Region Analysis*) is an extension to the LLVM `clang` compiler, whose main goal is to enable various analyses on the basis of *code regions*. The core concept of a *Code Region* is that it encapsulates a block of code that conveys a specific semantic meaning to a researcher. For example, a *commit region* captures a code section that belongs to a specific commit. For the scope of this thesis however, our main focus is on *feature regions*. As the name suggests, feature regions encapsulate code that belong to a specific feature. Listing 2.11 shows an example of a code snippet where we have different feature regions belonging to the features *encryption* 🔐 and *compression* 📦(Regions spanning multiple lines are highlighted in different colors). The code example also shows how the features interact in places where the respective feature regions overlap.

To build these code regions, VaRA adds their own analyses passes to the LLVM compiler infrastructure. Initially, VaRA starts with a pre-defined set of code locations from which it starts building the feature regions. There are two options to provide this initial feature informations to the VaRA framework. One option is to provide an external Extended Markup Language (XML) file, that contains a feature model for the program that is compiled. In this XML file, each feature can have a list of code locations that belong to the specified feature. These code location can correspond to e. g., classes, function or variables in the code. In the example given above, the initial code locations could be, for example the variables `UseCompression` and `UseEncryption`. Another option is to annotate the respective variables, functions or classes directly in the source code. For this purpose, one can add *attributes* to a C++ program. Listing 2.12 shows an example of this technique.

Listing 2.12: Example how C++ attributes can annotate feature variables directly in code

```
__attribute__((feature_variable("ENCRYPTION"))) bool useEncryption;

// Some other code

if(useEncryption){
  ...
}
```

In the LLVM framework, the source code compiles into an LLVM-specific Intermediate Representation (IR). Developers can now provide their own passes that operate on this LLVM-IR to e. g., implement code optimizations. By building the passes for the LLVM-IR, they are independent of the original source code language and can be easily re-used. VaRA adds static analysis passes that operate on the LLVM-IR to build feature regions form an initial small set of individual feature locations. For this, VaRA uses PhASAR[10] in the background. PhASAR is a LLVM-based framework that focusses on inter-procedural static analyses. VaRA adapts

10 `https://phasar.org/`

and extends the analyses that PhASAR provides to lift their analyses to code regions. After identifying the code regions in a program, VaRA can insert specific hooks at the beginning (entries) and ends(exits) of a feature region. Profilers and other analysis tools can later use these hooks for dynamic analyses.

While VaRA provides some powerful capabilities to identify and instrument code regions, it does have some limitations. First of all, at the moment it can only analyse C and C++ programs. While most of the static analyses run on the LLVM-IR level, some of the capabilities also require specific changes to the LLVM front-end. At the time of writing this thesis the only available front-ends that supports VaRA are `clang/clang++`. Furthermore VaRA may not be able to correctly identify all relevant feature regions. One reason for this can be compiler optimizations that run before the feature detection passes. Another reason is the instruction threshold uses internally. Internally VaRA uses an instruction threshold to decide whether a function is relevant for analysis. By default this limit is 100 instructions, but can be adapted by the user. However at the moment, VaRA only employs a simple estimation of the instruction count by using the number of LLVM-IR instructions in a region. In a most extreme example this could result in a region, that only contains a single `call` instruction. VaRA would only count this as a single instruction, regardless of the target function of the `call`.

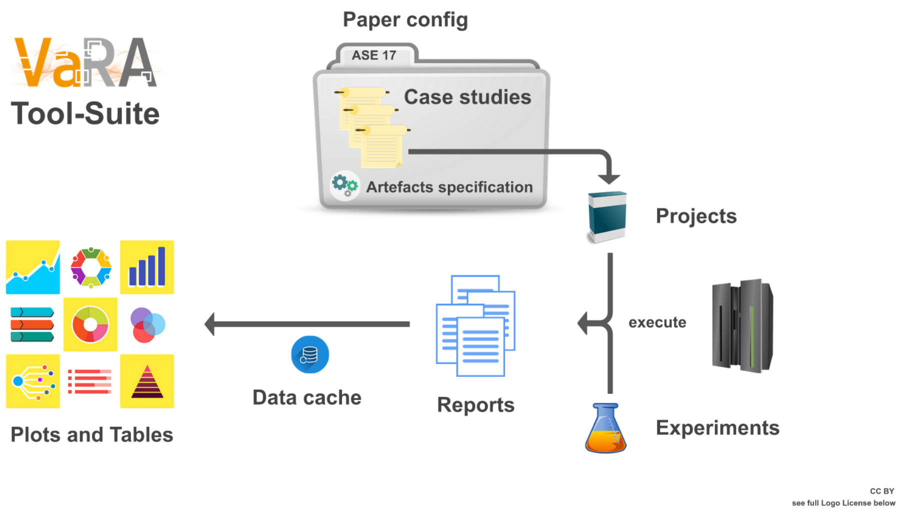


Figure 2.4: Overview of the VaRA-Tool-Suite pipeline[11]

VaRA provides an additional set of tools, the "VaRA-Tool-Suite" to manage reproducible experiments and analyses of the results thereof. Figure 2.4 provides an overview of the general pipeline. The VaRA-Tool-Suite allows to define a common experiment and project structure to make it easy to consistently reproduce results. We add our subject systems to the VaRA-Tool-Suite as new projects and also define our experiments. We furthermore implement our evaluation pipeline with the VaRA-Tool-Suite which produces most of the tables and plots in this thesis. In our accompanying material, we provide the raw data of our experiment results as well as instructions how one can reproduce the tables and plots from this thesis using the VaRA-Tool-Suite.

11 https://vara.readthedocs.io/en/vara-dev/#logo-license

3 Methodology

In this chapter, we present our research questions and give an overview of the methodology we use to evaluate our research questions. We present a detailed description of each research questions, as well as a description of our subject systems, the profilers we consider and regressions that we evaluate.

3.1 Research Questions

Enabling profilers to be configuration-aware by enhancing them with feature-specific information is a new and relatively unexplored domain. To use performance profilers to their fullest potential, one needs to be aware of its capabilities and even more importantly, its limitations. The goal of our research questions is to give a better understanding of what configuration-aware profilers are capable of. We specifically focus our research questions on the *sensitivity*, *precision*, *recall*, and *accuracy* of configuration-aware white-box profilers.

***Sensitivity* (*RQ_1*)** In our context, *sensitivity* describes how much impact a regression needs to have on the execution time of a program in order to be detected. We refer to this as the *severity* of a regression. With this in mind, we define:

Definition 1. The *sensitivity* of a profiler describes, at what severity a profiler is able to detect a performance regression

Knowing this is imperative to choosing a profiler for a proper performance analysis. For example when one wants to build a detailed model of all functions and components that participate in a programs execution time, a profiler with low sensitivity should not be the tool of choice. On the other hand when integrating performance profiling into a continuous integration pipeline to automatically detect performance regressions a profiler with high sensitivity might induce too much overhead.

Motivated by this we formulate our first research question:

***RQ_1*:** *How sensitive are configuration-aware performance profilers in detecting performance regressions in compile-time configurable systems?*

For this research question we perform a comparison between a black-box profiler and configuration-aware white-box profilers. Besides identifying the sensitivity of configuration-aware white-box profilers in general, we also compare the results of our black-box and white-box evaluation. This comparison provides us with information about whether configuration-aware profilers can provide us with qualitative information about the occurrence of a regression even though it only considers the feature specific code. This can in effect identify whether a regression is configuration-specific or not.

Precision & Recall **(*RQ_2*)** With this thesis, we also provide first insights on how useful configuration-aware profilers are in practice. One aspect of this is whether configuration-aware profilers are able to attribute feature-specific regressions to the correct features, which is particularly relevant, as a correct attribution achieves multiple benefits. First, it makes it possible to isolate the code in question, i. e., identifying the cause of the regression. Second, one can select the proper developer to investigate the issue. Especially in highly specialized software such as the HPC domain, it is likely that experts are involved in the development of specific features. When knowing the feature that regresses right away one can directly involve them in the process of fixing a regression. Similarly to this, if a configuration-aware profiler attributes a regression to the wrong feature, this can lead to misallocation of time and resources. Motivated by this, we formulate our second research question:

RQ_2: *With what precision and recall can configuration-aware performance profilers attribute performance regressions in compile-time configurable software systems to specific features or feature interactions?*

Accuracy **(*RQ_3*)** When analysing performance changes, it is also important that a profiler provides accurate measurements. To define accuracy in our evaluation we follow the description by Lilja[44] and adapt it to our domain of performance profilers:

Definition 2. The *accuracy* of a profiler describes the absolute difference between the mean value of a performance measurement obtained from a series of test results and a reference value.

The accuracy of the measurements have effect both in the validity of our other experiment results and the applicability of configuration-aware profilers in practice. For our experiment results, a low accuracy could lead to both features being incorrectly detected as regression, or regressed features not being identified as regressions (Depending on whether the inaccuracy leads to higher or lower measured times). In practice an accurate profiler is useful to properly prioritize regressions. An inaccurate profiler could lead to wrong features being prioritized for further investigation and bug-fixing. We ask:

RQ_3: *How accurate can configuration-aware performance profilers measure feature-specific performance changes in compile-time configurable systems?*

Chapter 4 presents a more thorough description of the experiment design we use to evaluate these research questions and how we operationalise them.

3.2 Subject Systems

In the following section we give an overview of our subject systems. We divide those into two groups: *synthetic* subject systems in which we implement small configurable software systems ourself, and *real-world* subject systems in which we use existing libraries and frameworks that are also used in practice. We specifically focus on applications from the HPC domain, as performance analysis is especially relevant in that field, as even small regressions can accumulate to a non-neglectable overhead.

For each of our subject systems we provide a brief description and an overview of the configuration options that can be used for them. Table 3.1 gives an overview of all our subject systems as well as the number of features and configurations we consider for the experiments.

Table 3.1: Overview of our subject systems. For each subject system we list the number of individual features present in them and the number of configurations we consider for them. Additionally we provide the number of synthetic regressions we have available for each subject system. Note that for each regression we have five different severities we can choose from.

	Type	# Features	# Configurations	# Regressions
Dune	Real-World	11	21	16
HyTeG	Real-World	5	12	5
CTCRTP	Synthetic	6	8	37
CTPolicies	Synthetic	6	12	12
CTSpecialization	Synthetic	6	8	13
CTTraits	Synthetic	4	16	13

3.2.1 Synthetic Subject Systems

Our synthetic subject system consist of small, hand-crafted implementation units with varying number of features and complexity. Our implementations specifically differ in how we use compile-time implementation techniques to realise variability. One main benefit of providing our own synthetic subject systems is that it allows us to provide a much more controlled evaluation environment. Real-world software often uses multiple implementation techniques in orchestration to realise a desired behavior. However, this combination of implementation techniques can impact the analysis capabilities of VaRA. One possible cause for this is that the analysis may not correctly identify certain complex code constructs. In this case, the limits of our analysis frameworks would negatively impact our evaluation.

To minimize these influences of external factors, our synthetic subject systems contain of small, isolated components that focus on a specific compile-time implementation technique. Moreover, our synthetic subject systems do not implement any complex logic but rather just emulate different operations with busy waiting loops.

We provide four different subject systems where each is driven by a different template implementation technique. Section 2.1.3 gives a more technical overview of these different template implementation techniques. We provide one subject system each using *Template Specialization* (*CTSpecialization*), *Traits* (*CTTraits*), *Policy Classes* (*CTPolicies*) and *CRTP* (*CTCRTP*). Our synthetic subject systems are part of a larger repository that hosts also other synthetic subject systems. VaRA uses these other subject systems for demonstration and testing purposes of specific analyses properties such as, field- or flow-sensitivity[1].

3.2.2 Real-World Subject Systems

This section provides an overview of our real-world subject systems. For these subject systems we focus HPC systems, as performance analyses are especially important for that domain. HPC applications are often deployed on massive compute clusters or run multiple simulations in parallel. Reasons for this can either be that the task at hand itself is so computationally complex that parallelizing is one measure to make it computationally feasible. Another reason for this massive parallelization could be that different variants of the same problem, such as slightly differing physical designs of an jet engine outlet, are run in parallel to find a more efficient configuration.

Due to this massive parallelization, performance analyses plays a significant role for these systems, as even small regressions quickly accumulate to a non-neglectable overhead in the total execution time. On the other hand, an accurate performance analysis can guide developers to find performance hot-spots and bottlenecks. Developers can use these informations to further fine-tune the performance of an application.

To limit the selection of our real-world subject systems we enforce specific requirements for our selection. First, we limit ourselves to systems that are implemented in C++. This limitation arises from the VaRA framework, which at the moment can only analyse applications written in C and C++. Furthermore, as we specifically investigate compile-time variability using C++ templates, we only consider systems of which we know that they implement variability using templates. Due to the potential complexity of systems we limit our selection to software systems for which we have a direct point of contact available. This point of contact helps us to understand the complex software architecture of the systems at hand and can additionally guide us in the identification of features of interest. After an initial evaluation, two software systems fulfil these requirements that we present in the following paragraphs.

Dune Dune stands *Distributed and Unified Numerics Environment* and is a C++-based open-source software framework. Dune solves partial differential equations under the use of grid-based numerical solutions. One of the main design goals of Dune is a clear separation of data structures and algorithms. This allows for a flexible and extensible development and makes it possible to freely combine different algorithm and grid types to build a tailor-made solution. To implement this variability Dune makes heavy use of generic programming techniques, such as C++ templates. One of the reason for this is to minimize the overhead of the necessary abstractions at compile-time. This strict design of Dune allows the different components to act mostly independently — solver components can be combined with different kinds of grids which themselves again are inter-compatible with different grid makers. For a more detailed overview of the architecture of Dune we refer to the relevant publications available dedicated to this topic[7, 57].

We selected Dune as part of our real-world subject systems for three main reasons:

1 https://github.com/se-sic/FeaturePerfCSCollection

1. As an HPC framework a sensitive and accurate performance analysis is imperative for its' evolution
2. It implements variability using C++ templates
3. We have a direct point of contact to one of the main developers

For our experiments we implement our own small project using the Dune framework[2]. The basis for this project is a performance test case from the original Dune project which solves a *Poisson equation*[3] with different types of solvers, grids and pre-conditioners. In our project, we separate the test case into individual source files for the different grid types. In addition we make some small adaptions that allow us to easily switch between different grids, solvers, pre-conditioners and grid makers.

HyTeG HyTeG(*Hybrid Tetrahedral Grids*) is our second real world subject systems we evaluate. HyTeG is a framework for high performance finite element simulations so it operates in a similar domain as Dune. HyTeG implements variability both at compile-time through C++ templates, but also uses load-time variability to e. g., switch between different implementations based on command-line parameters. Likewise to Dune, we have contact to one of the developers of HyTeG who assist us in framework-specific issues that arise during implementation of our case study.

One of the main design goals of HyTeG is to provide a flexible, extensible software framework with the capability of providing capabilities of massive parallel simulations. HyTeG is modular in its' core design to allow the user to tailor the framework in a way optimal for the application scenario. Internally, HyTeG operates on partitioned, hierarchical grids. Through this partitioning, HyTeG can employ dynamic load balancing to fully utilise parallel computing capabilities[36]. By design, HyTeG is highly performance sensitive which makes it a suitable candidate for our evaluation. Frameworks like HyTeG rely on precise and accurate performance profiling capabilities to properly identify the root cause of performance regressions.

The example we use for our experiments with HyTeG we work on our own fork of the HyTeG development repository[4]. We choose this approach as it allows us to make modifications to the codebase if it is necessary for our setup. The basis for our evaluations is based on a small profiling application that is part of the HyTeG examples[5]. Similar to the Dune example, the profiling example of HyTeG also solves a small Poisson equation. The example allows to alternate between different solver and smoother implementations.

3.3 Profiling Approaches

In this thesis we evaluate three different profiling approaches (for brevity, we refer to them as "profilers" from here on). Two of these profilers are based on existing technologies (See

2 https://github.com/se-sic/dune-VaRA
3 For more details on the Poisson equation and its' applications we refer the reader to the corresponding literature[12, 33]
4 https://github.com/se-sic/hyteg-VaRA/
5 https://github.com/se-sic/hyteg-VaRA/blob/master/apps/profiling/ProfilingApp.cpp

Section 2.3), while one is a custom profiler that VaRA provides. This section gives a brief overview of how we utilize VaRA to enable feature-aware profiling with the three approaches.

LLVM XRay (VXRay) The first profiler we evaluate is based on the LLVM XRay profiling framework. We will refer to is as VXRay (VaRA XRay) through the remainder of this thesis. As Section 2.3.1 already outlines, the LLVM XRay profiler inserts "no-op sleds" at the entry and exit of function calls. VaRA now uses a similar approach to enable configuration-aware profiling. It inserts calls to profiling functions to the beginning and end of each feature region. As opposed to the original XRay profiling, the feature-aware profiling it is not possible to dynamically en- and disable profiling. In addition, the feature-aware profiling of VXRay produces a separate trace-file than the standard function tracing of XRay. However, as the instrumentations that VaRA inserts use the same underlying APIs of the XRay framework, it ensures that e. g., the timestamps both rely on the same reference. With this it is easily possible to merge the feature-traces and function-traces to interpret them in unison.

For our evaluation however, our only interest is the feature-specific trace file that VXRay produces. This trace file also uses the TEF to store events about feature region entries and exits. The information about the times of feature region entries and exits allows to reconstruct the time spent in each feature and interaction of features. By reconstructing the time spent in each feature, we can evaluate whether a regression occurs for a specific feature.

eBPF (eBPFTrace) To enable configuration-aware performance profiling with the eBPF framework, VaRA inserts custom Statically Defined Tracing (SDT) probes into the binary that mark the beginning and end of a feature region. These probes themselves are anchor points in the executable realized as `nop` instructions. VaRA then adds a map from feature-regions to specific addresses of these anchors, along some additional information, into the static section of a binary. This setup allows to attach a eBPF-based profiler to the binary and collect feature information.

In our case we make use of the SDT probes VaRA inserts into the binary and use the `bpftrace` profiler to attach to them. We will refer to this custom eBPF based profiler as eBPFTrace for the remainder of this thesis. With the SDT probes that VaRA inserts, eBPFTrace registers each entry and exit to a feature region and creates a feature trace of a program run. Interpreting these traces is similar as it is for VXRay as they come in a similar format and representation.

PIMTracer The last profiler we consider is the *performance influence model tracer* (PIMTracer), that is custom profiler embedded into VaRA. The design of PIMTracer follows state-of-the-art proposals from literature[59, 68]. The PIMTracer collects, for a single program run, a trace that directly represents a PIM. During execution, PIMTracer persists a dynamic feature-stack (to track the currently active regions), by tracking entries and exits to a region. By tracking the timestamps of the individual entries and exits and the current state of the feature stack, PIMTracer automatically aggregates the times spent in each feature and interaction. At the end of a program run it creates a corresponding trace that summarizes the total time spent in each feature. To enable profiling with PIMTracer, VaRA inserts the respective instrumentation code directly into the binary.

3.4 Regressions

In our experiments, we introduce one or multiple regressions into a software system to evaluate if and how well our configuration-aware white-box profilers can detect them. Throughout this thesis we only consider regressions that affect the execution time of a program and not other performance measures such as memory consumption or throughput. All regressions we introduce are of a synthetic nature. From an implementation perspective they simply introduce a busy loop at a chosen point into the program code that runs for a pre-determined time. To introduce a regression into our case studies, we use the `patch` functionality of Git, which allows to easily introduce small code changes into our subject systems. An example for a software systems before and after introducing an artificial regression is shown in Listing 3.1 shows an example of a code snippet where we introduce a synthetic regression of 100 ms into the function `foo`.

Listing 3.1: Example how we introduce synthetic regressions into our subject systems. A `git patch` adds the green lines to the program before compilation. The argument to the function call controls the severity of the regression.

```cpp
namespace fp_util {
 void busy_sleep_for_msecs(unsigned MSecs){
   auto start_us = std::chrono::duration_cast<std::chrono::microseconds>(
         std::chrono::high_resolution_clock::now().time_since_epoch());
   auto end_us = start_us + std::chrono::milliseconds(MSecs);
   auto current_us = start_us;

   while (current_us < end_us) {
     for (long counter = 0; counter < 100'000; ++counter) {
      asm volatile("" : "+g"(counter) : :); // prevent optimization
     }
     current_us = std::chrono::duration_cast<std::chrono::microseconds>(
     std::chrono::high_resolution_clock::now().time_since_epoch());
   }
 }
}

void foo() {
 fp_util::busy_sleep_for_msecs(100);
 prepare();
 performOperation1();
 performOperation2();
 finalize()
}
```

This approach offers both flexibility and precision for our evaluation. It is flexible with respect that a regression can be introduced at an arbitrary location in the source code as patches are easy to create and apply. This is important for our use cases as we specifically focus on feature-specific regressions. With the flexible patching approach, we can easily introduce a regression into a specific feature region of the code. An additional benefit of this approach is that regressions in different feature regions can often be freely combined, especially if the regressions affect distinct features or source files. Our approach also allows us to precisely control the severity of the regression up to a millisecond precision. Controlling that parameter is especially important when we evaluate the sensitivity (RQ_1) and accuracy(RQ_3) of the different profiling approaches. For each of our subject systems we identify possible points of interest that are within a feature region to create appropriate patches. For each point of interest we then create patches of five different severities: 1 ms, 10 ms, 100 ms, 1000 ms and 10 000 ms.

4

Experiments

The following chapter describes our experiment setup through which we answer our research questions. We first provide an overview of our general experiment setup, followed by a more detailed description of the experiments specific to our research questions. Furthermore, we provide a description of how we operationalise our research questions and outline our expectations for each of our research questions.

4.1 Setup

For all experiments, we use a similar pipeline to collect our measurements. Figure 4.1 provides a conceptual overview of this pipeline. For each of our subject systems, we collect performance measurements from both a black-box profiler and multiple white-box profilers. For the black-box compiler we collect the full execution time of a program, while for our white-box profiers, we collect data about the time spent in individual features.

For each of our subject systems, we create different *base variants*(c_b). These base variants differ with respect to the selected features of the subject system at hand. From a code perspective, these different base variants differ only in the selected template parameters used in the final implementation. Depending on the selected features for a base variant, a pre-defined set of regressions are available, that can be applied to the base variants code to create respective *regressed variants*(c_r). Section 3.4 gives an overview of our regressions and how we introduce them into our subject systems. Our individual experiments differ in two points: First, the different experiments use different strategies to select regressions to create the regressed variant c_r. Second, in each experiment we perform a different evaluation of the data we collect. Section 4.2 gives a more detailed overview, for each of our experiments, how we perform these steps.

A base variant and its respective regressed variants serve as basis for the evaluation. For each base and regressed variant, we first perform a black-box performance measurement using the `gnu time`[1] utility, which measures the execution time of a program. While `gnu time` can also report other statistics, such as memory usage and I/O operations, for our evaluation we only consider execution time. We use the black-box measurements to establish a baseline that we can compare our approaches against as black-box performance profiling techniques are well established[5, 27, 37, 42, 49].

The main focus of this thesis however, is in an evaluation of configuration-aware white-box profilers. We want to explore the capabilities and limitations of configuration-aware white-box

1 https://man7.org/linux/man-pages/man1/time.1.html

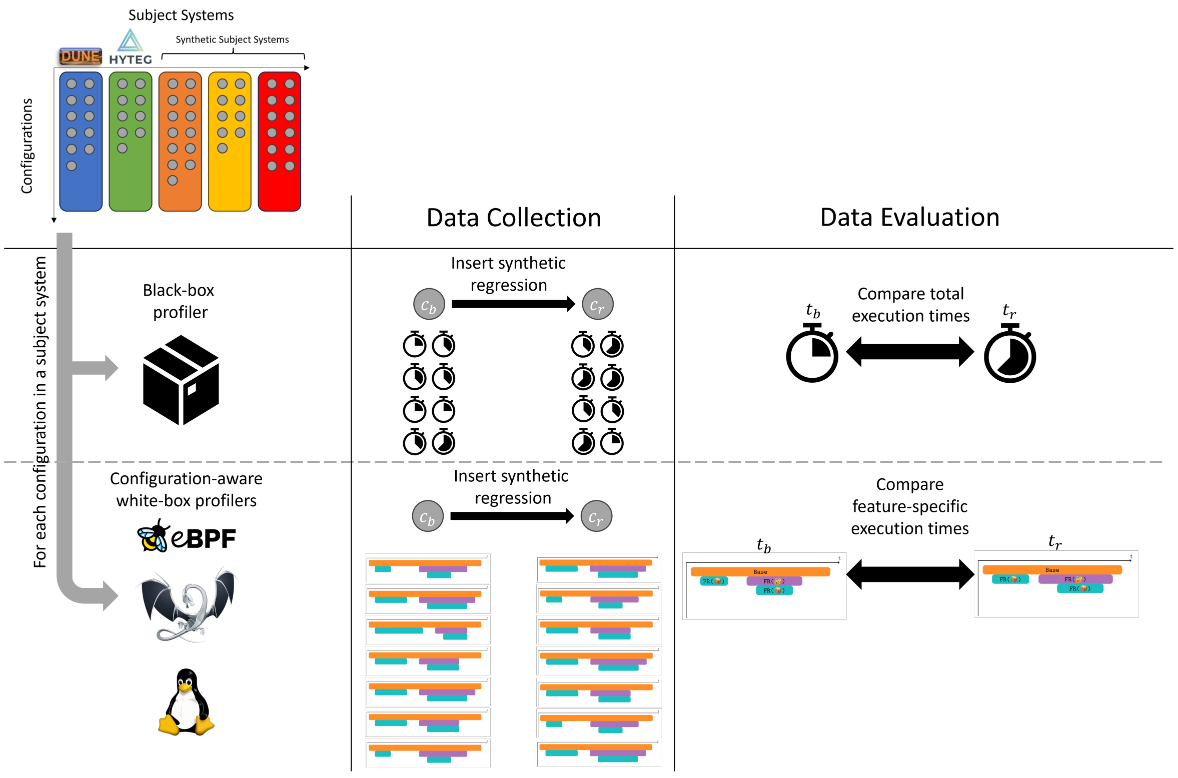

Subject Systems
DUNE
HYTEG
Synthetic Subject Systems
Configurations
Data Collection
Data Evaluation
For each configuration in a subject system
Black-box profiler
Insert synthetic regression
c_b
c_r
Compare total execution times
t_b
t_r
Configuration-aware white-box profilers
eBPF
Insert synthetic regression
Compare feature-specific execution times

Figure 4.1: Overview of our general experiment pipeline

profilers on with regards to regression detection. For this reason, we also run performance measurements using different white-box profilers. We use the VaRA framework to enhance the different profilers with feature-specific information in order to conduct a feature-aware white-box performance analysis. As features are the main building blocks for configurations, this in effect enables a configuration-aware performance analysis. Section 2.3 provides a detailed overview of the different profilers we evaluate while Section 3.3 describes how VaRA enhances these profilers with feature information. The benefit of these white-box profiling approaches is now that it provides a more fine-grained overview of a programs execution. We determine, by analysing the output trace files that these profilers generate, the time that is spent in each individual feature. This allows us to create an accurate PIM for each of our variants. We use these PIMs as a basis for our feature-aware regression detection.

Feature-Specific Ground Truth To correctly evaluate where our regression have impact on a programs' performance, we also need an additional *feature-specific* ground truth. This ground truth needs to provide us with information about which features or interactions our synthetic regressions can affect and also how often these are affected. This information helps us for a better interpretation of the results of our other experiments.

To determine which features our regressions affect, we design a separate experiment. Our experiment design follows the idea that each of our synthetic regressions introduces a specific feature "detection" that we can identify in a feature trace. Such a setup allows us to answer two questions:

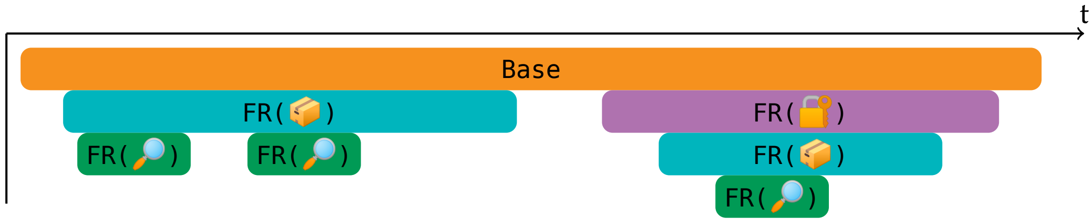


Figure 4.2: Example of the feature trace of Figure 2.3 after introducing our `detect` function. All regions that `FR(`🔎`)` interacts with are affected by a corresponding patch.

1. Which features does our regression affect?
2. How often does our regression affect specific features?

Take, for example, the trace of Figure 4.2, in which we introduce the special detection feature 🔎. To determine which features our regression affects we now simply determine all feature interactions between our detection feature and other feature regions. In the example trace, our detection feature interacts with the feature compression (📦) and the interaction of compression (📦) and encryption (🔐). Therefore, we conclude that these are the feature (interactions) we expect to regress for the patch. Furthermore, through the trace we determine that the 📦, 🔐 interaction only interacts with the detection feature once, while the single compression feature interacts twice. Therefore, if we now assume that our regression has a severity of, e. g., 1000 ms for each occurrence, in our regressed variant we expect to measure a regression of 2000 ms for the compression feature and 1000 ms for the interaction of compression and encryption. On an implementation level, we realize this detection through the introduction of a `detect` function, that is annotated with a special feature variable. Through different individual patches, we introduce the `detect` function in locations where our regression patches usually introduce a synthetic regressions.

Listing 4.1: Conceptual example of how we add our `detect()` function into the subject systems. A `git patch` introduces the green lines before compilation.

```
namespace fp_util {
  __attribute__((feature_variable("__VARA__DETECT__"))) void detect() {
    long foo = 0;
    asm volatile("" : "+g"(foo) : :);
    foo++;
  }
}

void foo() {
  fp_util::detect();
  prepare();
  performOperation1();
  performOperation2();
  finalize()
}
```

Listing 4.1 shows an example how a patch introduces the `detect` function into code. To build the feature-specific ground truth for each regression patch, we compile the code that is patched with the `detect` function with VaRA and its support for VXRay, such that we can produce a feature-aware trace of the program execution. Analysing this feature trace allows us to identify the affected regions and build the feature-specific ground truth.

Minimizing External Influences External influences such as system noise, different hardware setups or external processes can all interfere with performance measurements[39]. For our performance measurements, we need to ensure that we minimize these influences, as they otherwise might influence our performance measurements which can lead to wrong interpretations of our results. To reduce the effect of the system noise, we perform 30 repetitions for all our measurements and use the mean value of measurements for our evaluation. In addition we run all our experiments on a slurm[2] cluster. Through the slurm infrastructure we ensure several properties: First, we ensure that all our experiments are run on cluster nodes with identical physical resources as well as the same software setup. This reduces the risk of any fluctuations that may be caused by different hardware resources or varying software packages and versions. In addition, slurm also ensures that our experiments are run in isolation on a cluster node. That means when we run an experiment no other compute job can be run on this cluster at the same time. Every cluster node consists of an AMD EPYC72F3@3.70 GHz CPU with 256GB of RAM, running a minimal Debian 11.

4.2 Operationalization

In the following section, we give an in-depth explanation how we operationalize our three research questions. For each research question, we present our experiment design, as well as how we evaluate the collected performance measurements to answer the research questions.

4.2.1 Operationalization of RQ_1

In RQ_1 our goal is to evaluate the sensitivity of different profilers. For this context, we refer to *sensitivity* as the impact of a regression on the execution time. In order to evaluate the sensitivity, we introduce synthetic regressions into each configuration of our subject system before compilation. We measure the performance of the different configurations of our subject systems once for the unregressed variant and then for each regressed variant individually. One of the main goals of RQ_1 is to identify whether configuration-aware white-box profilers exhibit similar sensitivity characteristics to a black-box profiler even when only considering feature-specific portions of the execution time.

Regression selection For our sensitivity experiment, we inject a single regressions into our subject systems. For each subject system and configuration, we consider the *relevant* patches for a specific configuration. A patch is *relevant* if it introduces a regression into one of the features that is selected by the current configuration. By that selection criterion, we ensure that every regressed variant should in fact change the performance. With our evaluation we are not aiming to find an exact bound for when profilers detect a regression, but rather only

2 `https://slurm.schedmd.com/documentation.html`

consider regression of severities of five different orders of magnitude: 10 000 ms, 1000 ms, 100 ms, 10 ms and 1 ms.

Data Collection Depending on whether we evaluate the black-box profiler or the white-box profilers, we collect the measurement data differently. For our black-box profiler we consider the total execution time in each configuration as basis for the evaluation. As our white-box profilers are configuration-aware they provide us with the feature-specific execution times for each configuration. Hence, for our sensitivity valuation of configuration-aware white-box profilers, we now consider the sum of the execution times that is spent in feature-specific code. By that, we disregard any time that is spent in non-feature code as the goal of our evaluation is to analyse the feature-specific aspects of white-box profilers.

Data Evaluation For the evaluation of our measurements, we compare for each subject system, configuration and regression, the measured times of the base variant c_b and the respective regressed variant c_r. We compare the measurements of c_b and c_r with an independent t-test to identify whether there is a significant change in the measurements. Iff the t-test reports a p-value of $p < 0.05$, we consider this a significant change and therefore account this as a detected regression. For each subject system we then report, grouped by severity, the total number of regressions considered and the relative number of cases where our evaluation successfully identifies a regression.

If we evaluate a regression that has a severity of 100 ms or higher, we use additional absolute and relative thresholds to prevent miss detection due to measurement noise. For the absolute threshold we ignore features where the total times spent in a feature in both the base and regressed variant is below 100 ms, as this is the minimum size we expect for at least the regressed variant. For our relative threshold we ignore features where the execution time difference of a feature between the base variant and the regressed variant is smaller than 1% of the execution time of a feature in the base variant. For the smaller severities of 10 ms and 1 ms, we decided not to use the thresholds as it could lead to actual regressions being ignored and not tested.

4.2.2 Operationalization of RQ_2

For RQ_2, our goal is to evaluate whether a configuration-aware profiler can attribute feature-specific regression to the correct features. Correctly identifying the regressed features provides important information to a development team. Especially in highly specialized applications, such as HPC systems, the development of different features may require specialized domain knowledge which is why specific developers are responsible for their maintenance and evolution. When a regression can be attributed to a specific feature, this helps to directly assign the further investigation and mitigation to the correct developers.

Regression selection With regressions that we consider in RQ_2, we want to cover multiple scenarios: First, we want to cover simple cases where only a single feature regresses. Second, we also want to create scenarios in which multiple different features regress at the same time and evaluate which of these are correctly identified. For this purpose, we create *patch lists* that consist of up to five different regression patches. Each patch list then represents a regression, which affects one or more features, that we introduce to our base variant c_b.

To create the patch lists for a specific subject system and configuration that introduce a regression, we first create a set of up to five patches. From this set, we generate the patch lists by taking the power-set of this set of patches (Excluding the empty set). The patch selection relies on the data from our *feature-specific ground truth* which provides us, for every subject system and configuration, with information about which features a single patch potentially has an influence on. To create the patch set, we follow a multi-step selection process.

In the first step of our patch selection process, we select up to three different patches. All of these three patches shall only affect a single feature and the patches must not affect the same feature. If more than three patches meet this criteria, we select the patches which have the least number of occurrences of the region it affects. The rationale behind that is, that if a patch affect a region multiple times this can indicate multiple entries and exits into the surrounding feature region. This induces additional profiling overhead. With our selection strategy we minimize the potential profiling overhead. If we have less than three patches that affect disjunct features, we select all available.

In the second phase of our patch selection process, we focus on patches that affect more than one feature in the current configuration. All patches that we select in the second phase should (a) affect at least one of the features also affected by one of the patches from the first phase, and (b) also add at least one new affected features. If more patches meet this criterion than we need to reach our total of five patches in our patch set, we prefer patches that add more new affected features first.

If our patch set does not contain five patches after the first two steps, we perform a last third selection of possible patches. In this final step, we greedily select patches based on the number of different features they affect. Therefore, we do not enforce any requirements of interaction or overlap with patches from the first two phases, but rather only select patches that affect many different features.

We have decided for the described patch selection process for multiple reasons. For starters the selection process is easy to implement, using the data we gather from our feature-specific ground truth experiment. Additionally our approach attempts to create patch sets with patches that interact in a manner, such that multiple patches can affect the performance of the same feature while at the same time, also affecting multiple different features. There are some cases where our approach cannot ensure this, for example when the first phase would not select any patches. By our last greedy selection phase however, we still ensure that we select some patches in any case. While a more sophisticated approaches that, for example, would ensure interactions for all kinds of scenarios, could lead to a more diverse landscape of covered scenarios, these are also complex to implement and can have other defects, where trying to ensure an interaction can lead to an empty patch set.

Data collection As for RQ_2 our interest is in feature-specific regressions, we only collect data from our three configuration-aware white-box profilers, as our black-box profiler cannot provide us with any feature-specific information. For each subject system and configuration, we profile the execution of the base variant c_b and each regressed variant c_r. From each of our white-box measurements, we can create a PIM which allows us to identify the time spent in each feature and feature interaction. These individual feature-specific times serve as a basis for our evaluation.

Data evaluation For the evaluation, of our RQ_2 we now compare the times spent in each feature or interaction individually, for each subject system and configuration. For each feature or interaction we determine whether there is a significant difference using an independent t-test. We consider a regression to be detected iff $p < 0.05$. Similar to the data evaluation from RQ_1, we also employ an additional absolute threshold of 100 ms and a relative threshold of 1%. From our feature-specific ground truth, we furthermore obtain a set of features for which we expect to detect a regression for a given subject system, configuration and patch list. Combining the data from our statistical tests and the feature-specific ground truth, we can determine the true positives(TP), false positives(FP), true negatives(TN) and false negatives(TN). For our context, a true positive is a regression that is both detected by a configuration-aware profiler and also predicted by our feature-specific ground truth. The classification for FP, TN, and FN follows analogously. This allows us calculate the precision (PPV) and recall (TPR) for each patch list of a configuration in a subject system:

$$PPV = \frac{TP}{TP + FP}$$
$$TPR = \frac{TP}{TP + FN}$$

For each of our subject systems we report the means for precision and recall across all configurations and patch lists. We report these separately for all three profilers.

4.2.3 Operationalization of RQ_3

Our goal for RQ_3 is to evaluate how close the measured change in execution time is in comparison to the expected change in execution time for our configuration-aware profilers. To calculate this, we compare the execution times and after introducing a performance regression, both from a black-box and white-box perspective.

Regression selection To select and introduce regressions, we use the same approach as in RQ_2. That is, we create patch lists consisting of up to five patches. We apply all patches in a patch list to our subject system to create our regressed variant. Section 4.2.2 gives a detailed description of the selection process.

Data collection To evaluate the accuracy of the different profilers, we collect the execution times before and after introducing a regression to determine the differences in execution time. For our black-box profiler, we determine the difference for the overall execution time. For our white-box profilers we calculate multiple differences: First, we calculate the total difference in execution time when considering *all* feature-specific code. This measure gives us an overview of the total feature-related performance difference. With this data, we can compare our accuracies with these from a black-box profiler. In addition, for our white-box profilers, we also calculate the mean differences for each individual feature and interaction of a program run. This feature-specific performance differences allow us to identify whether different white-box profilers exhibit different performance characteristics in a fine-grained feature-specific performance analysis.

Data evaluation We evaluate the accuracy of our profilers by introducing three error measures. These error measures specify a relative difference between the expected regression and the measured regression. In general the errors are calculated as:

$$\left|1 - \frac{|\delta_m - \delta_e|}{\delta_e}\right|$$

where δ_e specifies the expected regression and δ_m the measured regressions. For the evaluation, we introduce the following three error measures: the *overall black-box error* (E_b), the *overall feature error* (E_f) and the *feature specific error* (ϵ_f). Generally, our error describes how different the measured values are from our expected values in relative terms to the expected change.

Overall Black-box Error (E_b) We obtain this error by comparing the measurements of our black-box profiler before and after introducing a regression through a patch list. The overall black-box error is specific to a combination of subject system, configuration ID and patch list. We obtain the expected regression δ_e through our ground truth experiment, by accumulating the expected regression for each regressed feature of the corresponding patch list. The measured error δ_m in this case is the difference of the total execution time with and without the patch list applied. This evaluation gives us important insights as it can indicate whether our ground truth is accurate.

Overall Feature Error (E_f) This error measure is similar to the overall black-box error E_b, but for white-box profilers. We obtain the expected regression in the same manner as for E_f. However to obtain δ_m we accumulate, for both the base and regressed variant of the current configuration, all time that is spent in *feature code* only. This way, we evaluate whether white-box profilers can exhibit a similar accuracy as a black-box case when regressions occur only in feature-specific code and not in base-code.

Feature Specific Error (ϵ_f) Lastly we introduce the feature-specific error for the evaluation of our white-box profilers. This error measure investigates the accuracy of white-box profilers on a more fine-grained level. We now consider each affected feature from a patch list individually. For each affected feature, we obtain the expected regression δ_e from our ground truth experiment, which tells us, for a specific patch and configuration, how often a regression has an impact on the execution time. When multiplying this with the severity of a patch we obtain the regression we expect for the current configuration. If multiple patches in a patch list affect the same feature, the expected regression size is the sum of the individual regressions per patch. For the measured regression, we compare the time spent in the feature in question before and after introducing a regression. The goal of the feature-specific error ϵ_f is to evaluate the accuracy of regression measurements for individual features as opposed to all affected features.

4.3 Expectations

In this section, we summarize our expectation for our individual research questions.

Expectations for RQ$_1$ For RQ$_1$ we expect that all profiling approaches detect a regression at least at the most severe level of 10 000 ms, as our unregressed variants all take under 10 *s* to execute. Therefore a regression that more than doubles our total runtime should be detected by all profilers. While our experiment setup tries to minimize the noise and external influences on our measurements, it is more likely that a small regression cannot be differentiated from noise which makes a correct detection more unlikely at lower severities. Therefore, we expect that the detection rates will decrease with lower severities. As our synthetic regressions are all designed to affect feature-specific code, we expect to observe that the white-box profilers perform similar to the black-box profiler, even if they only consider feature specific code.

Due to the fact that white-box profilers only considering a sub-section of the execution times, this could lead to an overall lower influence caused by external factors. Therefore, a clearer distinction between noise and actual regressions is possible which would lead to more detections of regressions for the white-box case. Due to potential different profiling overheads we also expect to observe that some profilers may be able to detect more regressions than others, especially for lower severities. Additionally, we expect that the overall detection rate should be higher for our synthetic case studies, as the high complexity of real world software might have impact on how well the profilers, or the VaRA framework in general, can identify feature regions.

Expectations for RQ$_2$ For RQ$_2$, we expect that most regressions are attributed to the correct feature and interactions by all profiling approaches, at least for the synthetic subject systems. Our rationale is that our regressions are specifically hand-crafted to affect a specific feature and our synthetic subject systems have a simple code structure. For our real-world subject systems it is possible that we observe a lower precision and recall. Possible reasons for this could be that the VaRA framework, or individual profilers, cannot properly identify different feature regions. This could lead to both inconsistencies in our feature specific ground truth, but also to regions not being properly measured in our actual measurements. These insights are important for our evaluation as they provide valuable information about the limitations of our approach.

Furthermore, we expect to observe differences in the precision and recall of different profilers, as the underlying technical implementations work differently. These insight can be valuable for the interpretation of our results. When choosing the appropriate compiler for debugging the performance of an HPC application it is imperative to understand the granularity and precision one can expect.

Expectations for RQ$_3$ For RQ$_3$, we differentiate between different comparisons. For the comparison of the total measured regression sizes between the black-box profiler and white-box profilers there are two things to consider: On the one hand the black-box profiler should capture all performance changes as it considers the whole execution time, whereas our white-box profilers only consider the feature-specific parts. Therefore, if a regression is not correctly attributed to feature code, or if a regression occurs outside of feature code, white-box profilers show a higher error. However, at least for our synthetic subject systems we expect to observe similar values for the overall errors in the black-box and white-box case. Our rationale behind this is, that our regressions are especially designed to only affect feature code and in addition our synthetic subject system have a simple structure such that all feature regions should be identified correctly. For Dune and HyTeG it is more likely that the black-box and white-box measurements differ, as the more complex code structures could lead to issues in the correct feature detection.

For our per-feature evaluation of the feature-specific-error we also expect to observe a difference between our synthetic and real-world subject systems. For the different white-box profilers it is possible that effects such as different sensitivities and profiling overheads lead to different results. In general, we expect the feature-specific error to be higher than the overall feature error, as in the feature-specific case we consider overall smaller portions of the execution time. As our error is a relative measure, a higher absolute error would results in a higher relative error.

A general risk that we have to consider for both RQ_2 and RQ_3 is our feature-specific ground truth. As we obtain this through applying a white-box profiler, it also suffers from the potential challenges of feature-aware white box profiling. Due to potential shortcomings in both the VaRA framework and the profiler we use, there is a risk that our feature-specific ground truth is flawed, for example if features are not detected correctly.

5 Evaluation

5.1 Results

In this section, we present the results from that we collected in our experiments.

5.1.1 Feature-Specific Ground Truth

Table 5.1: Overview table for HyTeG of the feature-specific ground truth experiment. The column names list the affected feature names. If a patch has an effect on a specific configuration ID, we list how often a feature is affected.

Patch name	Config ID	CGSolver	SymmetricSORSmoother	GMResSolver	SORSmoother	MinResSolver
cg_solve_detect	0	2	-	-	-	-
	1	2	-	-	-	-
	6	1	-	-	-	-
	7	1	-	-	-	-
gmres_solve_detect	4	-	-	2	-	-
	5	-	-	2	-	-
	10	-	-	1	-	-
	11	-	-	1	-	-
minres_solve_detect	2	-	-	-	-	2
	3	-	-	-	-	2
	8	-	-	-	-	1
	9	-	-	-	-	1
sor_solve_detect	0	-	-	-	24	-
	2	-	-	-	24	-
	4	-	-	-	24	-
	6	-	-	-	12	-
	8	-	-	-	12	-
	10	-	-	-	12	-
symmetric_solve_detect	1	-	24	-	-	-
	3	-	24	-	-	-
	5	-	24	-	-	-
	7	-	12	-	-	-
	9	-	12	-	-	-
	11	-	12	-	-	-

For our feature-specific ground truth experiment, we collected the respective coverage information for all configurations of our projects. Table 5.1 provides an example of this coverage data for the HyTeG subject system. For each patch name, we list which features they affect in the different configurations. For example, the patch `cg_solve_detect` introduces code that affects the *CGSolver* feature two times in configurations IDs 0 and 1, but only one time for the IDs 6 and 7. For brevity, we excluded rows in which a patch has no impact on any feature.

In the shown example for HyTeG each patch only ever affects one feature and no interactions. We do, however also collect data on feature interactions that occur in the subject systems. For HyTeG there simply were no interactions of the features we selected for observation. In addition, we also observe for our other subject systems that a single patch can affect multiple features or interactions at the same time.

We use the data we collect this way to provide a more informed ground truth for our RQ_2 and RQ_3. With this information we obtain the information for which features we expect to detect a regression and in addition also how much impact the regression has on the execution time. Moreover this information also provides insights that are useful for the interpretation of our sensitivity experiment.

5.1.2 Sensitivity (RQ_1)

Table 5.2 shows the results for our sensitivity experiment. For each subject system, we report the number of regressions($\mathbb{R}$) we introduce in total for each severity. Then, for each of our profiler and regression severity we report a relative number of many regressed variants show a significant difference in execution time to the base variant.

For our synthetic subject systems, we observe that both the black-box profiler and all three of our white-box profilers identify all regressions that have a severity of 10 ms or higher. At a severity of 1 ms the white-box profilers still detect all regressions, while the black-box profiler detects at most 25%.

For our real-world subject system HyTeG we observe similar results as in our synthetic subject systems. A notable difference is though, that the black-box profiler detects 58% of the regressions of a severity of 1 ms. Furthermore, eBPFTrace does not detect all regressions of 1 ms, but still the vast majority of 96%. For our real-world subject system Dune we generally achieve lower detection rates for most cases than in the other subject systems. The only exception is the black-box profiler, which detects a considerable high amount of 90% of regressions for a severity of 1 ms.

We furthermore observe, that the black-box profiler outperforms all three white-box profilers for severities of 100 ms and above. For the smaller severities, we achieve comparable results between the black-box profiler and the white-box profilers, with the exception of eBPFTrace. Generally, eBPFTrace achieves the lowest detection rates of all profilers. Overall however, all white-box profilers still detect over 95% of regressions for most cases, and at least 80% for the few remaining cases, most of which can be attributed to the lower detection rates of eBPFTrace.

5.1.3 Precision and Recall (RQ_2)

Table 5.3 shows a summary of our precision experiments for each subject system and profiler. We observe differences both between the different subject systems and the different subject

Table 5.2: Sensitivity of different profiling approaches with regard to the introduced regression severity in milliseconds. On the left, we show the total amount of regressed program variants that are considered for each regression severity. Furthermore, the table depicts for each profiler the relative amount of regressions that were detected.

		Black-box					VXRay					PIMTracer					eBPFTrace				
	ℝ	1 ms	10 ms	100 ms	1000 ms	10 000 ms	1 ms	10 ms	100 ms	1000 ms	10 000 ms	1 ms	10 ms	100 ms	1000 ms	10 000 ms	1 ms	10 ms	100 ms	1000 ms	10 000 ms
Dune	73	0.90	0.89	0.95	0.99	1.00	1.00	0.99	0.97	0.96	0.96	0.96	0.99	0.88	0.93	0.93	0.84	0.82	0.81	0.82	0.82
HyTeG	24	0.58	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00
CTCRTP	102	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CTPolicies	64	0.25	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CTSpecialization	35	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CTTraits	52	0.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

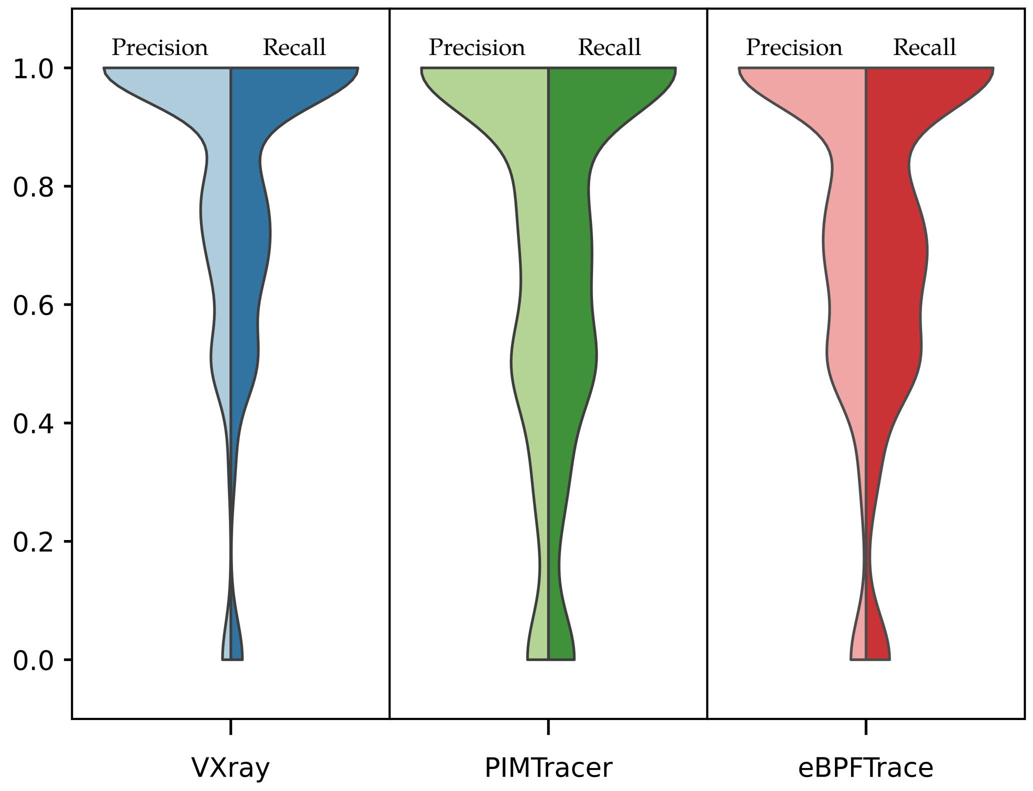


Figure 5.1: Comparison of precision (left) and recall (right) between the individual profilers. The violin plots visualize the likelihoods that a specific precision and recall value was observed.

systems. Overall, for our synthetic subject systems, we achieve a recall and precision values of at least 60%. For three out of our four synthetic subject systems, the differences between the different profilers are at most 10 percentage points. For *CTCRTP* however, we observe a difference of around 30 percentage points between VXRay and the other two white-box profilers. Our synthetic *CTTraits* subject system reports the highest values for both precision and recall among all of our synthetic subject systems.

For our real-world subject systems we observe varying results. For HyTeG, we achieve considerably higher values for precision and recall across all three profilers. Notably, VXRay attributes nearly all regressions to the correct features. While PIMTracer and eBPFTrace achieve slightly lower results for precision, they still attribute at least 86% for precision. Dune generally achieves the lowest values for recall for all subject systems. While Dune generally also reports lower values for precision than HyTeG, in some occurrences the performance is comparable to our subject systems. For the eBPFTrace, Dune achieves a similar precision to our *CTCRTP* and *CTPolicies* subject system. For recall, VXRay and PIMTracer detect around 50% of the regressions correctly, while eBPFTrace is slightly lower at 45%.

Figure 5.1 shows a distribution for precision and recall for all three profilers. In this plot, we consider all patch lists across all subject systems. A distinction by our individual subject systems is shown in Appendix A. Generally the plots show that VXRay seems to perform slightly better than the other profilers when considering all subjects systems. The plot for

Table 5.3: Summary of precision and recall of different white-box profilers with regard to attributing feature-specific regressions to the correct features. For each subject system, we show the total number of regressed features ($\mathbb{F}$) and the means of precision (PPV) and recall (TPR).

		VXRay		PIMTracer		eBPFTrace	
	$\mathbb{F}$	PPV	TPR	PPV	TPR	PPV	TPR
Dune	324	0.59	0.50	0.56	0.49	0.65	0.45
HyTeG	48	0.92	1.00	0.89	1.00	0.86	1.00
CTCRTP	656	0.94	0.92	0.61	0.59	0.64	0.62
CTPolicies	1376	0.70	0.68	0.72	0.69	0.68	0.67
CTSpecialization	464	0.81	0.77	0.87	0.81	0.79	0.77
CTTraits	486	1.00	1.00	1.00	1.00	1.00	1.00

eBPFTrace stands out the most, as the precision and recall are distributed more evenly across the whole range than for the other profilers.

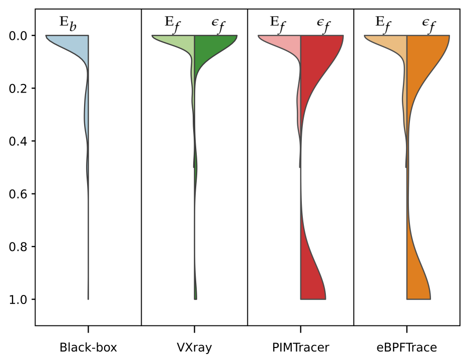
5.1.4 Accuracy (RQ$_3$)

Table 5.4: Summary of our accuracy measurements, separated by subject system. For each subject system we list the total number of patch lists across all configurations($\mathbb{P}$) and the total number of regressed features across all patch lists($\mathbb{F}$). For each profiler we report mean the accuracies with regards to the total regression measured across the whole execution (E) and the mean of accuracies of regressions measured for each individual feature(ϵ).

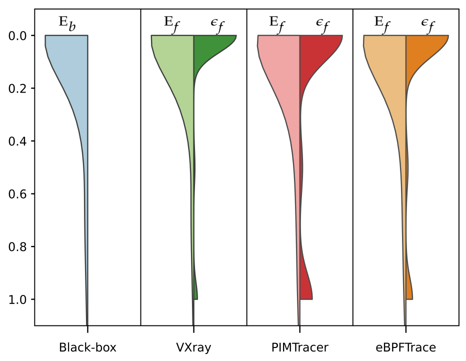
			Black-box	VXRay		PIMTracer		eBPFTrace	
	$\mathbb{P}$	$\mathbb{F}$	E_b	E_f	ϵ_f	E_f	ϵ_f	E_f	ϵ_f
Dune	183	324	1.21	0.97	1.04	0.91	1.19	0.95	0.88
HyTeG	36	48	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CTCRTP	248	656	0.06	0.02	0.04	0.05	0.37	0.04	0.35
CTPolicies	308	1376	0.17	0.16	0.09	0.20	0.24	0.19	0.15
CTSpecialization	168	464	0.07	0.07	0.12	0.07	0.12	0.07	0.12
CTTraits	192	486	0.00	0.00	0.16	0.00	0.16	0.00	0.16

Table 5.4 shows a summary of the results of the accuracy experiments. For all subject systems and profilers, we report the means of the overall feature error(E_f) and the means of all feature-specific errors(ϵ_f). In addition we show the means of the overall black-box error(E_b). The error values describe, in relative terms, how much difference we observed between the expected regression size and the measured regressions size. That is, a value of 0.00 is obtained for situations where the measured regression is exactly the size of the expected regression. The further the measured regression deviates from the expected regression, the

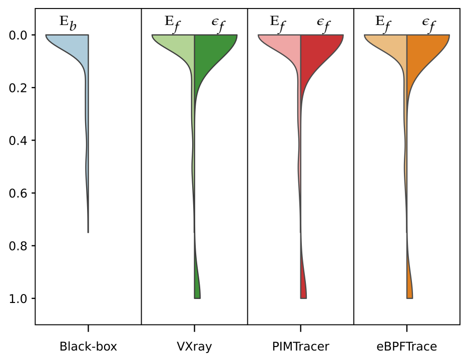
higher are the error values. For example, if we expect a regression of 1000 ms, but measure a regression of 1350 ms, the resulting error would be 0.35 (Or 35%).



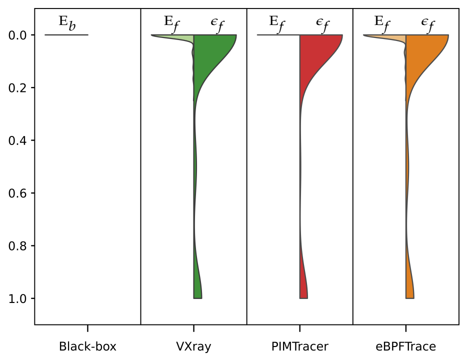
(a) *CTCRTP*



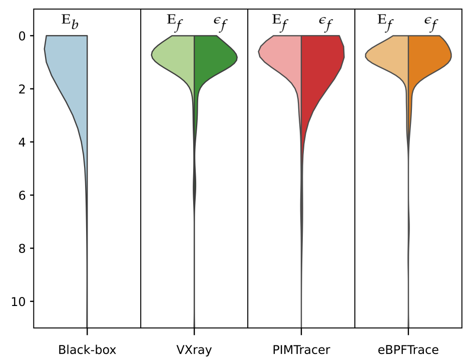
(b) *CTPolicies*



(c) *CTSpecialization*



(d) *CTTraits*



(e) DUNE

Figure 5.2: Overviews for the overall and feature specific errors separated by profiler for four of our subject systems. For each profiler, we show the likelihoods that a specific error occurs.

Generally our results show that for all our subject systems, the white-box profilers achieve comparable or even a lower overall feature error than the overall black-box errors. For our white-box profilers, we observe that for most cases the feature-specific error tends to be higher than the overall feature error. For four of our six subject systems, the overall (black-box and feature) errors we observe is below 10%, while the feature specific errors are below 40%. The *CTPolicies* subject system has slightly higher error values with an overall error of up to 20% and a feature-specific error of up to 25%. A clear outlier of our subject systems is DUNE, which exhibits notably higher error values than our other subject systems for both the overall and feature-specific errors.

Figure 5.2 shows the distribution of the errors for a selection of our subject system and profiler. We omit the plots for HyTeG as all values were all very close to 0.0, so the violin plot does not yield any relevant information. While the plots show some general differences between our subject systems, within most subject systems the distributions show a (visually) similar distribution between the different white-box profilers. In addition, the white-box profilers show a similar distribution for the overall error as the black-box profiler. For Dune not all of these observations hold. For starters, none of the white-box profilers shows a similar distribution for the overall feature error than the black-box profiler. Within the white-box profilers, we observe a distribution that generally has a higher density for lower error values, both for the overall and feature-specific error, which is consistent with the data from Table 5.4. Overall, the error values for Dune are, at least, of one order of magnitude higher than for our other subject systems.

5.2 Discussion

This section discusses the results of our experiments. We first discuss the insights of our ground truth experiment and then discuss our further experiments with specific regards to the corresponding research questions.

5.2.1 Discussion of the Feature-Specific Ground Truth Experiment

Our initial run of the feature-specific ground truth experiment revealed some important insights about the feature detection capabilities of VaRA. For once, initially none of the features related to any of the smoother components of HyTeG were detected. An investigation of the source code reveals the reason for this. In complex systems instrumenting every region, regardless of their size, can induce a large profiling overhead. Therefore, by default VaRA only instruments regions that are larger than 100 instructions. However in HyTeG, the smoother classes essentially only perform a `dynamic_cast` to the specific smoothing interface before further delegating this call. Listing 5.1 shows an original implementation of the `SORSmoother` class in HyTeG[1]. As VaRA currently does not recursively follow function calls when calculating the size of a code region, this results in an IR-code that does not reach the threshold of 100 instructions. To mitigate this we ran our further evaluations with an instruction threshold of 0 for the HyTeG subject system. For Dune, we kept the default limit of 100 instructions as in Dune we also explicitly include the grid types in our feature diagrams. We suspect that the numerous calls to functions accessing individual elements of the grids would lead to an infeasible profiling overhead.

For the Dune subject system our initial evaluation with the ground truth also revealed some short-comings in the feature detection of VaRA. For some out-of-line definitions for

1 https://i10git.cs.fau.de/hyteg/hyteg/-/blob/f4711dad/src/hyteg/solvers/SORSmoother.hpp

Listing 5.1: Implementation of the `SORSmoother` class in HyTeG. With default settings, VaRA will not instrument the `solve` method as it does not reach the instruction threshold.

```cpp
template < class OperatorType >
class SORSmoother : public Solver< OperatorType >
{
public:
   SORSmoother( const real_t& relax )
   : relax_( relax )
   , flag_( hyteg::Inner | hyteg::NeumannBoundary )
   {}

   void solve( const OperatorType&      A,
               const typename OperatorType::srcType& x,
               const typename OperatorType::dstType& b,
               const walberla::uint_t      level ) override
   {
      if ( const auto* A_sor = dynamic_cast< const SORSmoothable< typename
           OperatorType::srcType >* >( &A ) )
      {
         A_sor->smooth_sor( x, b, relax_, level, flag_ );
      }
      else
      {
         throw std::runtime_error( "The SOR-Smoother requires the SORSmoothable
             interface." );
      }
   }
private:
   real_t relax_;
   DoFType flag_;
};
```

(template-) member functions, the respective function was not annotated with the respective feature, even though the class it belongs to is explicitly annotated as such in our feature model. As a result, our ground truth reports that a regression occurs in the base functionality of a program. For some occurrences, annotating the respective out-of-line definitions in addition to the class resolves this issue. However in other cases the feature detection could still not annotate the functions properly.

In addition for Dune, we observed for a few regressions that they seem to not have any effect on the program execution. That is, in the traces of our ground truth experiments, we did not have any enter or exit event for our detection feature. This could either be a shortcoming in the feature detection of VaRA. Another option is that the location we select for our regression simply is not executed in our configuration. The fact that some of our regression patches cannot be attributed to features correctly, has different implications on our evaluations for our research questions.

Implications for RQ$_1$ As we consider all applicable patches for a configuration for RQ$_1$, patches that are not attributed correctly to specific features, but rather to the base functionality of a program have impact on the results for our white-box profilers. For the white-box profilers we only consider the time spent in feature-specific code for the regression detection. Hence, patches that affect only the base functionality of a code cannot be detected as regressions, which leads to a lower recall. Furthermore, patches that do not seem to have any effect on the program execution can impact both black-box profilers. If the potentially regressed code in fact does not get executed, this leads to a lower recall for both the black-box and white-box case.

Implications for RQ$_2$ and RQ$_3$ For RQ$_2$ and RQ$_3$, we only select patches that have an impact on feature-specific code for our patch lists. Hence, we do not expect any implications for our evaluation due to the shortcomings that our ground truth experiment revealed.

5.2.2 Discussion of RQ$_1$

When interpreting the results of our sensitivity experiment with regards to RQ$_1$ we make several observations. First of all, our results indicate that a regression detection using configuration-aware white-box profilers can outperform black-box profilers for simple subject systems. We also observe that the white-box profilers outperform the black-box approach for one of our real-world subject system. For simple subject systems, we conclude that white-box profilers detect regressions of at least 1 ms.

For real-world subject systems the results are not as clear. While we observe also observe higher detection rates for the white-box profilers in HyTeG, we do not observe the same effect in Dune. There are multiple possible reasons why we observe this behavior. First of all, for HyTeG, we consider a smaller configuration space than for Dune. It is possible that we, by accident, selected a configuration space that VaRA could analyse particularly well, so we do not experience any issues caused by the feature detection limitations of the VaRA framework. As we pointed out in Section 5.2.1, not all regressions that we introduce into Dune are properly detected by VaRA. In addition, we also use different instruction thresholds for the two subject systems. Therefore, it is possible that in Dune we lose information due to small feature regions not being measured. The mixed results of Dune and HyTeG indicate that there is a need for further analyses of complex real-world systems. One option for this is to consider more features for HyTeG to evaluate a larger configuration space. Additionally, one could include additional real-world subject systems in an extended evaluation.

Answer RQ$_1$

For synthetic subject systems, configuration-aware white-box profilers detect feature-specific regressions for a severity of at least 1 ms *and hence, outperform a black-box profiler. For real-world subject systems, white-box profilers still show high detection rates, but our experiments show mixed results which highlight the need for further investigation.*

5.2.3 Discussion of RQ_2

With regards to RQ_2, our experiment results indicate multiple things. First-off, white-box profilers are in general capable to attribute feature-specific regressions to the correct features in the majority of occurrences. However, there seem to be differences between different profilers and template-implementation techniques. While all profilers correctly attribute nearly all feature-specific regressions for one of our real-world systems, some also exhibit lower results for the Dune subject system. Similar to our discussion of RQ_1 this could be an effect of the different instruction threshold we use for Dune. Additionally, additional analysis revealed that `bpftrace` sometimes drops tracing events when profiling Dune. The reason for this is how eBPF stores and processes measurement events. To reduce the profiling overhead eBPF uses a non-blocking ring-buffer to store unprocessed measurement events. As a result, when events occur faster than the eBPF framework can process them, new events may override old ones. For the other profilers we evaluate this is not the case as in such cases the profiler will block the application until new space is available in the ring buffer.

Answer RQ_2 *Feature-aware white-box profilers can correctly attribute feature-specific performance regressions to the respective features. However, the attribution capabilities seem to be influenced by the implementation techniques. For complex real-world systems, the technical limitations of the VaRA framework or the underlying profiler can limit the feature detection capabilities.*

5.2.4 Discussion of RQ_3

Our accuracy results show two major findings. First, in terms of overall error, in which we compare the total measured regression across all features, white-box profilers have a comparable, or even slightly better accuracy than black-box profilers. With regard to feature-specific errors, our data shows that generally the feature-specific error tends to be higher than the overall error. The only exception is the HyTeG subject system which shows near-perfect accuracy data (Differences below 1%) for both overall and feature-specific errors.

One possible explanation for the generally higher error in the feature-specific case is that the profiling overhead has a higher impact on the individual measurements as individual features have a smaller absolute time to consider. For HyTeG our ground experiment (Section 5.1.1) shows that the patches that affect the "Smoother" component affect the corresponding feature 12 or 24 times. Thus, a regression with a severity of 1000 ms has an overall impact of 12 s or 24 s respectively, which can be an explanation why HyTeG does not exhibit higher values for the feature specific error, as the profiling overhead might become irrelevant at that size.

While Dune exhibits the highest overall error values, the underlying issue does not seem to be limited to the feature-aware profiling case as it also occurs for the black-box case. This indicates that the issue might in fact, be our ground truth. To further investigate this we manually inspected the data for cases in which the feature-specific error was larger than 0.4. We chose this threshold, as it is also the highest reported mean of feature specific errors among our other subject systems. The manual inspection of the expected regression obtained from our ground truth and the actual measured regression revealed two patterns:

1. Cases were the measured regression was close to 0, when the ground truth was not
2. Cases were the measured regression was much higher than the expected regression

In addition, our investigation shows that high error rates only occur for four features and do not occur for any feature interactions. Three of the affected features are our three different solver features, while the last is one of our three different preconditioners. All in all our manual inspection reveals that for the DUNE subject system our ground truth might have a systematic issue for certain features which leads to high error values. Causes for this issue might be the general issues we already discussed that the VARA framework has when detecting features in complex software systems. The reason why the profilers do not show this problem for our other real-world subject system HYTEG could lie in the smaller configuration-space that we cover for HYTEG.

Answer RQ_3 *Feature-aware white-box profilers detect feature-specific regressions with a similar accuracy than a black-box profiler. The feature-specific error of configuration-aware white-box profilers is slightly higher than the overall feature error. For real-world systems we obtain highly divided results, which highlight the need for further investigation of real-world scenarios.*

5.3 Threats to Validity

In the following, we discuss threats to internal and external validity of our experiments and evaluation.

Internal Validity: A general threats to internal validity is that the analysis in the VARA framework may not generate the correct instrumentations points that are then evaluated by the profiler. While the framework is consistently improved and evaluated there are some known occasions in which either the beginning of regions is not identified correctly, leading to regions that are being closed without a corresponding open instrumentation. However, due to the continuous improvements to the VARA framework this only occurs in rare edge cases. If such an edge case still occurs, we mitigate this by manually investigating the potential cause and eventually discarding the measurements of the affected regions.

For RQ_1, our evaluation approach has the shortcoming that we only consider regressed cases in the first place. That is, we have no measure of evaluating if our regression detection identifies false positives. In an extreme case a profiler that always reports a significant change, regardless of the underlying performance, would achieve perfect scores in our evaluation. However, for our data we did not observe this effect for any of our profilers. However to mitigate this threat for future work, one could additionally include non-regressed cases in the sensitivity analysis to calculate both recall and precision.

The main threat to internal validity of our results to RQ_2 and RQ_3 is our feature-specific ground truth. As the evaluation of RQ_3 shows, our ground truth resulted in highly inaccurate results for the Dune subject system. We account this mainly to the complex code structure of Dune which may result in less feature regions being detected correctly during the ground truth evaluation. For future work, we recommend using another method to build a feature-specific ground truth, that does not solely rely on the feature detection of VaRA to identify where the regression has an impact. One example for an alternative ground truth could be to build a combined trace with both function call information from LLVM XRay and feature information from VaRA. With this combined trace one could identify in which feature regions our synthetic regression function is called. Due to time constraints we could unfortunately not create this alternative ground truth in this thesis and re-evaluate our results.

External Threats: When evaluating our results, we have to consider multiple threats regarding external validity. First, all the projects under inspection are written in C++. This may limit the generalizability of our results to other languages. However, the focus of this work are systems that are used in HPC environments. Due to the performance sensitive nature of HPC systems, they tend do be written in C or C++ to maximise the performance. In addition only few languages offer a mechanism as powerful as the C++ templates for compile-time variability.

Finally, our investigation only includes a few real-world systems, which can limit the generalizability of our findings. One of the main reasons for this is that we need a sufficient understanding of the available features and restrictions of the systems. While it would be possible to manually extract this information, it would be a tedious and error prone process. Therefore, we build on the assistance of domain experts of these systems. Due to this, we limit the scope of real world projects to Dune and HyTeG where we have a point of contact from the developer team.

Lastly, our evaluation only considers simple synthetic regressions that where, effectively, a single line of code in one file introduces a regression. We chose this approach, such that regressions always span across a single feature region. Real-world regressions however, are likely to be more complex and also cross-cutting between different source files, which could mean that different feature regions (belonging to the same or different features), participate in it. Therefore a more thorough investigation including real-world regressions is necessary to further evaluate the limitations of our approach. While it was initially planned to also include real-world regressions for Dune and HyTeG, it was not possible to integrate these into our evaluation pipeline due to time constraints. However we plan to address this in our future work.

6

Related Work

In the past years, several works have been published that aim to tackle configuration-aware profiling. We differentiate between four categories of related work: variability-aware performance analysis, compile-time configurable systems, C++ templates in practice and finally performance analysis of HPC frameworks.

Variability-Aware Performance Analysis In recent years, several works have been published that aim to understand the relationship between features, feature interactions and the performance of a configurable software system. Specifically, multiple publications support the claim that for certain scenarios, white-box analysis can yield better results in that regard. Kolesnikov et al. have shown that a relationship between feature interactions and performance exists and that it is possible to predict these feature interactions by performing source-code analysis [38]. In a larger scale empirical study, Rhein et al. discovered that variability-aware static white-box analysis can outperform sample based black-box techniques in regards of effectiveness and efficiency [54]. Muehlbauer et al. present work that explores performance beyond the configurability dimension: They propose an approach that was not only able to detect performance changes across different variants, but at the same time also across different revisions of these variants. Using this approach, they were able to accurately identify performance changes and pin-point them to specific feature interactions across different configurations [49].

Weber et al. present a white-box performance-analysis-based approach that can track the configuration dependent performance on a method level. Their framework uses a two-tiered approach, were a coarse-grained profiler is used in a first step to identify methods that are potentially configuration- and performance-sensitive. In a second step, these methods are then analyzed by a more fine-grained profiler which in effect allows to build a more accurate performance model of the system. They evaluate their approach on several real-world Java software systems [68]. An even more fine-grained approach is presented by Velez et al. with ConfigCrusher[65], which proposes a white-box analysis tool for Java systems that can analyse the control- and data-flow of a program to infer the influence of individual configuration options on the overall systems' performance.

Compile-Time Configurable Systems There are different works that discuss how to implement compile-time configurable systems. Czarnecki and Eisenecker present an extensive overview of different implementation techniques with a focus on the C++ language and specifically the template mechanisms [15]. Similarly, Filman et al. give an overview on the implementation of variability using aspects [19]

In the domain of analyses of compile-time configurable systems, numerous works have investigated the C++ pre-processor and the usage of `#ifdef` directives to implement variability. This work includes general investigations of the challenges around this type of variability [17, 43, 46, 61], improving tooling support [18, 32, 40], and lifting compile-time variability into other forms of variability [28, 55, 58].

C++ Templates in Practice C++ templates are a challenging implementation technique which is, in part, represented in research. Already in 2006 Porkoláb et al. present Templight, a debugging framework for C++ templates[52]. To make the usage and invocations of C++ templates better understandable, Gscheind et al. present TUAnalyzer [23], which extract the template structure of C++ programs based on the internal representation of the GCC compiler. To further elaborate the challenges of C++ templates in practice, Kelling et al. present an approach to port C++ template based designs to other implementation techniques[34].

To understand how C++ templates are used in practice, Chen et al. present an empirical study that analyses the usage of templates in 50 open source software systems[14]. Other practical applications of C++ templates include mathematical frameworks in analytics and algebra. While various implementations of such frameworks exist[36, 57, 67], Brandt et al. present a more theoretical and generalized overview of how to employ templates in the design of an computational algebra library[9].

Performance Analysis of HPC Frameworks Extensive work has been done on analysing HPC frameworks. Various work has been presented that investigates the benefits and challenges of specific frameworks. One example for this is the investigation of CODA by Wagner et al. While they focussed on analysing the challenges of CODA, a closed-source computational fluid dynamics solver used for aerospace engineering, some of their findings may be generally applicable to other frameworks. Notably Wagner et al., they also mentioned that the heavy use of C++ templates complicates performance analyses as current tooling is not well equipped for analysing the generated code [67].

Madsen et al. present a general-purpose analysis framework for HPC application: Timemory presents a modular framework to easily embed profiling capabilities into the codebase of a project. While this approach gives project maintainers many flexibilities of embedding different performance analyses into their work, it also exhibits the downside that projects need to be specifically tailored to use the Timemory framework in order to analyse them. Therefore, Timemory cannot be used on arbitrary applications without modifying their source code [47].

A more generalized approach to performance analysis, specifically for modern C++ systems, is presented by Zhou et al.. In their work, they present a approach to identify performance bottlenecks and parallel scaling issues with two different performance analysis tools[69].

7 Conclusion and Outlook

In summary, our results indicate that configuration-aware white-box profiling approaches can detect and measure performance regressions with at least the same granularity as a black-box profiler. On a more important note, our results show that configuration-aware profilers correctly attribute performance regressions to the features that cause them which provides a major benefit over black-box profilers. However our results also show that these results are not yet applicable to all software systems and special consideration should be taken when analysing complex real-world subject systems. Lastly, we also reveal a shortcoming in our approach that can aid others who conduct similar research to design their experiments.

Our results show that with regards to *sensitivity*, white-box profilers are at least as sensitive to performance regressions as a black-box profilers for five out of our six subject systems. Only for Dune, one of our real-world case studies, a black-box profiler outperforms the white-box profilers slightly. This shows that while white-box profilers can generally out-perform a black-box approach, special care might be required when working with complex real-world systems.

For *precision* and *recall* our research shows that white-box profilers can generally attribute most feature-specific regressions to the correct features, while at the same time identifying little false-positives. As black-box profilers are lacking the information to attribute regressions to specific features, this gives configuration-aware white-box profilers a major benefit for regression detection and performance debugging. Our results indicate that a feature-aware regression detection is more precise for certain implementation patterns. For HyTeG, one of our real-world subject systems, our approach worked exceptionally well such that all introduced regressions were attributed to the correct features while nearly no false-positives are reported. For Dune however, the different profiling approaches could, at maximum, identify and attribute half of the regressions to the correct features. However with regards to false-positives, we achieve similar results to our synthetic subject systems. Overall we observe that eBPF shows the lowest values for precision and recall which, upon further investigation, we could attribute to a implementation choice of the eBPF framework.

Lastly in our evaluation of the accuracy of different configuration-aware white-box profilers, we show that white-box profilers achieve comparable errors than a black-box profiler overall. Our results show that the average overall error reported across five of our six subject systems is at most 20%. For the feature-specific accuracy of configuration-aware white-box profilers, all our profilers report a mean error of at most 37% for five out of six subject systems. For our sixth subject system Dune however, our accuracy results report error rates of up to 122% on average. Our investigation shows that this error is most likely caused by an error in our ground truth experiment that we use to determine the expected regression size.

For the domain of HPC applications, our results indicate that feature-aware white-box profilers are applicable for performance investigation on complex real-world systems. However, the implementation techniques used in a real-world software system might affect the interpretation of the results. Our results also indicate that configuration-aware white-box profilers can detect feature-specific regressions at lower severities, which might be interesting for application that run in parallel on large compute clusters, as for these detecting smaller regressions can be important. However, our results also show that some applications are harder to instrument than others due to e. g., their complexity. Therefore performance engineers should be aware that there might be limitations to the generalizability of our results to all kind of real-world systems. We recommend to people who want to perform configuration-aware white-box analyses on their systems, to first conduct a small controlled experiment to unravel potential system-specific short-comings.

For the planned future work we want to address two main aspects: Strengthening the validity of our results by fine-tuning and improving our methodology and evaluating other applications of configuration-aware performance analyses.

Strengthening Validity To improve the validity of our results we want to address multiple aspects that were not possible to asses in the scope of this thesis. First off, we plan to address the short-comings of our feature-specific ground truth by investigating other means of obtaining an accurate feature-specific ground truths. With that we can re-evaluate the results of both RQ_2 and RQ_3 to verify that our observations still apply. Furthermore, to get a better understanding of the applicability of real-world subject systems we plan to run additional experiments both with an extended configuration-space for HyTeG and also for new additional subject systems from the HPC domain. To evaluate the applicability of our approach on non-synthetic regressions, we also plan to include real-world regressions in an additional evaluation. For Dune we already identified a suitable feature-specific regression to evaluate. We plan to also discuss possible real-world regressions with the development team of HyTeG to include in this evaluation.

Other Applications of Configuration-Aware Performance Analyses Being able to associate performance information with specific features, enables a more fine-grained analysis of software performance. As our results for RQ_2 and RQ_3 show, the attribution and measurements of feature-specific performance can be both precise and accurate. With further fine-tuning this opens lots of possibilities to enhance existing profiling approaches or open even new analysis domains. At the moment, our approach only considers a software at a single state of development. However, as software evolves over time, we plan to extend our approach to enable more performance analyses over a period of time, enabling an analysis of both the (configuration) space dimension and the *time* dimension. The fine-grained performance information for each version of a program could, for example, allow an investigation of feature-specific performance change points. Current work ([49]) relies on learning PIMs from black-box measurements. With our white-box approach we have a more direct way of identifying the influences of individual features. We want explore whether we can use this property to enable faster and easier change-point detection.

Apart from performance change points, we also want to explore other parts of the feature-specific performance changes through the evolution of a software system. As a configuration-aware white-box performance profiler can easily identify the influences of individual features on the performance, we plan to explore this evolution of the software performance over time. We want to investigate whether there are features that participate more or less in the overall performance over time, and investigate the reasons for that. We also believe that this enables multiple synergies with research from the socio-technical software analytics domain[21, 29].

We furthermore want to explore the capabilities of our configuration-aware white-box profiling approach to detect *workload* specific performance changes. Similar to performance change points, current works ([42, 50]) use black-box approaches to classify the overall influence of workloads on performance. Using our approach one could perform a more fine-grained, feature-specific, evaluation of the impact of workloads in configurable software systems.

A Appendix

The following figures show the different distributions of precision and recall for RQ$_2$ for the individual subject systems. Please note, that we do not include the distribution for *CTTraits*, as it achieves perfect precision and recall, thus a distribution plot would just show a single line at 1.0.

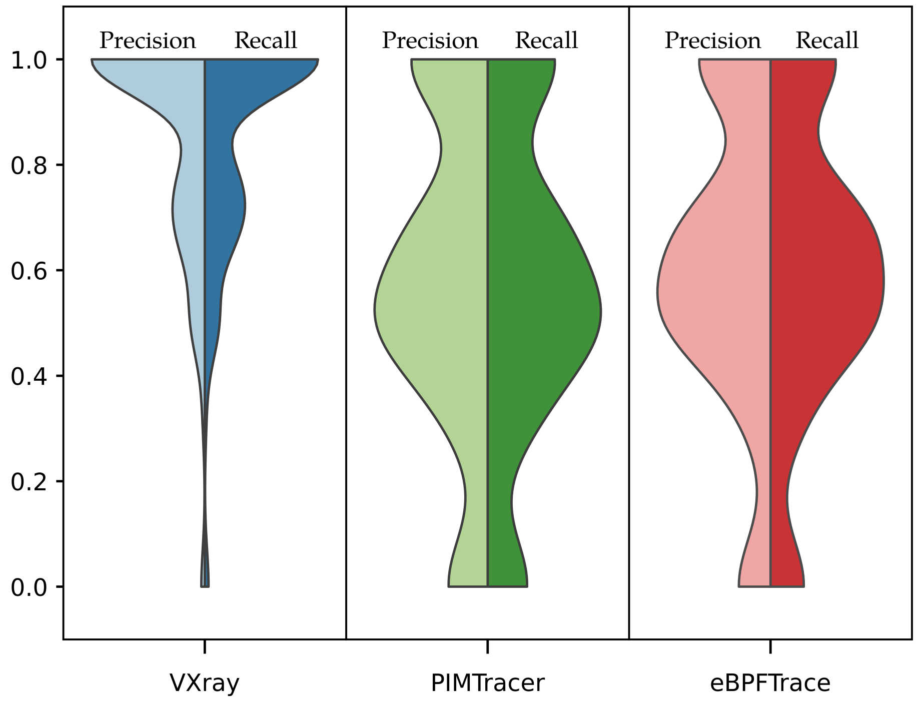


Figure A.1: Distribution for sensitivity and recall for the *CTCRTP* subject system.

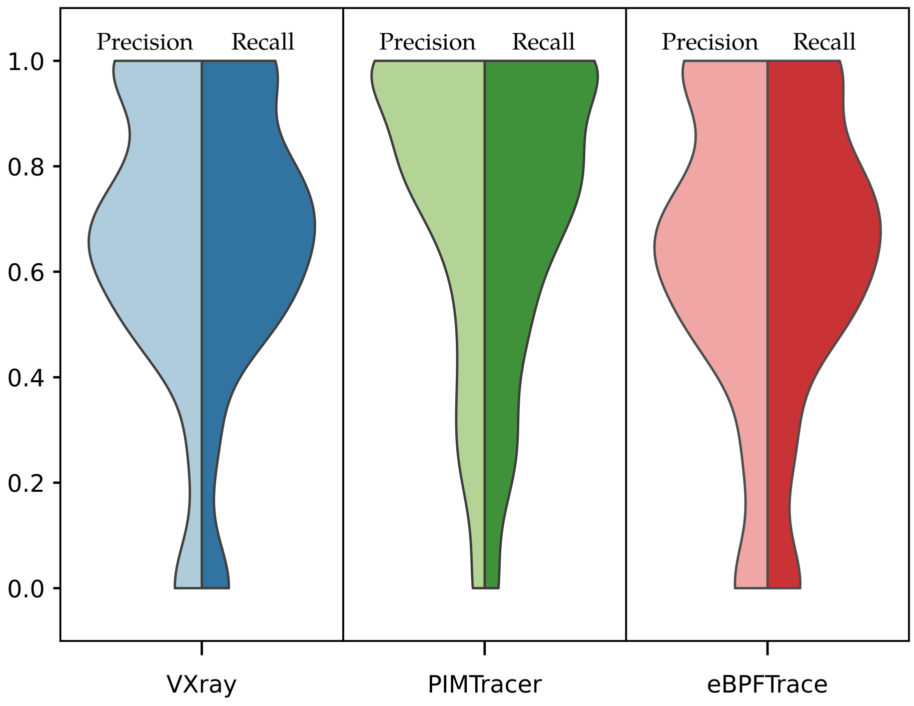


Figure A.2: Distribution for precsision and recall for the *CTPolicies* subject system.

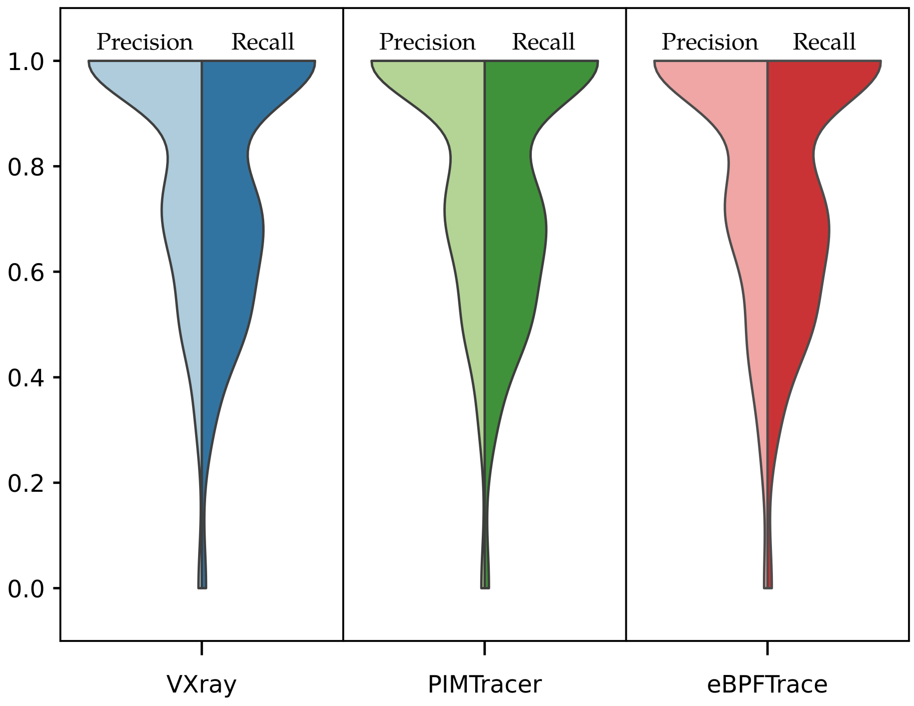


Figure A.3: Distribution for precision and recall for the *CTSpecialization* subject system.

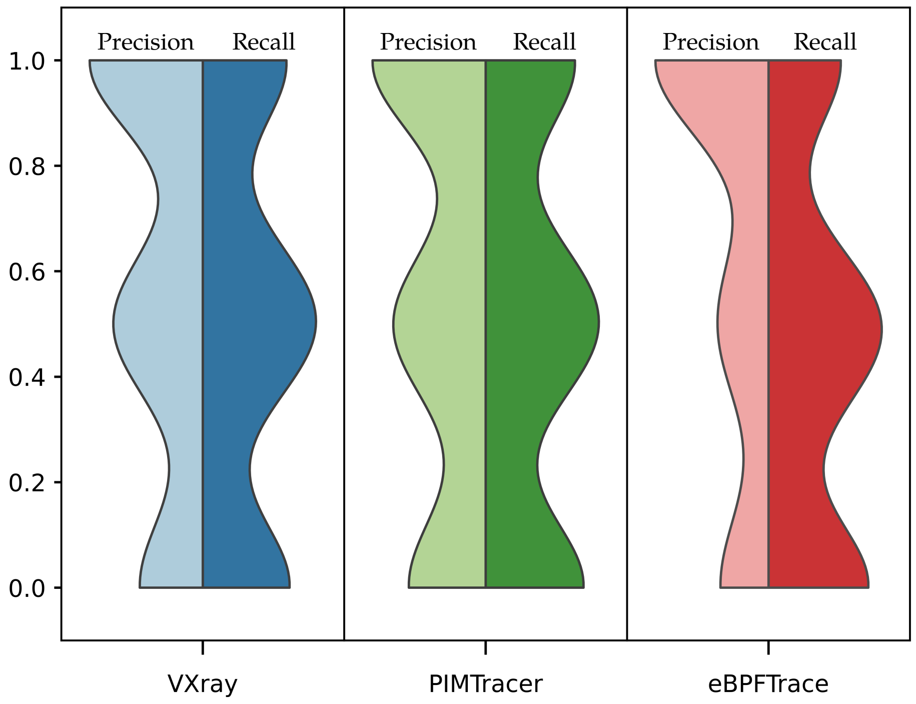


Figure A.4: Distribution for precision and recall for the Dune subject system.

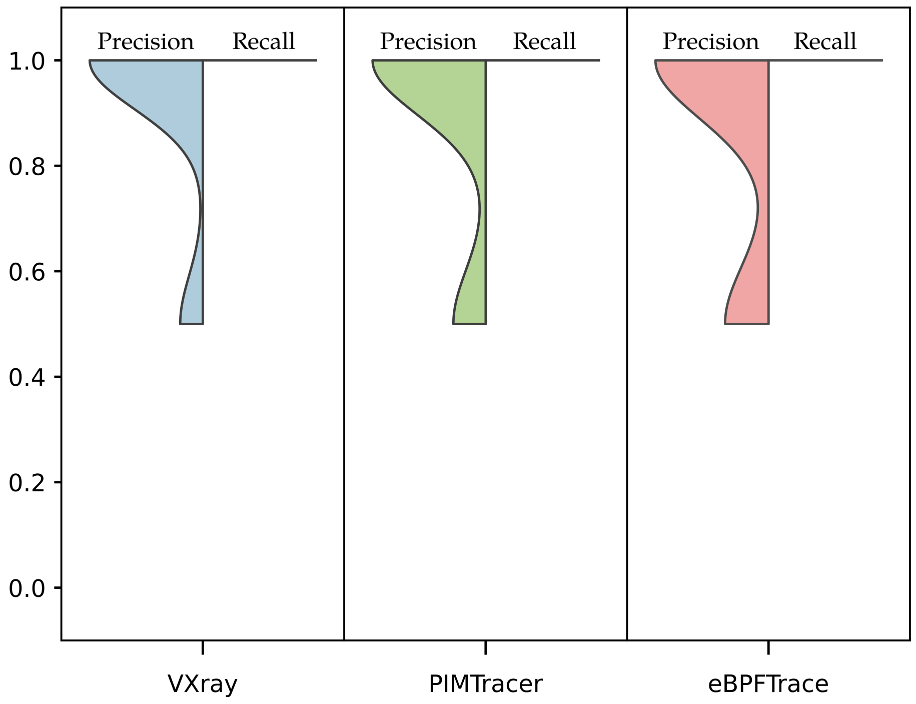


Figure A.5: Distribution for precision for the HyTeG subject system. The recall distribution is not visible, as HyTeG achieved perfect recall of 1.0 in all cases.

Bibliography

[1] Iago Abal, Jean Melo, Stefan Stanciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wasowski. "Variability Bugs in Highly Configurable Systems: A Qualitative Analysis." In: *ACM Trans. Softw. Eng. Methodol.* 26.3 (2018), 10:1–10:34.

[2] A. Ahmad, A. Noor, H. Sharif, U. Hameed, S. Asif, M. Anwar, A. Gehani, F. Zaffar, and J. Siddiqui. "Trimmer: An Automated System for Configuration-Based Software Debloating." In: *IEEE Trans. Softw. Eng.* 48.09 (2022), pp. 3485–3505.

[3] Jonathan Aldrich, David Garlan, Christian Kaestner, Claire Le Goues, Anahita Mohseni-Kabir, Ivan Ruchkin, Selva Samuel, Bradley Schmerl, Christopher Steven Timperley, Manuela Veloso, Ian Voysey, Joydeep Biswas, Arjun Guha, Jarrett Holtz, Javier Camara, and Pooyan Jamshidi. "Model-Based Adaptation for Robotics Software." In: *IEEE* 36.2 (2019), pp. 83–90.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[5] Per Alexius, B. Maryam Elahi, Fredrik Hedman, Phillip Mucci, Gilbert Netzer, and Zeeshan Ali Shah. "A Black-Box Approach to Performance Analysis of Grid Middleware." In: *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007, Rennes, France, August 28-31, 2007, Revised Selected Papers*. Springer, 2007, pp. 62–71.

[6] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.

[7] Peter Bastian, Markus Blatt, Andreas Dedner, Nils-Arne Dreier, Christian Engwer, René Fritze, Carsten Gräser, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. "The DUNE Framework: Basic concepts and recent developments." In: *Comput. Math. Appl.* 81 (2021), pp. 75–112.

[8] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. *XRay: A Function Call Tracing System*. Tech. rep. 2016.

[9] Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. "Employing C++ Templates in the Design of a Computer Algebra Library." In: *Int. Conf. Mathematical Software (ICMS)*. Springer, 2020, pp. 342–352.

[10] Rolando Brondolin and Marco D. Santambrogio. "A Black-box Monitoring Approach to Measure Microservices Runtime Performance." In: *ACM Trans. Archit. Code Optim.* 17.4 (2020), 34:1–34:26.

[11] Michael D. Brown and Santosh Pande. "Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating." In: *Workshop on Cyber Security Experimentation and Testing (USENIX CSET)*. USENIX Association, 2019.

[12] Fatih Calakli and Gabriel Taubin. "SSD: Smooth Signed Distance Surface Reconstruction." In: *Comput. Graph. Forum* 30.7 (2011), pp. 1993–2002.

[13] Jeffrey C. Carver and Alexander Serebrenik. "Software Maintenance and Evolution and Automated Software Engineering." In: *IEEE Softw.* 35.2 (2018), pp. 102–104.

[14] Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. "How C++ Templates Are Used for Generic Programming: An Empirical Study on 50 Open Source Systems." In: *ACM Trans. Softw. Eng. Methodol.* 29.1 (2020), 3:1–3:49.

[15] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - Methods, Tools and Applications*. Addison-Wesley, 2000.

[16] Chris Edwards. "Moore's Law: what comes next?" In: *Commun. ACM* 64.2 (2021), pp. 12–14.

[17] Jean-Marie Favre. "Understanding-In-The-Large." In: *Int. Workshop on Program Comprehension (WPC)*. IEEE, 1997, pp. 29–38.

[18] Janet Feigenspan, Michael Schulze, Maria Papendieck, Christian Kästner, Raimund Dachselt, Veit Köppen, and Mathias Frisch. "Using Background Colors to Support Program Comprehension in Software Product Lines." In: *Proc. Int. Conf. Evaluation & Assessment in Software Engineering(EASE)*. Institute of Engineering and Technology, 2011, pp. 66–75.

[19] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[20] Johann Wolfgang von Goethe. *Faust. Eine Tragödie.* 1806.

[21] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. "Analysing Time-Stamped Co-Editing Networks in Software Development Teams using git2net." In: *Empir. Softw. Eng.* 26.4 (2021), p. 75.

[22] Brendan Gregg. *Systems Performance*. Pearson, 2020.

[23] Thomas Gschwind, Martin Pinzger, and Harald C. Gall. "TUAnalyzer – Analyzing Templates in C++ Code." In: *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE Softw., 2004, pp. 48–57.

[24] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. "Variability-Aware Performance Prediction: A Statistical Learning Approach." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2013, pp. 301–311.

[25] Huong Ha and Hongyu Zhang. "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Networks." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2019, pp. 1095–1106.

[26] Xue Han and Tingting Yu. "An Empirical Study on Performance Bugs for Highly Configurable Software Systems." In: *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2016, pp. 1–10.

[27] Jens Happe, Hui Li, and Wolfgang Theilmann. "Black-box Performance Models: Prediction Based on Observation." In: *Proc. Int. Workshop on Quality of Service-Oriented Software Systems (QUASOSS)*. ACM, 2009, pp. 19–24.

[28] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. "Effective Analysis of C Programs by Rewriting Variability." In: *Art Sci. Eng. Program.* 1.1 (2017), pp. 1–25.

[29] Mitchell Joblin, Barbara Eckl, Thomas Bock, Angelika Schmid, Janet Siegmund, and Sven Apel. "Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study." In: *ACM Trans. Softw. Eng. Methodol.* 32.4 (2023).

[30] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models." In: *Proc. Int. Software Product Line Conf. (SPLC)*. ACM, 2012, pp. 46–55.

[31] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. "Distance-Based Sampling of Software Configuration Spaces." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2019, pp. 1084–1094.

[32] Christian Kästner, Sven Apel, and Martin Kuhlemann. "Granularity in Software Product Lines." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2008, pp. 311–320.

[33] Michael M. Kazhdan, Matthew Bolitho, and Hugues Hoppe. "Poisson surface reconstruction." In: *Proc. Eurographics Symp. Geometric Processing*. Ed. by Alla Sheffer and Konrad Polthier. Eurographics Association, 2006, pp. 61–70.

[34] Jeffrey Kelling, Sergei Bastrakov, Alexander Debus, Thomas Kluge, Matt Leinhauser, Richard Pausch, Klaus Steiniger, Jan Stephan, René Widera, Jeff Young, Michael Bussmann, Sunita Chandrasekaran, and Guido Juckeland. "Challenges Porting a C++ Template-Metaprogramming Abstraction Layer to Directive-Based Offloading." In: *Accelerator Programming Using Directives*. Springer International Publishing, 2022, pp. 92–111.

[35] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. "TypeChef: toward type checking #ifdef variability in C." In: *Proc. Int. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2010, pp. 25–32.

[36] Nils Kohl, Dominik Thönnes, Daniel Drzisga, Dominik Bartuschat, and Ulrich Rüde. "The HyTeG finite-element software framework for scalable multigrid solvers." In: *Int. J. of Parallel, Emergent and Distributed Systems* 34.5 (2019), pp. 477–496.

[37] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. "Tradeoffs in Modeling Performance of Highly Configurable Software Systems." In: *Softw. Syst. Model.* 18.3 (2019), pp. 2265–2283.

[38] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. "On the Relation of Control-Flow and Performance Feature Interactions: A Case Study." In: *Empir. Softw. Eng.* 24.4 (2019), pp. 2410–2437.

[39] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking*. Springer-Verlag, 2020. Chap. Measurement Techniques, pp. 131–148.

[40] Duc Le, Eric Walkingshaw, and Martin Erwig. "#ifdef confirmed harmful: Promoting understandable Software Variation." In: *Proc. Int. Symp. Visual Languages and Human-Centric Computing (VLHCC)*. IEEE, 2011, pp. 143–150.

[41] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. "There's plenty of room at the Top: What will drive computer performance after Moore's law?" In: 368 (2020).

[42] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. "Using Black-box Performance Models to Detect Performance Regressions Under Varying Workloads: An Empirical Study." In: *Empir. Softw. Eng.* 25.5 (2020), pp. 4130–4160.

[43] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. "An Analysis of the Variability in Forty Preprocessor-based Software Product Lines." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[44] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[45] Max Lillack, Christian Kästner, and Eric Bodden. "Tracking Load-Time Configuration Options." In: *IEEE Trans. Softw. Eng.* 44.12 (2018), pp. 1269–1291.

[46] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. "A quantitative analysis of aspects in the eCos kernel." In: *Proc. Europ. Conf. Computer Systems (EuroSys)*. ACM, 2006, pp. 191–204.

[47] Jonathan R. Madsen, Muaaz G. Awan, Hugo Brunie, Jack Deslippe, Rahulkumar Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. "Timemory: Modular Performance Analysis for HPC." In: *Proc. Int. Supercomputing Conf. High Performance Computing*. Springer-Verlag, 2020, pp. 434–452.

[48] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. "A Comparison of 10 Sampling Algorithms for Configurable Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2016, pp. 643–654.

[49] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Identifying Software Performance Changes across Variants and Versions." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2021, pp. 611–622.

[50] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. "Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2023, pp. 2085–2097.

[51] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. "Finding Near-Optimal Configurations in Product Lines by Random Sampling." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 61–71.

[52] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos. "Debugging C++ Template Metaprograms." In: *Proc. Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2006, pp. 255–264.

[53] Václav Rajlich. "Software Evolution and Maintenance." In: *Proc. Int. Conf. Future of Software Engineering*. ACM, 2014, pp. 133–144.

[54] Alexander Von Rhein, JöRG Liebig, Andreas Janker, Christian Kästner, and Sven Apel. "Variability-Aware Static Analysis at Scale: An Empirical Study." In: *ACM Trans. Softw. Eng. Methodol.* 27.4 (2018), pp. 1–33.

[55] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. "Variability Encoding: From Compile-time to Load-time Variability." In: *J. Log. Algebraic Methods Program.* 85.1 (2016), pp. 125–145.

[56] Liz Rice. *What Is eBPF?* O'Reilly Media, 2022.

[57] Oliver Sander. *DUNE - The Distributed and Unified Numerics Environment*. Springer, 2020.

[58] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. "Static Data-Flow Analysis for Software Product Lines in C." In: *Autom. Softw. Eng.* 29.1 (2022), p. 35.

[59] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems." In: *Proc. Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. ACM, 2015, pp. 284–294.

[60] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. "Scalable Prediction of Non-Functional Properties in Software Product Lines: Footprint and Memory Consumption." In: *Inf. Softw. Technol.* 55.3 (2013), pp. 491–507.

[61] Henry Spencer and Geoff Collyer. "#ifdef Considered Harmful, or Portability Experience with C News." In: *Proc. USENIX Technical Conf.* USENIX Association, 1992, pp. 185–197.

[62] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10, 000 Feature Problem." In: *Proc. Europ. Conf. Computer Systems (EuroSys)*. ACM, 2011, pp. 47–60.

[63] Xhevahire Tërnava, Mathieu Acher, Luc Lesoil, Arnaud Blouin, and Jean-Marc Jézéquel. "Scratching the Surface of ./configure: Learning the Effects of Compile-Time Options on Binary Size and Gadgets." In: Springer, 2022, pp. 41–58.

[64] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates: The Complete Guide (2nd Edition)*. Addison-Wesley, 2017.

[65] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems." In: *Proc. Int. Conf. Automated Software Engineering (ASE)* 27.3 (2020), pp. 265–300.

[66] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2021, pp. 1072–1084.

[67] Michael Wagner, Jens Jägersküpper, Daniel Molka, and Thomas Gerhold. "Performance Analysis of Complex Engineering Frameworks." In: Springer-Verlag, 2021.

[68] Max Weber, Sven Apel, and Norbert Siegmund. “White-Box Performance-Influence Models: A Profiling and Learning Approach.” In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2021, pp. 1059–1071.

[69] Huan Zhou, Christoph Niethammer, and Martin Herrerias Azcue. “Usage Experiences of Performance Tools for Modern C++ Code Analysis and Optimization.” In: Springer, 2021.