Automatic Performance Modeling and Analysis of Configurable Scientific Software and Workflows

Larissa Schmid

Larissa Schmid

Automatic Performance Modeling and Analysis of Configurable Scientific Software and Workflows

**The Karlsruhe Series on Software Design and Quality
Volume 42**

Dependability of Software-intensive Systems group
Faculty of Computer Science
Karlsruhe Institute of Technology

and

Software Engineering Division
Research Center for Information Technology (FZI), Karlsruhe

Editor: Prof. Dr. Ralf Reussner

Automatic Performance Modeling and Analysis of Configurable Scientific Software and Workflows

by
Larissa Schmid

Karlsruher Institut für Technologie
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

Automatic Performance Modeling and Analysis of Configurable
Scientific Software and Workflows

Zur Erlangung des akademischen Grades einer Doktorin der Ingeni-
eurwissenschaften von der KIT-Fakultät für Informatik des Karlsruher
Instituts für Technologie (KIT) genehmigte Dissertation

von Larissa Schmid

Tag der mündlichen Prüfung: 5. Dezember 2024
1. Referentin: Prof. Dr.-Ing. Anne Koziolek
2. Referent: Prof. Dr. ir. Alexandru Iosup

Abstract

Modern software systems are configurable and allow users to set many parameters according to their needs. Usually, the same functionality can be achieved with varying performance due to many non-functional parameters. However, it is unclear how the parameters affect performance, for example, when trying to understand application scalability. Performance models express application performance as functions of input parameters, helping users and developers understand application behavior. While performance models can be created manually, the cost associated with this is high due to the need for performance experts. Therefore, developers often select an arbitrary subset of the configuration space for evaluation.

Automatic performance modeling generates models from empirical measurements that ideally cover all configuration options. However, the number of options makes exhaustive measurements of all configurations infeasible. For applications with many parameters, modeling all options results in a trade-off between model quality and number of measurements, leading to modeling only a subset of options. Current modeling approaches do not consider filtering options that do not impact performance, and do not use known interactions between options for designing measurements, resulting in expensive modeling processes.

To manage increasingly complex and data-intensive simulations, many scientific communities design their applications as workflows with many jobs linked by data dependencies between them. Additionally to finding optimal configurations of the single applications, the challenge of orchestrating them arises. The scientific community is increasingly interested in using serverless solutions. However, serverless platforms and orchestration systems offer not only different APIs and capabilities, but also have fundamentally different programming models, diverging in the statelessness of functions and the static nature of graph definition. Publications use different applications to benchmark the performance of new ideas, do not cover the same classes of

workloads, and do not always compare against the same subset of platforms. Consequently, it is unclear to developers which platform will be best suited for their workloads. Developers need to conduct extensive and reliable benchmarking to estimate the performance of their workloads and understand platform limitations.

Therefore, this thesis addresses how the cost of automatic performance modeling methods can be decreased and how the performance of workflow orchestrations can be analyzed comparatively. As first contribution, we introduce a novel white-box measurement methodology that uses parametric profiles to understand the impact of parameters on the performance of program functions. By analyzing them, we can derive conclusions on parameter interactions in the program's control flow. Applying the deduced conclusions to the experiment design removes measuring points unnecessary to model parameter interactions. By evaluating with two applications, we show that we can significantly lower experimentation costs by a factor of up to 34 compared against two state-of-the-art modeling workflows, while maintaining the accuracy of the resulting models.

Our second contribution is an approach to automatically determine performance-irrelevant configuration options, allowing users to select options to model from a reduced set of only performance-relevant configuration options without losing predictive power in the resulting performance model. In the evaluation with the multi-physics solver Pace3D, we show that the approach can accurately classify options as performance-irrelevant and can save measurement costs for creating performance models despite the additional steps introduced.

Our third contribution is a serverless workflows benchmarking suite to support developers in choosing the right platform for their workflow and to support the quickly growing research activity in serverless workflows by providing a benchmarking methodology and a baseline to compare against. We propose a model for serverless workflows based on the Petri Net formalism, providing a platform-agnostic workflow definition to model control- and data-flow. Modeled applications can automatically be transcribed into a cloud's proprietary presentations, enabling developers to run near identical workloads on different systems. We evaluate the expressiveness and overhead of our model by reviewing the literature on serverless workflows. We provide a ready-to-use benchmarking suite with application and micro-benchmarks

and extensively analyze performance, cost, scaling, stability, and how well serverless platforms support scientific workflows by deploying our benchmarks to three different platforms.

This thesis contributes to providing methodology for domain scientists and software developers to facilitate better understanding of application performance. By reducing the costs and complexity associated with performance modeling and offering a comprehensive benchmarking framework, the contributions of this work enable domain scientists to better understand and optimize performance in their applications, fostering more efficient use of resources and accelerating scientific innovation.

Zusammenfassung

Moderne Softwaresysteme sind konfigurierbar und ermöglichen es den Nutzer:innen, viele Parameter nach ihren Bedürfnissen einzustellen. Aufgrund vieler nicht-funktionalen Parameter kann in der Regel dieselbe Funktionalität mit unterschiedlicher Performance erreicht werden. Es ist jedoch unklar, wie sich die Parameter auf die Performance auswirken, zum Beispiel beim Versuch, die Skalierbarkeit der Anwendung zu verstehen. Performance-Modelle drücken die Performance der Anwendung als Funktionen von Eingabeparametern aus und helfen Benutzer:innen und Entwickler:innen, das Anwendungsverhalten zu verstehen. Performance-Modelle können zwar manuell erstellt werden, aber die damit verbundenen Kosten sind hoch, da für deren Erstellung Performance-Expert:innen erforderlich sind. Daher wählen Entwickler:innen oft eine beliebige Teilmenge des Konfigurationsraums für die Bewertung aus.

Methoden zur automatischen Performance-Modellierung erzeugen Modelle aus empirischen Messungen, die idealerweise alle Konfigurationsoptionen abdecken. Aufgrund der Anzahl der Optionen sind jedoch erschöpfende Messungen aller Konfigurationen nicht durchführbar. Bei Anwendungen mit vielen Parametern führt die Modellierung aller Optionen zu einem Kompromiss zwischen Modellqualität und Anzahl der Messungen, so dass nur eine Teilmenge der Optionen modelliert wird. Aktuelle Modellierungsansätze ziehen nicht in Betracht, Optionen von der Modellierung auszuschließen, die sich nicht auf die Performance auswirken, und nutzen keine bekannten Interaktionen zwischen Optionen für das Design von Messungen, was zu teuren Modellierungsprozessen führt.

Um immer komplexere und datenintensivere Simulationen zu verwalten, entwickeln viele wissenschaftliche Communities ihre Anwendungen als Workflows mit vielen einzelnen Jobs, die durch Datenabhängigkeiten miteinander verbunden sind. Neben der Suche nach optimalen Konfigurationen für die einzelnen Anwendungen ergibt sich auch die Herausforderung, diese zu

orchestrieren. Die wissenschaftliche Community ist zunehmend an der Verwendung von serverlosen Lösungen interessiert. Serverlose Plattformen und Orchestrierungssysteme bieten jedoch nicht nur unterschiedliche APIs und Fähigkeiten, sondern haben auch grundlegend unterschiedliche Programmiermodelle, die sich in der Zustandslosigkeit von Funktionen und der statischen Natur der Graphendefinition unterscheiden. Veröffentlichungen verwenden unterschiedliche Anwendungen, um die Performance neuer Ideen zu bewerten, decken nicht die gleichen Klassen von Workloads ab und vergleichen nicht immer mit der gleichen Teilmenge von Plattformen. Folglich ist es für die Entwickler:innen unklar, welche Plattform für ihre Workloads am besten geeignet ist. Entwickler:innen müssen umfangreiche und zuverlässige Benchmarks durchführen, um die Performance ihrer Anwendungen abzuschätzen und die Grenzen der Plattformen zu verstehen.

Daher befasst sich diese Arbeit mit den Fragen, wie die Kosten für Methoden der automatischen Performance-Modellierung gesenkt werden können und wie die Leistung von Workflow-Orchestrationen vergleichend analysiert werden kann. Als ersten Beitrag stellen wir eine neuartige White-Box-Messmethodik vor, die parametrische Profile verwendet, um die Auswirkungen von Parametern auf die Performance von Programmfunktionen zu verstehen. Durch die Analyse dieser Profile können wir Rückschlüsse auf die Interaktionen der Parameter im Kontrollfluss des Programms ziehen. Durch die Anwendung der abgeleiteten Schlussfolgerungen auf das Design von Messungen werden Messpunkte entfernt, die für die Modellierung von Parameterinteraktionen unnötig sind. Anhand von zwei Anwendungen zeigen wir, dass wir die Messkosten im Vergleich zu zwei modernen Modellierungsworkflows um einen Faktor von bis zu 34 senken können, während die Genauigkeit der resultierenden Performance-Modelle erhalten bleibt.

Unser zweiter Beitrag ist ein Ansatz zur automatischen Bestimmung von für die Performance relevanten Konfigurationsoptionen, der es Nutzer:innen ermöglicht, die zu modellierenden Optionen aus einer reduzierten Menge von ausschließlich performance-relevanten Konfigurationsoptionen auszuwählen, ohne dass das resultierende Performance-Modell an Vorhersagekraft verliert. In der Evaluierung mit der Multi-Physik Anwendung Pace3D zeigen wir, dass der Ansatz Optionen akkurat als irrelevant für die Performance klassifizieren kann und trotz der zusätzlichen Schritte Messkosten für die Erstellung von Performance-Modellen sparen kann.

Unser dritter Beitrag ist eine Benchmarking-Suite für serverlose Workflows, um Entwickler:innen bei der Auswahl der richtigen Plattform für ihren Workflow zu unterstützen und die schnell wachsende Forschungsaktivität im Bereich der serverlosen Workflows durch die Bereitstellung einer Benchmarking-Methode und einer Vergleichsgrundlage zu fördern. Wir schlagen ein Modell für serverlose Workflows vor, das auf dem Petri-Netz-Formalismus basiert und eine plattformunabhängige Workflow-Definition zur Modellierung von Kontroll- und Datenflüssen bietet. Modellierte Anwendungen können automatisch in die proprietären Repräsentationen der Cloud-Plattformen übertragen werden, so dass Entwickler:innen nahezu identische Workloads auf verschiedenen Systemen ausführen können. Wir bewerten die Ausdruckskraft und den Overhead unseres Modells, indem wir die Literatur über serverlose Workflows prüfen. Wir stellen eine gebrauchsfertige Benchmarking-Suite mit Anwendungs- und Mikro-Benchmarks zur Verfügung und analysieren ausführlich Performance, Kosten, Skalierung, Stabilität und wie gut serverlose Plattformen wissenschaftliche Workflows unterstützen, indem wir unsere Benchmarks auf drei verschiedenen Plattformen ausführen.

Diese Arbeit trägt dazu bei, Methoden für Wissenschaftler:innen und Softwareentwickler:innen bereitzustellen, um ein besseres Verständnis der Anwendungs-Performance zu ermöglichen. Durch die Verringerung der Kosten und der Komplexität, die mit der Performance-Modellierung verbunden sind, und durch die Bereitstellung eines umfassenden Benchmarking-Frameworks ermöglichen die Beiträge dieser Arbeit Domänen-Wissenschaftler:innen ein besseres Verständnis und eine Optimierung der Performance ihrer Anwendungen, wodurch eine effizientere Nutzung von Ressourcen gefördert wird und wissenschaftliche Innovationen beschleunigt werden.

Contents

Part I.

Prologue

1. Introduction

Scientific computing describes the use of advanced computing capabilities to solve complex problems from different areas of science and engineering. It encompasses the development and application of mathematical models, numerical methods, and algorithms to simulate and analyze natural and engineered systems. The impact of scientific computing spans numerous fields, such as biology, physics, or material science. For instance, in biology, it enables complex simulations of molecular dynamics, which are essential for understanding biological processes at the molecular level. In physics, it allows for the modeling of particle interactions and the behavior of materials under extreme conditions, leading to discoveries in condensed matter physics and astrophysics. Material science leverages scientific computing to design and analyze new materials with desired properties, accelerating the development of advanced materials for various applications.

Applications in scientific computing can vary widely, ranging from single, standalone applications to complex, multi-step workflows. Single applications are often highly specialized programs designed to perform specific tasks, such as molecular dynamics simulations or finite element analysis. In contrast, scientific workflows integrate multiple computational tasks and data processing steps into a coherent, automated sequence. These workflows facilitate the management of complex analyses, ensuring reproducibility, scalability, and efficiency.

The computational demands of scientific computing necessitate the use of diverse computing infrastructures. Two prominent paradigms in the domain are supercomputing and cloud computing. Supercomputers offer high processing power, enabling the execution of large-scale simulations and data analyses at unprecedented speeds. These high-performance computing (HPC) systems are essential for applications requiring intensive computational resources and low-latency interconnects. Cloud computing, on the other hand, provides scalable and flexible computing resources over the internet. Platforms such as

Amazon Web Services, Google Cloud, and Microsoft Azure allow researchers to dynamically allocate computing resources, facilitating cost-effective and accessible computational power. The on-demand nature of cloud computing makes it particularly suitable for applications with variable workloads. Cloud computing has democratized access to high-performance resources, enabling a broader range of researchers to conduct advanced computational research.

Performance is an important quality attribute for scientific software as it affects how quickly and how much research can be done. Scientific simulations and data analyses often need to process large amounts of data and perform complex calculations, which can take a lot of time. Faster runtimes allow researchers to do more experiments in the same amount of time, speeding up the pace of discovery and innovation. Also, shorter runtimes can reduce computational costs, especially when using pay-per-use resources like cloud computing. Efficient applications enable researchers to execute more detailed simulations and test different simulation parameters, which leads to more accurate results. In areas where time is critical, like climate modeling and drug discovery, quick computational performance is essential for making timely decisions and responding to important challenges. Therefore, optimizing runtime performance is crucial for getting the most out of scientific research.

There are many challenges in developing and using scientific software and workflows, like complex configurations, ensuring scalability, and varying performance across different computing environments. These issues can make it hard to achieve good performance, slowing down research and making it less effective. However, this thesis focuses on two important aspects: configurability of scientific software and choosing the best option among multiple different workflow orchestration offerings of cloud platforms. By providing practical methods and tools, we aim to support scientists with these tasks, helping them to work more efficiently and effectively.

1.1. Configurable Scientific Software

Many software systems are configurable, allowing the user to set functional and non-functional properties of the software according to their needs. For

example, in a materials simulation [80], users select algorithm settings (non-functional) and properties to simulate (functional), impacting performance metrics such as response time and throughput [142, 165]. While they set fixed values for functional options, non-functional options can be chosen optimally depending on the execution environment and the choice of functional options. Choosing a set of parameters that yields the best performance is challenging, as it is non-trivial to determine how a single configuration option influences performance [73, 141]. Generally, developers and users do not know how configuration options interact and which combination of options will yield the best performance [34]. An example of such a challenge is understanding application scalability, i.e., the interaction between problem size and the number of processes [17].

Performance modeling helps users understand application behavior and guide further development by expressing application performance as functions of input parameters [78, 90]. Since the advent of High-Performance Computing (HPC), performance experts manually identify and model the parts of the application they consider critical to performance. Due to the high cost of performance experts, HPC developers often rely on evaluating application performance with a few configurations, with only their intuition to guide them in navigating the configuration space. Automatic performance modeling generates models from empirical measurements for all components of an application and can systematically cover all configuration options. Even though configuration spaces are constrained and smaller than combinatorial explosion would suggest [119], the number of options still makes exhaustive measurements infeasible.

Therefore, building empirical performance models for highly-configurable HPC applications is an *expensive* process. Two factors dictate the construction cost [37]: First, the number of required experiments required to measure the system's performance. It increases with every option added to the model (known as the curse of dimensionality). Second, the costs of running an experiment on an HPC system: The costs of operating an HPC system are in the range of millions of euros per year [23, 30]. While computing centers do their best to design and run clusters economically, developers must ensure that their applications scale efficiently on computing clusters. During the execution of performance experiments, the infrastructure is occupied, preventing other applications from running.

1.2. Scientific Workflows

To account for increasingly complex and data-intensive simulations, many scientific communities design their applications as workflows with many jobs that have complex constraints for the precedence of the different jobs [87]. Additionally to finding optimal configurations of the single applications, the challenge of orchestrating them arises. Serverless computing gained major adoption in the industry [47, 102], with 50-70% of cloud customers using serverless functions and containers [42]. In the last decade, serverless computing has become one of the most popular programming paradigms in the cloud with rising interest in the scientific community to use serverless solutions [48]. This interest is accompanied by experimentation with serverless offerings of the cloud platforms [105] and management systems for serverless execution of workflows [83, 85, 126, 127].

In the Function-as-a-Service (FaaS) programming model, users implement stateless functions and invoke them through a REST interface. The actual function deployment and resource scheduling becomes the responsibility of the cloud operator, who gains a significant advantage in optimizing resource utilization. Users are no longer concerned with managing their applications, and thanks to the pay-as-you-go billing system, they are charged only for resources used to handle function invocations. While the primitiveness of FaaS can be an important benefit, it is also a major drawback: a single function is insufficient to cover all use cases. Functions must be aggregated and connected to build larger applications, keep the design modular, target heterogeneous resources, or use pre-defined and standardized functions, e.g., for machine learning inference. For wider adoption in cloud applications, serverless needs a more capable programming model to support non-trivial control and data flow.

Serverless workflows introduced the ability to chain and aggregate multiple functions into a single application, creating a graph of functions and automating the execution of a sequence through control and data dependencies. Workflows include the most important control-flow components - conditions and loops - which allows them to represent full computations such as multi-stage machine learning pipelines or parallel processing of large data sets in a MapReduce job. Users implement functions and define the workflow structure in a cloud-specific format. They are not responsible for orchestrating the involved functions. Instead, cloud operators control the

workflow invocation and retain the ability to optimize resource consumption, e.g., through optimized function placement, oversubscription, targeting idle resources, and co-locating functions that depend on each other [1, 38, 103].

1.3. Challenges and Research Questions

Modeling frameworks for applications with many configuration parameters can be classified into black-box and white-box approaches. The former suffers from high sampling costs and relies on heuristics to reduce the experiment design [141], introducing the risk of excluding interactions between options from the sample set. The likelihood of this problem can be decreased by using more samples – leading to a hard to quantify trade-off between model accuracy and the number of samples [66, 94, 123]. White-box approaches can support modeling of numerical options but do not use known interactions between options, resulting in an expensive full-factorial experiment design [37] or using heuristics to reduce the number of samples [159]. Other white-box procedures concentrate on binary and binary-encoded options [153, 154] and do not consider numerical parameters such as the problem size or the number of processes.

> **Problem P1:**
> Automatic performance modeling with many parameters is expensive, and sampling optimization approaches rely on imprecise heuristics.

Moreover, in most cases, only a subset of the options strongly impact application performance [82, 94]. Therefore, it is possible to remove the performance-irrelevant ones from the experiments to be executed without affecting prediction accuracy of the resulting model. This speeds up model construction by reducing the number of required performance experiments that must be conducted on HPC computing systems. However, as the performance influences and interactions among configuration options are complex, it is hard to identify the configuration options that have a major impact on performance and are therefore worthwhile to model.

> **Problem P2:**
> Users have to intuitively select a small subset of options to create performance models, or decide to model all options, resulting in a hard-to-quantify trade-off between model quality and number of experiments.

Workflows have been adopted by the most popular commercial cloud platforms [7, 9, 62] and make up almost a third of serverless applications [48]. However, just like every FaaS platform is different [40], serverless workflows are quite distinct from each other. Not only do they offer different APIs and incompatible graph syntax and format, but they also have fundamentally different programming models: workflow platforms diverge in the statelessness of functions and the static nature of graph definition (Section 2.5). Serverless platforms already differ in their external features and internal behavior [40]. Even though FaaS platforms might seem like the same product, they offer drastically different performance, reliability, and cost [40, 104, 158]. With workflows built as an orchestration of already existing functions, the functionality of the new programming model is affected by both the orchestration service and the existing differences in the underlying compute infrastructure.

> **Problem P3:**
> End users need to conduct extensive and complex benchmarking of serverless cloud services to estimate the performance of their workloads and understand platform limitations.

Based on these problems, we define the following main research questions:

> **Research Questions:**
>
> **RQ1** How can the cost of automatic performance modeling be decreased while deterministically maintaining accuracy of the resulting models?
>
> **RQ2** How can performance-irrelevant configuration options be identified automatically?
>
> **RQ3** How can workflows be modeled and transcribed to different platforms to enable comparative evaluation of their performance?

1.4. Thesis Contributions

The contributions of this thesis address how the cost of automatic performance modeling methods can be decreased and how the performance of workflow orchestrations can be analyzed comparatively.

> **Contribution C1:**
> A white-box measurement methodology to derive an optimized minimal subset of required measurements for performance modeling.

As our first contribution **C1**, we present *Performance-Detective* [134], a novel white-box measurement methodology that significantly reduces experimentation costs while preserving the accuracy of the resulting models. Unlike previous sampling optimizations that rely on imprecise heuristics, this approach utilizes parametric performance models derived from a taint-based system analysis [37] to assess the impact of parameters on program functions. By conducting a step-by-step analysis of parameter interactions, *Performance-Detective* draws conclusions about these interactions within the program's control flow. Applying these insights to the experiment design eliminates unnecessary measuring points for modeling parameter interactions.

As a result, *Performance-Detective* can decrease the dimensionality of experiments from exponential to polynomial, while avoiding the risk of omitting important parameter dependencies. We empirically evaluate *Performance-Detective* through two case studies, demonstrating that it maintains the accuracy of the resulting performance models while reducing measurement costs by up to 34 times, compared to both the Extra-P empirical performance modeling tool [32, 123] and Performance-Influence Models [141].

> **Contribution C2:**
> An approach that automatically determines performance-irrelevant configuration options and removes them from the remaining modeling process.

The second contribution **C2** presents a method to automatically identify configuration options that do not impact performance [135]. Users can use the approach when creating performance models for their simulation scenario to filter performance-irrelevant options, enabling them to focus on a smaller subset of performance-relevant options for modeling without compromising the accuracy of the performance model.

This method involves three primary steps. First, users begin by conducting small-scale experiments to gather samples, considering all available configuration options. Next, we use an existing performance modeling technique to learn an initial performance model from these samples. Based on this model, we categorize each configuration option as either *performance-relevant* or *performance-irrelevant*. Configuration options deemed irrelevant are then excluded from the subsequent modeling process. We validate our approach using the multi-physics solver Pace3D [80], demonstrating that it accurately identifies performance-irrelevant options and can reduce the costs associated with creating performance models, even with the additional classification steps.

> **Contribution C3:**
> A serverless workflows benchmarking suite and a workflow model to model control- and data-flow, automatically transcribe the application into a cloud's proprietary presentations, and enable users to run near identical workflows on different systems.

The third contribution, **C3**, introduces a serverless workflows benchmarking suite [133] designed to support end users in selecting the appropriate platform for their workflows and to support the research activities in serverless workflows. This suite establishes a baseline and provides a benchmarking methodology for evaluating and comparing the optimizations and systems developed to enhance the performance and efficiency of workflows.

We propose and formalize a model for serverless workflows, extending a platform-agnostic workflow definition to capture both control-flow and data-flow [24]. This model allows workflows to be automatically transcribed into the proprietary formats of various clouds, enabling users to run nearly identical workloads across different systems. To evaluate our model, we review existing literature on serverless workflows, showing its expressiveness and negligible overhead. We improve and extend a benchmarking suite with six real-world application benchmarks, among them one scientific workflow, and four microbenchmarks [24] and deploy benchmarks to three major cloud platforms, extensively analyzing performance, cost, scaling, stability, and how well serverless platforms support scientific workflows.

Our research contributes to providing methodology for automatic performance modeling and analysis. The developed tools are easily applicable as they do not require performance engineering expertise, thus empowering domain scientists and end users to configure and deploy their applications for optimal performance.

1.5. Outline

The rest of this document is structured as follows: First, we introduce terminology and basic concepts used throughout the thesis in Chapter 2. Chapter 3 presents our first contribution *Performance-Detective*, using insights about the interplay of configuration options to deterministically reduce measurement cost. Chapter 4 covers our second contribution, the empirical identification of performance-irrelevant configuration options. In Chapter 5, we present our third contribution *SeBS-Flow*, a serverless workflows benchmarking suite and platform-agnostic workflow model. Finally, we conclude the thesis in Chapter 6 and discuss future work in Chapter 7.

2. Foundations

In this thesis, we introduce methodologies for automatic performance modeling of configurable scientific software and performance analysis of serverless workflows. In this chapter, we describe the foundations for our work and introduce key concepts and terminology.

First, we introduce configurable software and highlight specificities of configurable scientific software in Section 2.1. Second, we present different techniques to automated performance modeling in Section 2.2. Next, Section 2.3 introduces scientific workflows. Section 2.4 then explains key concepts of serverless computing before Section 2.5 presents and compares different serverless workflows platforms. Finally, Section 2.6 introduces Petri Nets and, more specifically, Workflow Nets as used for our workflow model later.

2.1. Configurable Software

Configurable software systems are designed with various adjustable parameters or configuration options that allow customization and optimization of the software regarding different execution environments, computations, or performance requirements. User can then modify these settings to tailor the software behavior to their needs for specific use cases during compile-time or during load-time [3].

The configuration space of configurable software systems can be classified into numerical and binary options. Binary options have two possible states, typically represented as *on* or *off*. These options enable or disable specific features or behaviors of the software [141], such as usage of a compression algorithm. In contrast, numerical options can take on a range of numerical values. They are often used to adjust existing functionality [141], thus fine-tuning the software's performance and behavior. Examples for numerical options are the problem size and resolution, and number of processes.

Further, we classify configuration options into functional and non-functional options: while the functional options define the problem to be solved, non-functional options are free to vary. Moreover, we distinguish between configuration options that affect performance and those that do not. Figure 2.1 shows examples for each type of option: The problem size is an example for a functional configuration option that also affects performance of the system. While the number of processes also impact performance, they do not influence the result of the computation. On the other hand, physical constants do influence the result of the computation, but are not performance-relevant as they do not influence the amount of computation done.

Potential constraints among the configuration options can limit the range of permissible configurations [119], i.e., a set of selected values for the configuration options of a software system. These constraints can include dependencies between options, such that setting one option requires enabling other options or that setting the value of one option restricts the acceptable values of others, e.g., setting the total problem size to solve may limit the number of processes that can be used.

Domains of configurable software systems include, but are not limited to databases, compilers, video encoders, and scientific software [34, 94, 166]. There are many challenges associated with developing and using configurable software systems, such as identifying which option will influence behavior of the system [84] or sampling valid configurations [51]. However, in the scope of this thesis, we focus on challenges regarding modeling the performance behavior of configurable software systems, as outlined in Section 1.3.

In the domain of scientific software, the challenges regarding the performance behavior are even more important as software is typically long-running and compute resources are expensive. Moreover, programs are usually parallelized using implementations of the Message Passing Interface (MPI) [109], the de-facto standard for communication between processes of a parallel application that run on a system with distributed memory. MPI supports point-to-point communication between individual processes as well as collective operations. To enable solving of bigger problems within reasonable time, users spawn more MPI processes, making scalability of applications a key concern for domain scientists.

	Performance-relevant	Not Performance-relevant
Functional	Problem size	Physical constant
Non-functional	Number of processes	

Figure 2.1.: Categorization of parameters depending on their type and effect on performance.

2.2. Empirical Performance Modeling

Performance models help users understand application behavior. They can be utilized to, e.g., pinpoint performance issues and therefore guide performance optimization [78, 141], analyze scalability of the software [32], and tune configurations [78, 166]. A complementary field of research is automated complexity analysis, aiming to automatically determine a theoretical upper bound of the computational cost of a program using static analysis [68, 125]. While the upper bound gives insights about the worst-case behavior of a system, it does not give insight into actual program performance for specific inputs in real-world environments.

In general, there are different approaches to performance modeling of software. We can distinguish between simulation, analytical modeling, and empirical modeling.

Simulation models are a virtual representation of the system that can be used to simulate its behavior under different scenarios. The accuracy of the predictions depends on the abstractions made during modeling the system and the granularity of the simulation: While there are accurate simulations for e.g., execution of code on a specific hardware, these are slower than actually executing the code on that hardware. Faster simulations, however, have to

15

abstract from details and therefore, bear the risk of abstracting away from critical details needed to accurately predict different scenarios and may not enable understanding the root cause of performance issues [78].

Analytical models use mathematical equations to represent system performance as a function of the considered input parameters, such as the number of processors used [32, 78]. Developers have to derive them manually by inspecting the source code and conducting performance tests of the software. This limits the applicability, as performance expertise is needed to create performance models, and limits the scope of performance models to selected kernels of an application, as manually creating models for the whole application would not be feasible [32].

In contrast, empirical performance models can be generated automatically based on measurement data collected from actual system runs, thus requiring neither a deep understanding of implementation details nor performance engineering expertise. This is especially beneficial in the context of software with many configuration options, as the influence of the options can be captured automatically. Moreover, it enables modeling of all functions within an application and updating performance models regularly during the development process [29].

Constructing empirical performance prediction models automatically usually involves four phases: First, the software system under consideration is analysed and the user selects parameters to consider for the model. Second, the experiment design phase, in which configurations to execute are determined. This involves utilizing a strategy for sampling from all the possible configurations that result from the parameters selected, as measuring all of them is not feasible. Sampling strategies can, for example, rely on achieving a specific coverage, mathematical criteria, or sample configurations randomly [66]. Next, the application under consideration is executed with the configurations derived from the experiment design phase to collect the required sample measurements. This step is the most expensive [37] as the user needs to conduct the performance experiments on the system they want to have a performance model for, typically a high-performance system. In the last step, the acquired measurement samples are supplied to the performance modeling tool that creates the empirical performance prediction model using a machine learning approach [66], such as Classification and Regression Trees or Multiple Linear Regression.
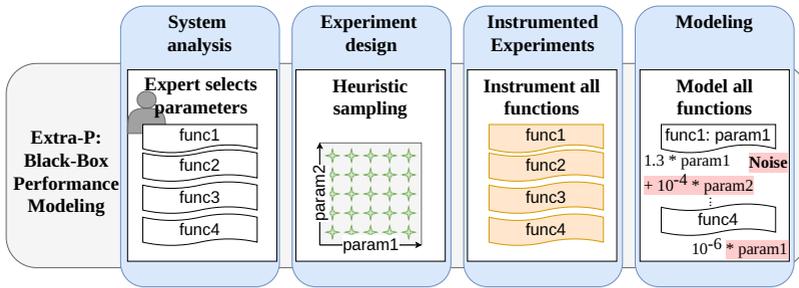
Figure 2.2.: Modeling workflow of Extra-P.

Different approaches can be applied to implement these steps. In the following, we present different automatic, empirical performance modeling workflows for both black-box (Section 2.2.1 and white-box modeling (Section 2.2.2).

2.2.1. Black-Box Modeling Workflows

Black-Box modeling treats the system to model as a "black box", solely relying on input-output data to create a model. This means that no knowledge of the internals of the system is needed or utilized. In the following, we first present the black-box modeling workflow of Extra-P in Section 2.2.1.1. Second, we introduce Performance-Influence Models in Section 2.2.1.2. Finally, Section 2.2.1.3 explains DECART.

2.2.1.1. Extra-P

Extra-P [32] is a performance modeling tool that expresses the effect of configuration parameters $x_i, i \in \{1, \ldots, m\}$ on a performance metric $f(x_1, \ldots, x_m)$. The result, such as $t(n, p) = 10 \cdot n \cdot \log(p)$, is a familiar, human-readable function, due to its similarity to how the complexity of algorithms is expressed. Figure 2.2 shows an overview of the modeling workflow.

In practice, the configuration parameters most often analyzed are problem size and process count, and the metric of interest is usually the runtime. This allows developers to identify performance bottlenecks in their applications.

Extra-P integrates with tools such as Score-P [93] that allows automatic instrumentation and measurement at the granularity of individual function calls, enabling automatic parsing and modeling of the resulting performance profiles. Binary configuration parameters, such as the choice between algorithms, are not supported.

The core assumption of the methods is that the complexity of most algorithms can be expressed using a small number of building blocks, summarized in Equation 2.2.1.1. The *Performance Model Normal Form (PMNF)* models a metric as a combination of polynomial and logarithmic expressions of configuration parameters:

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

This limits the possible search space sufficiently to allow for a fast traversal while still being sufficiently flexible to cover the overwhelming majority of applications.

In practice, gathering five measurement points for each parameter is sufficient, however, when considering the impact of multiple parameters simultaneously a full factorial design is necessary [29]. Recent work [123] proposes heuristics that lower the necessary measurements to a polynomial rather than exponential growth, at the cost of model accuracy. However, even this heuristic requires five measurements for each parameter, with additional samples providing diminishing returns with respect to accuracy and predictive power.

An issue not addressed in the state-of-the-art approaches is modeling more than three parameters, limiting their ability to scale to applications with many parameters. In practice, noise makes detecting parameters with a smaller impact on metrics of interest effectively impossible for even four parameters.

2.2.1.2. Performance-Influence Models

Performance-Influence Models (PIMs) [141] describe how configuration options and their interactions influence the performance of a system. They

support a theoretically unlimited number of binary and numerical configuration options. PIMs can be created using different sampling and learning techniques. Different heuristics are employed for sampling binary and numerical options [141].

Heuristics for sampling binary parameters include option-wise sampling, negative option-wise sampling, and pair-wise sampling. While option-wise sampling aims at providing one configuration per option such that the respective option is activated while all others are disabled. In contrast, negative option-wise sampling aims at disabling the respective option while activating all others. Pair-wise sampling selects configurations such that each pair of options is activated per configuration, while the other options are disabled. This results in the number of measurements being quadratic in the number of binary options.

Numerical options are sampled using four different heuristics: Box-Behnken, Plackett-Burman, Central Composite, and random selection, with the Plackett-Burman and random selection heuristics achieving best results. The extended Plackett-Burman design [157] requires defining a set number of levels for all of the configuration options, i.e., how many different values to sample across their value range. A sequence of numbers from zero to the number of levels is then used as seed to determine configurations to measure. The length of the seed depends on the number of configuration options and number of measurements the user is willing to execute. The complete configurations to measure are determined by enriching each of the configurations selected for the binary options with the numerical sampling set, resulting in $c_b \times c_n$ configurations to measure with c_b configurations selected for binary options and c_n configurations selected for numerical options. Samples are typically taken by measuring the total system runtime.

The models are then learned in a black-box manner by providing the used configuration and measured runtime as inputs to the selected technique. Siegmund et al. [141] initially proposed stepwise linear regression to fit models to a number of linear, quadratic, and logarithmic terms that represent the influence of options and their interactions. However, many techniques have been proposed since [153], such as Gaussian Processes and Classification and Regresion Trees. Due to the high number of supported techniques, it is generally not clear which combination of methods will provide the best accuracy of the resulting model. However, there are works [66, 88] investigating the interplay of sampling and learning techniques as well as the influence of the
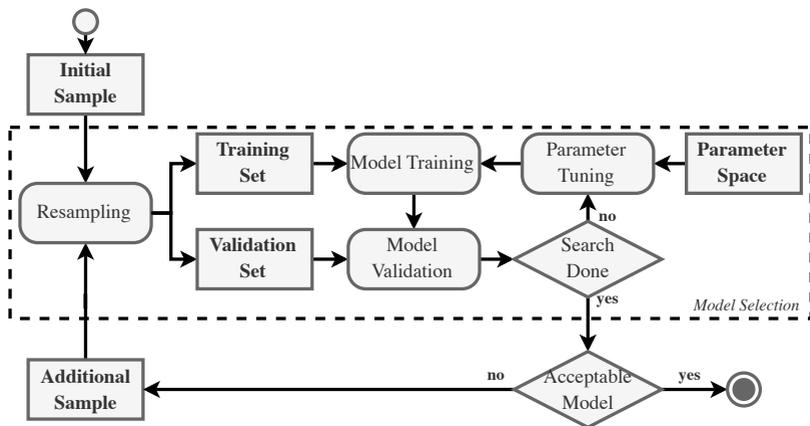
Figure 2.3.: Overview of DECART, as depicted in Guo et al. [70].

amount of samples on model accuracy and giving recommendations on how to choose between them.

2.2.1.3. DECART

The performance modeling tool DECART [70], building on CART [69], uses random sampling for the experiment design and Classification and Regression Trees as a learning technique to create a model of the correlation between option selections and performance measurements. Figure 2.3 shows an overview of the modeling process. DECART employs automated resampling and parameter tuning to reuse the available measurement data efficiently. It uses resampling to partition the samples into a training set for learning the performance prediction model and a validation set to evaluate the produced results. This allows integrated model validation without requiring additional validation measurement sets. Parameter tuning is used to systematically and automatically search through the parameter space of CART in order to find the parameter values that produce the performance prediction model with the highest prediction accuracy. DECART only supports binary configuration options. Support for numeric options is left for future extensions.

Figure 2.4.: Perf-Taint: applying the taint-based analysis to an application produces a JSON-like parametric performance profile.

2.2.2. White-Box Modeling Workflows

White-box performance modeling creates performance models based on a detailed understanding of the implementation of the software system, leveraging knowledge about the control flow of the software system to predict performance influences of parameters more accurately. By incorporating this detailed information, white-box models can provide more precise and reliable predictions compared to black-box models, which rely solely on input-output observations. In the following, we introduce two different workflows for white-box performance modeling in the context of configurable software: Perf-Taint in Section 2.2.2.1, and White-Box Performance-Influence models in Section 2.2.2.2.

2.2.2.1. Perf-Taint

Perf-Taint [37, 39] is a hybrid modeling tool that enhances the black-box and analytical modeling tools with program information. Perf-Taint applies a sequence of static and dynamic analyses to the program to understand how its computational effort is affected by a change in input parameters. The core assumption is that changes in performance metrics, such as the runtime, are related to to changes in the number of iterations of loop constructs. The result of the analysis is a *parametric profile* of a program, consisting of a list of performance-relevant functions and parameters which values can change the performance. Perf-Taint is built on top of LLVM [99], and it supports C/C++ applications and distributed parallel programs implemented with MPI. Internally, the tool uses static LLVM loop analysis [59] and DFSan [43], a dataflow taint analysis library. In addition, the taint analysis implementation is extended with support for control-flow tainting [163].

We present Perf-Taint on an example program with two functions in Figure 2.4. The function *important* is performance-relevant because its complexity changes with the value of parameter *x*. Thus, both the parameter and the function are included in the performance profile on the right side. On the other hand, the function *unimportant* involves only a constant amount of computation: Since the performance of the function is not affected by changes in the parameters, it should not be included in the modeling process, and its performance model will be a constant function. The analysis does not determine the actual number of loop iterations, nor does it analyze analytically the relation between *x* and runtime of function *important*. The result is binary, and it is the responsibility of the performance modeling tool to derive accurate and correct performance models from experimental measurements.

The information obtained from program analysis improves the overall modeling workflow by restricting the modeling tasks to program elements that are performance-relevant. Without the performance profiles, models with false parametric dependencies could be selected as good approximations for the experimental data because of noise. However, since the performance profile provides the parametric dependencies for each function, the modeler can remove models with false positives caused by noise in the experimental data. In practice, many short-running functions are particularly affected by noise and could otherwise generate false models that would make the modeling process more difficult and error-prone. If Perf-Taint identified them as constant, they can even be automatically excluded from the modeling process.

2.2.2.2. White-Box Performance-Influence Modeling

To overcome the trade-offs between measurement effort and accuracy in black-box performance influence modeling, Velez et al. [153, 154] proposed White-Box Performance-Influence Modeling for binary and binary-encoded configuration options. First, they proposed *ConfigCrusher* which uses static data-flow analysis to identify where the configuration options influence the control-flow statements of the program. Using these insights, they help find an optimal sample set for these options. However, the overhead of the static analysis limits the scalability of the approach to small systems. In contrast, *Comprex* uses dynamic data-flow analysis and is able to scale to large systems. It collects information about the control-flow influences of configuration
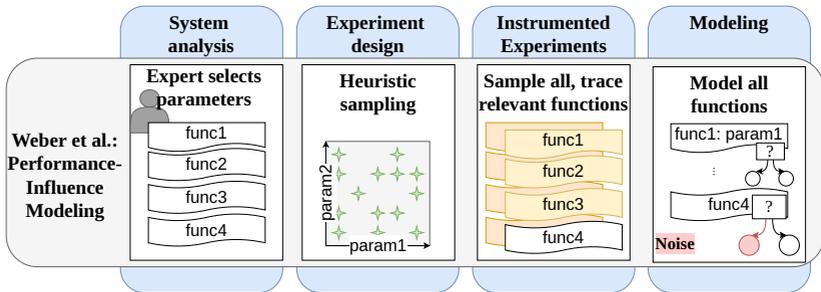
Figure 2.5.: Modeling workflow of Weber's PIMs.

options iteratively by repeatedly executing the system with different configurations. Thereby, Comprex derives different sets of configurations that result in different control-flows in certain regions. To build performance-influence models, they then systematically repeat executing and measuring the system with different configurations such that all different control-flow paths in all regions are executed at least once. The local models per region are then composed, eliminating the need for a learning technique. However, both approaches do not consider numerical options such as the problem size. The problem size is typically continuous and would be tedious to encode.

Weber et al. [159] present a two-step process for creating PIMs on the function level that considers numerical and binary options. Weber's PIMs use the extended Plackett-Burmann design [157] for sampling the numerical options and feature- and pair-wise sampling for binary options. As a learning technique, classification and regression trees are used. Figure 2.5 shows an overview of their modeling workflow. First, they execute and measure the methods' runtime for each configuration in the sample set and learn a PIM for every function based on these. For the functions that could not be learned with a specified accuracy, but also have a certain minimum runtime and contribution to the overall system runtime, they use tracing to enable learning more accurate models in a second step. In their work, white-box refers to learning PIMs on the function level. They do not analyze the system or use any knowledge about the system for learning.

2.3. Scientific Workflows

A wide range of scientific disciplines utilizes scientific workflows, such as bioinformatics, astrophysics, and earthquake science [19]. Scientific workflows are designed to represent and manage complex scientific analyses and computations, enhancing their reproducibility by providing a structured and documented sequence of steps [58]. Moreover, individuals steps can be reused and reconfigured for different analyses, supporting efficiency and flexibility.

Scientific workflows are structured sequences of computational or data-processing tasks used to conduct scientific research [52]. The tasks usually exhibit data dependencies, where the output of one task serves as the input for subsequent tasks. For example, a data preprocessing task must complete before the data analysis task can begin, thus implying a temporal ordering between the tasks. Quantities of data transferred between tasks are usually large [16]. Scientific workflows are typically visualized as directed acyclic graph, with nodes representing tasks and edges representing data dependencies [52]. Figure 2.6 shows an example with four different tasks: First, *preprocess* is executed. Once it finishes, the tasks *computeA* and *computeB* receive the output of *preprocess* and can start. Finally, *mergeResults* merges the outputs of the two preceding computations.

The constraints between the tasks can grow quite complex and large workflows can span a million individual tasks [52]. Workflow management systems help in creating and monitoring scientific workflows and can automate their execution [52] by scheduling tasks according to their data dependencies and passing data between them. This eases the design and execution of workflows for domain scientists, as they can abstract from technical details. However, different workflow management systems support different workflow execution models [52] and require different languages for specification of workflows, complicating comparatively evaluating different management systems and workflows.

2.4. Serverless Computing

Serverless computing brought two main innovations to the cloud: automatically scaled fine-grained workers and pay-as-you-go billing, where users are charged only for resources consumed and not allocated. We summarize the
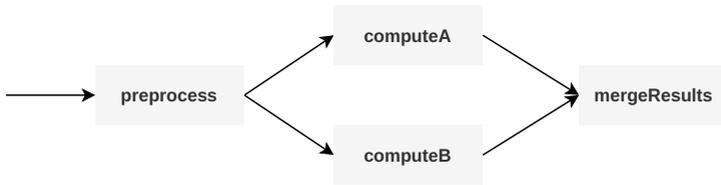
Figure 2.6.: Example workflow consisting of four different tasks, with arrows showing dependencies among tasks.
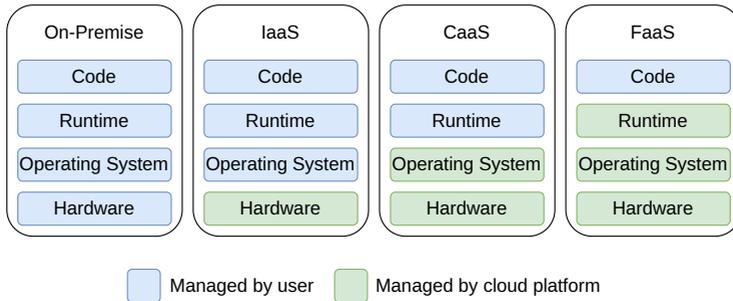
Figure 2.7.: Comparison of on-premise computing with the cloud compute models Infrastructure-as-a-Service (IaaS), Container-as-a-Service (CaaS), and Function-as-as-Service (FaaS) regarding respective management responsibilities.

main features and challenges of Function-as-a-Service (FaaS), the dominating programming model in serverless.

The runtime and deployment of serverless functions are controlled entirely by the cloud provider. Users only define functions by providing source code and dependencies (see Figure 2.7). Then, they invoke functions using *triggers* to pass the invocation data, e.g., by sending a POST request through a REST interface or uploading a file to the cloud storage. Function invocations are scheduled in dynamically allocated sandboxes, such as containers and micro virtual machines. This design allows cloud providers to optimize the placement of functions to reuse containers, target underutilized servers, and increase data locality. In this model, users pay only for the memory and CPU time they consume, and cloud operators are encouraged to oversubscribe resources and ensure as little idle time as possible. In addition to a more

```
tasks = []
for i in range(10):
  tasks.append(context.call_activity("process", i)
res = yield context.task_all(parallel_tasks)
```

Listing 2.1: Workflow invoking function *process* in parallel, with inputs from zero to nine and results written to *res*, implemented for Azure Durable Functions

efficient billing model, the other main advantage of serverless is that users are no longer responsible for scheduling their applications.

Functions are stateless and execute in an ephemeral environment. While consecutive invocations can be placed in the same container, data persistence is not guaranteed, as the cloud operator can evict the container at any time. Container reuse is preferred, resulting in *warm* invocations with all dependencies fetched and loaded to memory. On the other hand, *cold* invocations require spinning up a fresh container and performing the expensive initialization, resulting in significant overheads [40, 46, 110]. The cloud provider determines the CPU time allocation. In addition, functions are configured with a static and user-defined memory allocation (AWS, GCP) or have the memory dynamically allocated (Azure).

2.5. Serverless Workflows Platforms

Serverless workflows bring high-level abstractions to FaaS that support control flow, data passing parallelism, and fault tolerance. With workflows, users can build stateful and complex applications decomposed into functions connected with control-flow and data-flow conditions. While software engineers are increasingly interested in serverless applications [161], they encounter a wide range of challenges while developing them, with the first questions about the different capabilities of the platforms arising before starting the implementation [128, 161]: Workflows have been adopted by all major cloud providers, but their implementations are significantly different in capabilities and functionalities (Table 2.1). We focus on AWS Step Functions, Google Cloud Workflows, and Azure Durable Functions, as they play a leading role.

```
{
  "assign_array" : {
    "assign" : [
      {
        "array":[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
      }
    ]
  }
},
{
  "process": {
    "call" : "experimental.executions.map",
    "args" : {
      "workflow_id":"map-workflow",
      "arguments":"${array}"
    },
    "result":"res"
  }
}
"separate map-workflow:"
{
  "main" : {
    "params" : [ "elem" ],
    "steps" : [
      {
        "map": {
          "call":"http.post",
          "args": {
            "url":"cloud.google.process",
            "body": {
              "payload":"${elem}"
            }
          },
          "result":"elem"
        }
      },
      {
        "ret": {
          "return":"${elem.body}"
        }
      }
    ]
  }
}
```

Listing 2.2: Workflow invoking function *process* in parallel, with inputs from zero to nine and results written to *res*, implemented for Google Cloud Workflows

```
"init": {
  "Type": "Pass",
  "Result": "States.Array(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)",
  "ResultPath": "$.array",
  "Next": "map"
},
"map": {
  "Type": "Map",
  "ItemsPath": "$.array",
  "Parameters": {
    "payload.$": "$$.Map.Item.Value"
  },
  "Iterator": {
    "StartAt": "process",
    "States": {
      "process": {
        "Type": "Task",
        "Resource": "arn:process",
        "Parameters": {
          "payload.$": "$.payload"
        },
        "End": true
      }
    }
  },
  "ResultPath": "$.res",
  "End": true
}
```

Listing 2.3: Workflow invoking function *process* in parallel, with inputs from zero to nine and results written to *res*, implemented for AWS Step Functions.

The most significant change lies in the programming model, affecting the implementation of the workflows, with unknown implications to workflow performance, an important property for developers [161]. As the different implementations are all provider-specific, moving workflows from one platform to another is complicated, causing vendor lock-in [128]. Azure uses the programming model of Durable Functions [28], where the workflow definition is encoded within a regular program structure of an orchestrator. The graph of functions is expressed using a mainstream programming language such as Python, as seen in the example of invoking the *process* function ten times in parallel (Listing 2.1). The computation model is built on top of stateless *activity* and stateful *entity* functions. This distinction enables a flexible execution while providing an easy-to-use interface.

On the other hand, developers need to define their workflow using a state machine on Google Cloud Workflows and AWS Step Functions. The workflow consists of states representing computations and transitions that connect them together. The main states include function invocations that perform user-defined computation, while supplementary states encode control flow. State languages defined with a syntax based on JSON and YAML files can be limited and verbose and, therefore, more difficult to debug, with missing tool support for testing and debugging already being a problem for developers [102, 160].

The example implementations for Azure Durable Functions, Google Cloud Workflows, and AWS Step Functions in Listing 2.1, Listing 2.2, and Listing 2.3, respectively, demonstrate how simple code snippets can become much more verbose compared to a native implementation of orchestrator. All three workflows invoke the function *process* in parallel with inputs from zero to nine and write the results to *res*. Google Cloud Workflows and AWS Step Functions both require multiple states to implement this behavior: Google Cloud requires separate states for assigning and returning of variables and even a separate workflow for the *map* state invoking the functions in parallel. AWS Step Function is less verbose but still requires three different states. In Durable Functions, the implementation of the same behavior requires less work and the single-source implementation is more readable and easier to debug. However, the static form of a state machine gives the cloud provider deep knowledge of the functions executed and their order, allowing for optimizations. While AWS Step Functions only allows fully static workflows, i.e., all functions that could be called need to be known beforehand, Google Cloud Workflows also allows for transferring the URL of the function to be called as an input parameter.

Platform	Programming Model	Model Flexibility	Parallelism Limit	Interface
AWS	State Machine	Static	40	JSON
Azure	Orchestrator Function	Dynamic	Unlimited	Durable Functions
Google Cloud	State Machine	Semi-dynamic	20	JSON/YAML

Table 2.1.: Comparison of key features of different serverless workflow platforms.

Platform	Compute time	Function invocation	Orchestration (1000 transitions)
AWS	$0.0000166667/GBs	$0.20 per 1M	$0.025
GCP	$0.0000025/GBs	$0.40 per 1M	$0.01 (internal), $0.025 (external)
Azure	$0.000016/GBs	$0.20 per 1M	$0.000355329

Table 2.2.: Pricing of different platforms according to the cloud vendors' documentation [6, 8, 10, 60, 63].

The programming model also has an impact on the billing system. In addition to the cost of executing functions within a workflow, cloud providers charge users for workflow orchestration. In Azure, users have to pay for the duration of the orchestration function. In AWS and Google Cloud, users are charged per each transition of the state machine. Moreover, Google Cloud differentiates between external transitions to resources outside of Google Cloud and internal transitions. Table 2.2 shows an overview. Note that we have to estimate the orchestration cost on Azure as billing is only at the granularity of complete workflows. We do so by deploying a function chain, where each function just returns without performing any computation, and splitting the incurred cost between the functions and the orchestrator according to their runtime and execution count.

The platforms also have different limits for allowed parallelism: Azure Durable Function sets no limit on parallelism. However, both AWS Step Functions and Google Cloud Workflows limit the parallelism of workflows to forty and twenty concurrent function invocations, respectively. These limits can, however, be worked around by invoking sub-workflows.

As of now, the definition of benchmark applications is highly platform-dependent, and applications are complex to port between platforms. With the different platform-specific implications of implementing a workflow, it is difficult for developers to predict workflow costs on a given platform. To efficiently support them during the development of serverless workflows, we need a higher-level construct to abstract away the differences between platforms, enabling evaluation of the same workflow on different platforms and therefore facilitating informed decisions about the right platform.

2.6. Petri Nets

Petri nets [115] are a modeling tool to describe and analyze the flow of information in systems. They are particularly useful for modeling concurrent and asynchronous systems, providing a clear visualization of how different parts of a system interact. A Petri net is a triple $C = \langle P, T, F \rangle$ consisting of places P, a finite set of transitions T, and a set of directed arcs $F \subseteq (P \times T) \cup (T \times P)$, i.e., connecting either a place to a transition or a transition to a place. We show an example in Figure 2.8, consisting of five places and four transitions.

Figure 2.8.: Petri net with transitions $T = \{t_1, t_2, t_3, t_4\}$ and places $P = \{p_1, p_2, p_3, p_4, p_5\}$

Dynamic properties of the system are modeled using tokens that pass through the net as execution progresses. The state of the net is defined by the locations of tokens in it and called a marking. A transition is enabled if there are tokens in all its input places $\bullet t = \{p|(p, t) \in F\}$. When it fires, it removes the token(s) from its input place(s) and moves them to its output place(s) $t\bullet = \{p|(t, p) \in F\}$. Figure 2.9 shows an example: The marked net in Figure 2.9a has transition t_1 enabled, as there is a token (visualized as black dot) in its input place p_1. After t_1 fired, the marking of the net changes to the one shown in Figure 2.9b: The token has been removed from p_1 and tokens were moved to its output places p_2 and p_3. This, in turn, enables transitions t_2 and t_3. Note that these transitions can fire independently of one another and in any chronological sequence, thus enabling two different next markings of the net.

(a) Petri net with token in place p_1, enabling transition t_1 (marked green).

(b) Petri net after transition t_1 fired, placing tokens in places p_2 and p_3 which enables transitions t_2 and t_3 (marked green).

Figure 2.9.: Marked Petri nets before and after firing of t_1 with tokens visualized as black dots.

Workflow nets are an extension of Petri nets, usually used to depict business workflows. A Petri net is a workflow net *iff* there is a single source place start without incoming arcs, a single sink place without outgoing arcs, and every node is on a path from source to sink [150]. For example, our previous example in Figure 2.8 is no workflow net, as there is no single sink place.

Figure 2.10.: WFD-net with transitions $T = \{t_1, t_2, t_3, t_4\}$ and places $P = \{p_1, p_2, p_3, p_4, start, end\}$.

We later base our model for serverless workflows on workflow nets with data (WFD-nets), a further extension of workflow nets. A WFD-net [150] N is a tuple as follows:

$$N = \langle P, T, F, D, r, w, d, grd \rangle$$
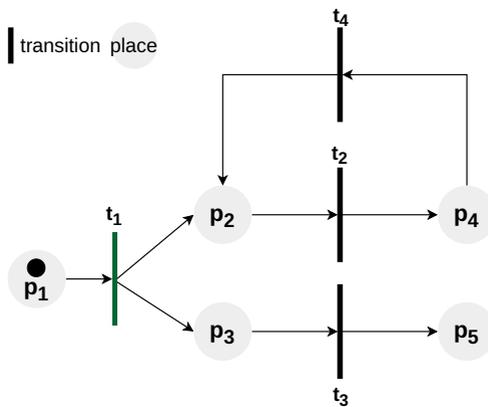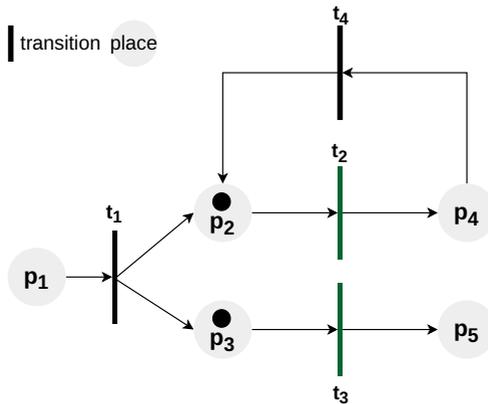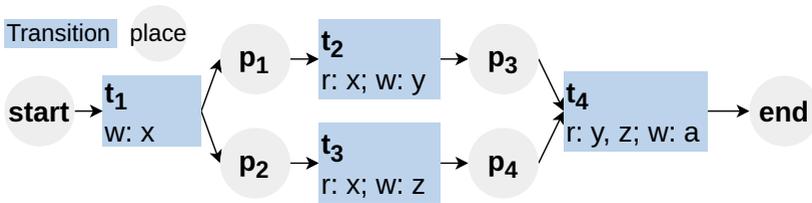
consisting of a Petri net $\langle P, T, F \rangle$, that is a workflow net, and additionally containing a set D of data elements on top as well as read, write, and destroy operations r, w, d on these data elements. The read operation is defined as $r : T \rightarrow 2^D$, and the write and destroy operations are defined similarly. Moreover, the guarding function $grd : T \rightarrow G_D$ can assign guards to transitions. We show an example WFD-net in Figure 2.10 where t_1 writes data to x, while t_2 and t_3 read from x and write to y and z, respectively. Finally, t_4 reads from y and z and writes the result of the workflow to a.

Part II.

Contributions

3. Experiment Design for Automatic Performance Modeling

In this chapter, we introduce *Performance-Detective*, a novel white-box modeling methodology that significantly lowers experimentation costs for creating performance models while maintaining the accuracy of the resulting models (Figure 3.1). In contrast to previous sampling optimizations that used imprecise heuristics, we use program information to *deduce* an optimized, minimal experiment design, removing measurement points that do not affect known interactions between non-functional parameters. We use *parametric performance models* obtained from the taint-based analysis [37] to understand the impact of parameters on program functions. Through a step-by-step analysis of parameter interactions, we derive conclusions on parameter interactions in the program's control flow. Applying the deduced conclusions to the experiment design removes measuring points unnecessary to model parameter interactions. Thus, *Performance-Detective* can reduce the dimensionality of experiments from exponential to polynomial while avoiding the risk of ex-

cluding parameter dependencies from the experiment. The experiment design of *Performance-Detective* is orthogonal to the modeling approach and can be used with black-box and white-box performance modeling.

The *deduced* experiment design makes performance modeling more affordable and is easily applicable alongside modern performance modeling systems. To quantify the increased efficiency and validate model correctness, we empirically evaluate *Performance-Detective* using a multi-physics solver and a particle transport application (see Figure 3.1). *Performance-Detective* maintains an accuracy of 91.4% and 89.2%, respectively, while reducing the costs of measurements by a factor of up to 34 times, compared against both the Extra-P empirical performance modeling tool [32, 123] and Performance-Influence Models [141].

This chapter thus constitutes our contribution C 1, the white-box measurement methodology *Performance-Detective*, composed of the following contributions:

- A novel *deductive* analysis that uses the results of performance tainting to derive an optimized minimal subset of required measurements out of a multi-dimensional configuration space.

- We identify main loops within applications using our deductive analysis, and leverage this to reduce the cost of experiments even further, in essence applying classical analytical performance modeling techniques to modern automatic modeling approaches.

- An extensive evaluation against state-of-the-art modeling workflows, proving high accuracy and reduced experiment size.

- Two case studies of modeling and evaluating applications using extrapolated and interpolated test points.

We focus on systems that operate primarily as batch jobs, where computations are initiated and run from start to finish without requiring user intervention. This assumption excludes interactive systems or those designed to provide on-demand services and support concurrent users, which are outside the scope of our approach.

(a) Multi-physics solver Pace3D.

(b) Particle transport application Kripke.
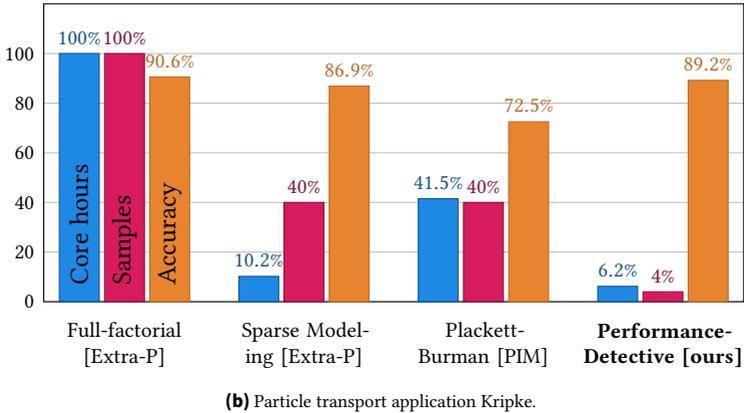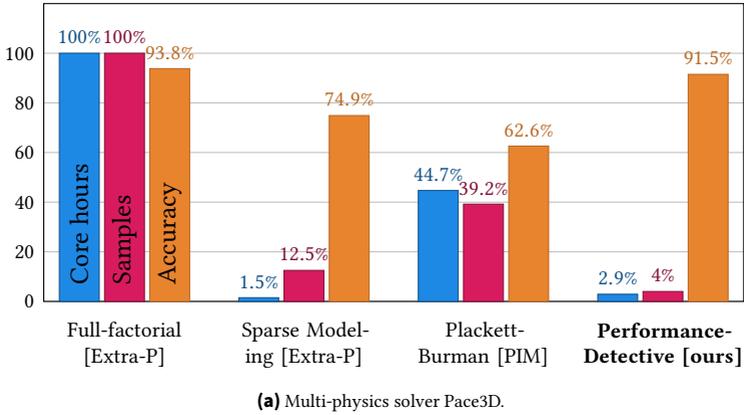
Figure 3.1.: Cost and accuracy of *Performance-Detective* vs Extra-P [32] and Performance-Influence Models (PIM) [141].

3.1. Limitations of Existing Modeling Frameworks

The state-of-the-art modeling approaches do not allow for efficient modeling of computing applications with many parameters and numerical options.

Performance-Influence Models (PIMs) [159], as introduced in Section 2.2.2.2, can derive performance models on the function level, but they do not use parameter dependencies and require expensive tracing instead of profiling.

Other White-Box PIMs, as introduced by Velez et al. [153, 154], do not support numerical options. Though numerical options can be encoded and discretized, the approach presented by Velez et al. [154] does not use a learning method and, therefore, can not predict configurations inter- or extrapolating training configurations.

Extra-P [32], as introduced in Section 2.2.1.1, relies on the *full-factorial* experiment design where all parameter combinations must be considered, leading to a combinatorial explosion of the number of measurements. To overcome this issue, heuristics have been proposed to optimize the experiment design [123], but they introduce a risk of missing parameter interactions and thus negatively affecting model quality. Furthermore, modeling with more than three parameters is particularly challenging due to the impact of noise present in measurements making it difficult to detect the impact of all configuration parameters simultaneously.

```
void f(int x) {
    for (int i = 0; i < x; i++) {
        calculate();
    }
}

void g(int x) {
    for (int i = 0; i < x; i++)
        calculate();
    }
}

f(x1);
g(x2);
```

Listing 3.1: Parameters x1 and x2 influence distinct functions.

Finally, Perf-Taint [37], as introduced in Section 2.2.2.1, does not use parameter knowledge to reduce the sampling set, and it does not propose a deterministic and effective methodology for manual user analysis of the experiment design. While Perf-Taint detects that there is no function depending on both x1 and

```
for (int i = 0; i < iters; i++) {
    f(x1);
    g(x2);
}
```

Listing 3.2: Parameter `iters` influences total runtime linearly.

Approach	Measurements	
	Lst. 3.1	Lst. 3.2
Extra-P	125	125
Perf-Taint	125	125
Performance-Detective	25	5

Table 3.1.: Measurements needed by Extra-P, Perf-Taint, and *Performance-Detective*.

x2 for the example in Listing 3.1, it does not make use of this knowledge to decrease the number of required experiments.

In Listing 3.2, all functions are called from a main loop. If the main loop is executed often enough, repetitions of experiments are not necessary as the repetition of the calculations already accounts for measurement noise. However, Perf-Taint does not make use of this to suggest fewer experiments. Even though 25, respectively 5, measurements are sufficient to model the performance of the examples in Listing 3.1 and Listing 3.2, Extra-P and Perf-Taint still require and suggest 125 measurements in both cases (cf. Table 3.1).

While heuristics for sampling can lower required measurements, they do so at the cost of lowering model accuracy. While more samples can increase the accuracy, there is no strategy on how to select them, leading back towards a full-factorial experiment design.

3.2. Approach

Figure 3.2 depicts the overall workflow of performance modeling, that we will detail in the following subsections, as compared to the modeling workflows of
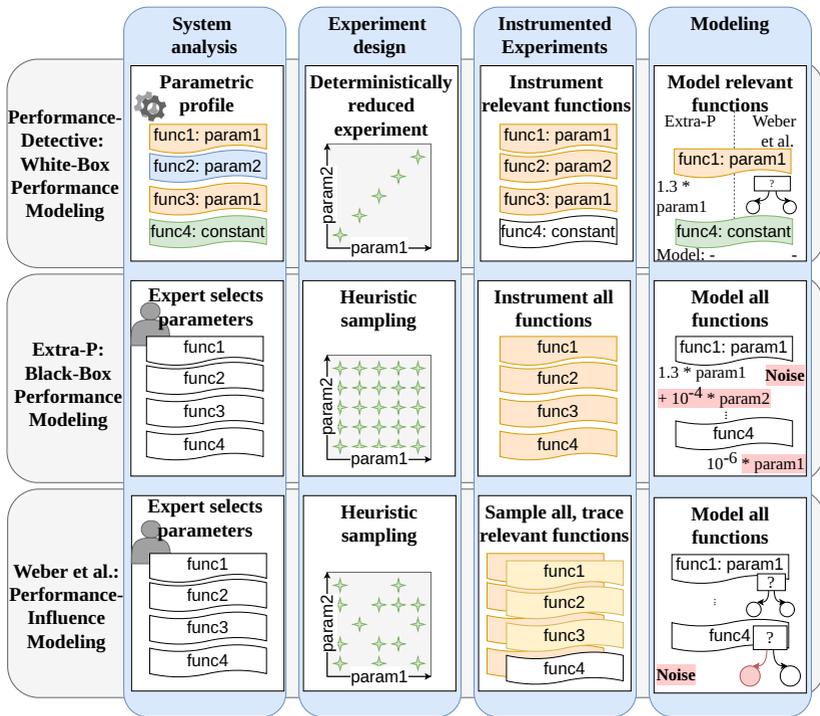
Figure 3.2.: Modeling workflow of *Performance-Detective* compared to Extra-P [29, 123] and Weber's PIMs [159]. Based on the system analysis, *Performance-Detective* deterministically reduces the experiment design.

Extra-P [29, 123] (Section 2.2.1.1) and Weber's PIMs [159] (Section 2.2.2.2). The deduction process of *Performance-Detective* is based on a program's parametric profile obtained with taint-based performance modeling provided by Perf-Taint (Section 2.2.2.1). *Performance-Detective* overcomes the limitations of existing performance modeling tools (Section 3.1), enabling efficient and reliable white-box modeling.

We start with the *system analysis* to trace the influence of configuration options on single functions (Section 3.2.1). *Performance-Detective* then uses the insights about which parameter influences the performance of which function to deduce a minimal *experiment design* that exploits insights about the interplay of options (Section 3.2.2). After executing the *instrumented*

experiments (Section 3.2.3), any learning methodology can be utilized for *modeling* the performance of the system (Section 3.2.4).

As motivated in Section 3, our approach targets systems that operate as batch jobs. Our assumption is that the system's performance varies in functions that contain non-constant loops, i.e., a configuration option determines how often they are executed and how many iterations they include. As introduced in Section 2.1, we classify configuration options along the two dimensions functional and non-functional, and performance-relevant and performance-irrelevant. In modeling, we aim to ignore options that affect neither the result nor the performance of a program, i.e., options that are non-functional and performance-irrelevant. However, the exact distinction between functional and non-functional options is subjective and domain-specific [154].

We will use the example program in Figure 3.3 as a running example. It takes the three parameters x1, x2, and iters as configuration options and does calculations based on them.

3.2.1. System Analysis

In the first step, we analyze the system to find out how configuration options influence its performance. We do this on a function level using the Perf-Taint approach introduced in Section 2.2.2.1. The Perf-Taint analysis outputs dependencies of functions to annotated parameters. Variables that are computed based on annotated parameters are considered to be dependent on all annotated parameters used for computing them. A function is always dependent on the variables that determine the number of iterations of loops in the function. If a function is called from within a loop, its performance is also dependent on the variables influencing the number of iterations in that loop. Additionally, the analysis gives us a list of performance-relevant functions.

While the user has to manually add the registration of variables corresponding to configuration options in the source code, the subsequent analysis is automated.

Example. In the running example in Figure 3.3, we manually add the registration of the input parameters x1, x2, and iters. The analysis then automatically determines the dependencies of the functions as shown in the

System: Parameters x1, x2, and iters

```c
void foo(int y, int z, int iters) {
  for (int i = 0; i < iters; i++) {
    for (int j = 0; j < y; j++) {
      //calculate something
    }
    bar(z);
    baz(z);
  }
}

void bar(int z) {
  for (int i = 0; i < z; i++) {
    //calculate something
  }
}

void baz(int z) {
  int flag = preCalculate(z);
  for (int i = flag; i < z; i++) {
    //calculate something
  }
}

int preCalculate(int z) {
  if (z < 10) return 0;
  return 1;
}

int main(int argc, char ** argv) {
  int x1 = atoi(argv[1]);
  int x2 = atoi(argv[2]);
  int iters = atoi(argv[3]);

  register_variables(x1, x2, iters)

  y = x1 * 5;
  z = x2 / 7;

  foo(y, z, iters);
}
```

time(**foo**) = f(**x1**, **iters**)

time(**bar**) = f(**x2**, **iters**)

time(**baz**) = f(**x2**, **iters**)

time(**preCalculate**) = constant

y depends on **x1**

z depends on **x2**

Figure 3.3.: Running example. The system takes the configuration options x1, x2, and iters.
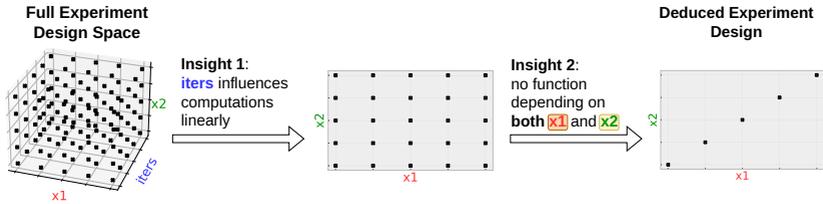
Figure 3.4.: Deduction of experiment design for the running example.

colored boxes and Table 3.2: As y and z are calculated based on the anno-tated parameters x1 and x2, respectively, they are dependent on them. The function preCalculate is not influenced by the configuration options and is identified as constant. foo is dependent on x1. bar and baz iterate over the parameter z and are therefore dependent on x2. We observe that iters has a multiplicative influence on the runtime of all functions and deduce that it linearly affects all computations. We do not consider foo to depend on x2 because we exclusively measure time spent in functions, therefore separating time spent in foo and bar and baz (cf. Section 3.2.3).

Function	Dependent on
foo	x1, iters
bar	x2, iters
baz	x2, iters
preCalculate	– (constant)

Table 3.2.: Dependencies of the running example in Figure 3.3

3.2.2. Experiment Design

Based on the derived dependencies from the system analysis, *Performance-Detective* deduces the minimal experiment design. We illustrate the approach of obtaining the experiment design in Figure 3.4. From a black-box view of the system, *Performance-Detective* starts with a full-factorial experiment design of the three parameters x1, x2, and iters. Since *Performance-Detective*

detects that `iters` influences the runtime of `foo`, `bar`, and `baz` linearly, measuring variations of `iters` will not provide us additional insights into the performance of the system. Therefore, *Performance-Detective* can exclude this parameter from the experiment design.

While the linear influence is easy to see in the example, this may not be the case for more complex real-world programs. To verify that the influence of the `iters` parameter is linear in such a case, we can measure execution times of loop iterations while executing the system for some value of `iters`. If the runtime does not change significantly between the iterations, i.e., the coefficient of variation is sufficiently small between iterations, we verify that the influence of the `iters` parameter is indeed linear.

Furthermore, we observe that `x1` and `x2` influence the performance of distinct functions and do not interact with each other. This means that changing the value of `x1` will only affect the runtime of `foo`, but not the runtime of the other functions. The same is true for `x2`: Changing its value will only change the runtime of `bar` and `baz`, but not the runtime of `foo`. Therefore, additional samples measuring their interactions will not provide further insights for performance modeling, and *Performance-Detective* can exclude them from the experiment design.

In general, we assume that if options influence different sets of functions, they do not interact with each other. Thus, *Performance-Detective* does not have to regard combinations of them for the experiment design and can vary them simultaneously. Also, *Performance-Detective* does not have to consider options linearly influencing system runtime, such as the number of iterations of the main loop. *Performance-Detective* derives both of these insights automatically based on the results of the system analysis.

As mentioned in Section 3.1, Extra-P requires a full-factorial measurement setup as of now [29]. Even the sparse modeling approach using heuristics to reduce the number of required measurements [123] requires at least five measurements per parameter. This requirement is to capture interactions between parameters. In contrast, *Performance-Detective* can detect the lack of interaction and strike out measurement configurations where only one parameter is changed, and the others are kept constant. This improvement reduces the dimensionality of the experiment compared to a full-factorial setup and provides further savings when compared to sparse modeling.

In the example above, Performance-Detective *deterministically reduces the number of measurements from 125 in a full-factorial experiment design to only five measurement points without sacrificing accuracy.*

3.2.3. Instrumented Experiments

We use an instrumentation-based approach to capture the total time spent in a function during one execution of the program. We do not separate different calls to the function for the same call stack but take the sum of all time spent inside the function during the execution. We measure time spent in a function exclusively, i.e., not including time spent in calls to other instrumented functions. Time spent in called, but not instrumented functions will be added to the calling function. We instrument all functions identified as performance-relevant by Perf-Taint, i.e., containing loops dependent on configuration options. Additionally, we instrument the main function executing the calculations to capture the total runtime of the program. This means that we do not instrument functions identified as constant.

Previous works suggested repeating each sample five times to account for measurement noise. However, when *Performance-Detective* identifies a linear dependency of parameter on the main calculation, *we can skip the repeated measurements and even halt execution after gathering at least five iterations of the main calculation loop.* While this approach is common in analytical modeling, it has been outside the reach of automated modeling due to the difficulty of identifying configuration parameters that control the number of executions of the main loop.

Example.   In Figure 3.3, *Performance-Detective* disregards the function `preCalculate` for instrumentation, as Perf-Taint identified it to be constant and we thus do not expect its performance to change. We instrument `foo`, `bar`, and `baz`, as they depend on annotated input parameters, and `main` to capture the total runtime of the application. In exclusive measurement, the time obtained for `foo` does not include the time spent in `bar`. However, the time of `baz` includes the time spent in `preCalculate`. As *Performance-Detective* identified that all functions depending on annotated input parameters linearly depend on `iters`, each configuration needs to be executed only once, using a value of five or greater for `iters`.

3.2.4. Modeling

Performance-Detective is orthogonal to the actual instrumentation and learning methodology, and it can be applied to reduce the costs of experiments in different performance modeling toolchains. The only requirement is that the modeling methodology is able to create a performance model based on empirical measurements of the software. In our evaluation (cf. Section 3.4), we consider two state-of-the-art modeling workflows: the black-box, empirical Extra-P performance modeling tool (Section 2.2.1.1) and Weber's Performance-Influence Models (Section 2.2.1.2). In both workflows, we create a performance model for each occurrence of a function within the call stack. To get a prediction of the overall performance of the entire program, we then sum up the estimates of each function.

3.2.5. Limitations

If all parameters are intertwined, no pruning of parameter combinations to measure is possible. Also, we do not regard binary options as switching between them results in performance jumps. By using Perf-Taint for the system analysis, we take on its assumption that performance-relevant behavior is located in computational loops and MPI communication routines. While for the applications in the evaluation it was possible to exclude the number of iterations, other applications may have a bigger co-variance across loop iterations or if-conditions that define the way loop iterations work. In the latter case, we rely on the output of the system analysis including information on tainting of control-flow branches. In both cases, *Performance-Detective* is conservative and includes the number of iterations as a parameter. Modeling of recursion is not supported, but recursive computations are rare in High-Performance Computing [37].

3.3. Case Studies

We illustrate the deduction process of *Performance-Detective* presented in Section 3.2 with two case studies: Kripke, a 3D discrete-ordinates (Sn) particle-transport proxy application, and a real-world case study from Pace3D (Parallel Algorithms for Crystal Evolution) [80], a multi-physics framework. For both

case studies, we only consider the execution of the main calculation loops as this is where most of the work happens.

Pace3D: Pressure calculation with the projected conjugate gradient method. Pace3D is developed at Karlsruhe Institute of Technology and University of Applied Sciences, Karlsruhe, since 2009. It encompasses around 550,000 lines of code and is used for digital materials research to simulate how a material reacts to outside influences, e.g., pressure or temperature. Performance is a key concern for the developers, as simulations can easily take multiple days [80, 136]. For our evaluation, we fix all functional parameters and model the pressure calculation with the projected conjugate gradient method. The calculation consists of two steps: First, an approximate solution is calculated on a coarse grid. This approximate solution is then used to calculate the real solution on the fine grid. The grids represent the material, with each cell corresponding to one cube of the material. The solver iterates until it reaches a convergence criterion or a given maximum number of iterations. We consider the number of processes and the number of coarse grid cubes as non-functional parameters. We also use the size of the material to predict how much material each process should get to achieve the best performance. Additionally, we consider the maximum number of iterations.

Kripke. Angular fluxes are calculated using different numbers of directions and groups. Kripke was built to research how different data layouts, programming paradigms, and architectures influence the performance of Sn solvers [97]. It enables exploring new programming paradigms and architectures in a lightweight fashion, allowing an evaluation how the data layout should look like and later adapting larger codes according to this evaluation. Therefore, it is important to thoroughly understand Kripke's performance. Moreover, Kripke has been used as case study for performance modeling in previous work [29, 31]. We consider the number of direction sets as well as the number of processes as parameters. Additionally, we are interested in the number of iterations.

3.3.1. System Analysis

We analyze both scenarios with Perf-Taint [37]. The only necessary modifications to the source code are annotating and registering the variables

corresponding to the parameters of interest when they are read from the user-provided configuration. We run the analysis on a small problem size using only a few iterations, assuming that the computations and analysis results are representative, and validate the results with larger measurements in Section 3.4.2.

3.3.1.1. Pace3D

We annotate variables in the code corresponding to the number of processes (*procs*), material size (*vol*), coarse grid size (*cubes*), and spacing of the coarse grid (*spacing*). Additionally, we annotate the number of iterations for the fine as well as for the coarse grid. This results in a total of 21 lines of code added.

The coarse grid spacing is a dependent parameter, i.e., not linearly independent, and is calculated by dividing the material size by the coarse grid size:

$$spacing = vol/cubes$$

Hence, we can replace the spacing with $vol/cubes$, substituting it with the independent parameters it is defined by. While we do this replacement manually for our case study, this could easily be automated. Figure 3.5 shows an example where we can then conclude automatically that the material size determines the performance of the loops: In the outer loop, the program iterates over the size of the coarse grid, whereas in the inner loop, it iterates over the spacing to finally access each of the fine grid cells. Replacing *spacing* with $vol/cubes$, we get:

$$complexity = cubes \cdot spacing = cubes \cdot (vol/cubes) = vol$$

Thus, we can conclude that it is the material size that determines the performance of the loops.

From the results of the system analysis, *Performance-Detective* automatically identifies that the maximum number of iterations on the fine grid determines the runtime of the main calculation. To verify that the influence is linear, we trace the iterations of the loop for one execution of the program and check whether the runtime of a single loop iteration changes. To do so, *Performance-Detective* calculates the coefficient of variation between them. In our scenario,

```
for (int i = 0; i < cubes; i++) {
    //spacing = vol / cubes
    for (int j = 0; j < spacing; j++) {
        //complexity = cubes * spacing
        //complexity = cubes * (vol / cubes)
        //complexity = vol
    }
}
```

Figure 3.5.: Program iterating over the spacing

(a) Pace3D   **(b)** Kripke
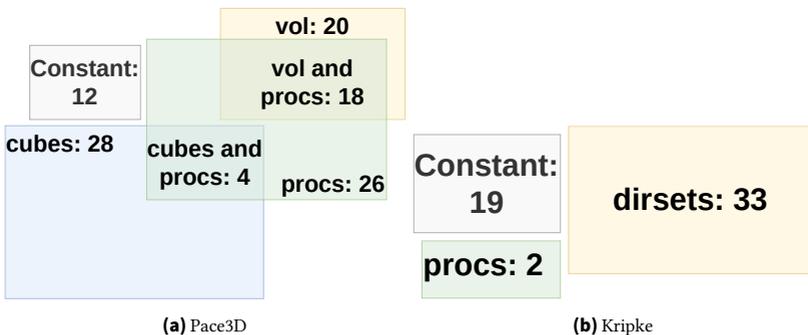
Figure 3.6.: Number of functions depending on the annotated parameters per case study as Venn diagrams.

the coefficient of variation is 0.05 for 100 loop iterations. We conclude that the influence of *iters* is indeed linear.

Furthermore, the analysis results show that 12 functions are constant, and 54 are dependent on the annotated parameters. For the 2 functions dependent on the spacing, we can automatically assess that they in fact depend on the total volume, using replacement as shown in the example in Listing 3.5. The results, as shown in Figure 3.6a, show that the material size *vol* and the coarse grid size *cubes* influence different functions. However, both interact with the number of processes *procs*.
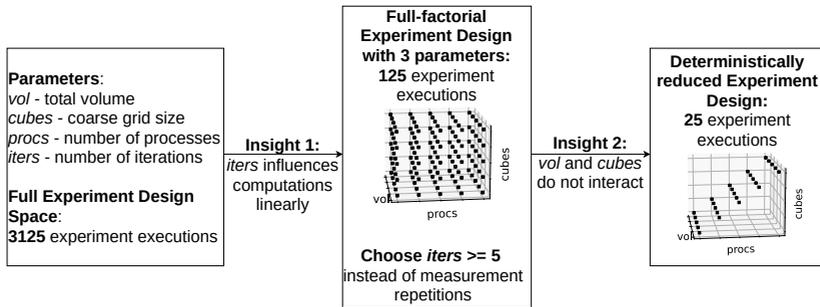
Figure 3.7.: Deducing the minimal experiment design for Pace3D.

3.3.1.2. Kripke

We annotate the number of processes (*procs*) as well as the number of direction sets (*dirsets*) and iterations (*iters*). This results in adding a total of five lines of code. Using the results from the system analysis, *Performance-Detective* analyzes that the number of iterations determines the runtime of the main calculation. As before, to verify that the influence is linear, we trace the loop iterations of one program execution. As the coefficient of variation between the measured 10 iterations is *0.0067*, we conclude that the influence of *iters* is in fact linear.

Moreover, the analysis shows that 19 functions are constant and 35 are dependent on the annotated parameters, with 33 depending on *dirsets* and 2 on *procs*. The number of processes influences different functions than the number of direction sets. We visualize the results in Figure 3.6b.

3.3.2. Minimal Experiment Design

Figure 3.7 shows the process of *Performance-Detective* deducing the experiment design for Pace3D based on the insights gained from analyzing the system. From a black-box view at the system, *Performance-Detective* starts with four parameters and a full-factorial experiment design, leading to 625 measurement points. To account for measurement noise, each point has to be executed five times, leading to a total of 3125 experiment executions.

As we know that *iters* linearly influences the runtime of the main computation, *Performance-Detective* can exclude *iters* from the parameter space (*Insight 1*). Also, we can skip repetitions of the execution of measurement points and choose a sufficiently high value ($\geq$ 5) for *iters* instead. This reduces the experiment design to 125 points that need to be executed only once. We also know that the functions affected by the coarse grid size are distinct from those affected by the total size. Thus, *Performance-Detective* can strike out configurations aiming to find interactions between them from the experiment design (*Insight 2*). This means that *Performance-Detective* varies *vol* and *cubes* simultaneously. As *procs* interacts with *vol* and *cubes* according to the analysis, *Performance-Detective* includes *procs* as interacting parameter into the experiment design and varies *procs* independently from *vol* and *cubes*. Figure 3.8 (Section 3.4.2) shows the resulting training data points for Pace3D as crossed circles: While we only measure the combinations (*vol*, *cubes*) = (*216000*, *1000*), (*432000*, *1728*), (*864000*, *3375*), (*1728000*, *8000*), (*2592000*, *27000*), we have to measure these combinations for all five numbers of processes *procs* = (*8*, *16*, *32*, *64*, *128*).

For Kripke, *Performance-Detective* deduces the experiment design similarly: As *iters* has a linear influence on the runtime of the main computation, *Performance-Detective* can exclude *iters* from the parameter space (*Insight 1*) and we can skip repetitions of experiments. Furthermore, *Performance-Detective* can remove configurations aimed at finding interactions between *procs* and *dirsets*, as they influence distinct sets of functions (*Insight 2*). Thus, *Performance-Detective* varies *procs* independently of *dirsets*, resulting in 5 measuring points with (*p*, *dirsets*) = (*8*, *8*), (*16*, *16*), (*32*, *24*), (*64*, *32*), (*128*, *64*).

Performance-Detective reduced the full experiment design space of 3125 experiment executions to only 25 executions needed for Pace3D, and from 625 to 5 for Kripke.

3.4. Evaluation

To evaluate *Performance-Detective*, we assess the accuracy of the performance models generated for the case studies presented in Section 3.3. As *Performance-Detective* is orthogonal to the instrumentation and learning methodology, it can be applied to reduce the costs of experiments in different performance

CPU	Intel Xeon Gold 6230 2.1GHz
Cores	40 on 2 sockets
Memory	96 GB
GCC	10.2 (Pace3D), 11.2 (Kripke)
MPI	OpenMPI 4.0.5 (Pace3D), OpenMPI 4.1.2 (Kripke)
Software	Score-P 7.0 [93] (Pace3D), Score-P 7.1 (Kripke), Perf-Taint, Extra-P

Table 3.3.: Measurement environment

modeling toolchains. We consider two state-of-the-art modeling workflows: the black-box, empirical Extra-P performance modeling tool (Section 2.2.1.1) and Weber's Performance-Influence Models (Section 2.2.2.2).

We evaluate the reduction of repetitions by inclusion of iterations and variation of independent parameters individually. Therefore, we formulate the following sub-research questions to RQ 1:

RQ1 How can the cost of automatic performance modeling be decreased while deterministically maintaining accuracy of the resulting models?

> **RQ1.1** What is the model accuracy when generating it from a single measurement with a high number of iterations?
>
> **RQ1.2** What is the model accuracy when generating it from a minimal experiment design by varying independent parameters simultaneously?

To answer the RQs, we compare our models with models generated following the conventional experiment and sampling designs. The expected outcome of the evaluation is maintained accuracy while reducing the dimensionality of experiment design and not repeating experiment executions, resulting in significantly decreased cost for measurements. For assessing the accuracy of the models, we evaluate them with testing configurations that inter- and extrapolate the training configurations. We measure cost of a model in core hours needed for executing the measurements required for creating it.
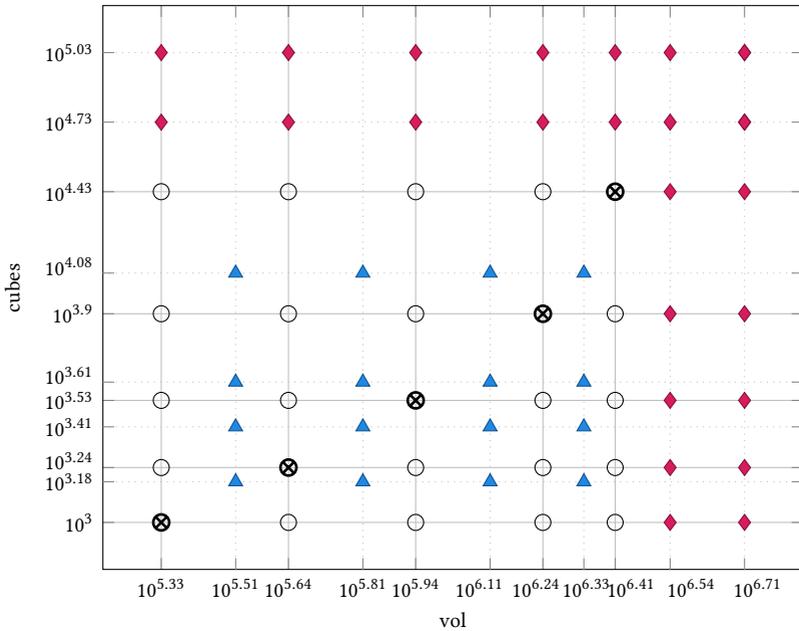
Figure 3.8.: Overview of training and test points used for Pace3D on logarithmic scales. Training data *Performance-Detective*: ⊗ , training data full-factorial: ◯ and ⊗ . Test data interpolated: ▲ , test data extrapolated: ◆ .

3.4.1. Instrumented Experiments

Table 3.3 shows the hardware and software systems used for measuring. We instrument the application using the list of important functions generated by Perf-Taint and repeat the measurement of each configuration five times to assess RQ1 and RQ2 separately. The coefficient of variation between the repetitions of the same configuration is 0.1 or less for all configurations. We always use a filter file containing important functions gained from system analysis to instrument only relevant functions to compare the predictions of all models. Otherwise, the evaluation would be less meaningful because the models generated when instrumenting all functions cannot predict the actual execution time as they have more profiling overhead that we cannot remove from the measurements.

3.4.2. RQ1.1: Modeling Using a Single Measurement

To evaluate whether the inclusion of iterations in the model can simplify the experiment design, saving us repetitions of the measurements, we generate a model from a single execution of each measurement and compare the predictions of this model with test points. For evaluating the accuracy, we use the mean time of the five repeated executions of each test measurement point. To obtain a prediction of the total execution time, we sum up the predictions for the single functions and evaluate them against the execution time of the evaluation configurations.

We use two testing sets, containing either inter- or extrapolated measurement points. Figure 3.8 shows an overview of the test data sets used for Pace3D. For the interpolated test data of both case studies, we measure configurations in between the training data points. For Pace3D, we do not interpolate the number of processes as this would not provide insight into the quality of the minimal experiment design: We still treat the number of processes as a separate parameter and vary it separately from *vol* and *cubes*, as it interacts with both according to the analysis. For the respective independent parameters, we use points between each *vol* and *cube* value used for training, as shown in Figure 3.8. This results in 80 measuring points for Pace3D and 16 for Kripke with the interpolated values $procs = (12, 24, 48, 96)$ and $dirsets = (12, 20, 28, 48)$.

For testing how well the model predicts extrapolated configuration points, we extrapolate parameters identified as independent and measure them together with the respective other parameter using a value used for the training set. This means that we measure the extrapolated values 54,000 and 108,000 for *cubes* together with the extrapolated values 3,456,000 and 5,184,000 for *vol* as well as with all the values for *vol* and *cubes* used for the training set, shown in Figure 3.8, for Pace3D. Following this approach results in 119 measuring points for the extrapolated test data set of Pace3D and 24 for Kripke, for which we extrapolate *procs* to 256 and 512 and *dirsets* to 96 and 128.

The results in Table 3.4 show that the accuracy remains about the same when using only one execution as for the models generated from the mean times of five repetitions. This suggests that it is sufficient to execute the measurement of each configuration point only once if the main calculation is executed sufficiently often. For Pace3D, we set the number of iterations of the main calculation to 100, and to 10 for Kripke. However, even five iterations per

Modeling approach	#executions	Mean error Pace3D		Mean error Kripke	
		interpolate	extrapolate	interpolate	extrapolate
Extra-P	5	8.31 %	9.31 %	4.56 %	18.32 %
Extra-P	1	8.05 %	8.74 %	4.33 %	15.17 %
Decision Trees	5	21.61 %	60.15 %	21.79 %	25.68 %
Decision Trees	1	22.62 %	62.80 %	22.38 %	25.31 %

Table 3.4.: Mean error of models generated from *Performance-Detective* experiment design using a single application execution vs five repetitions.
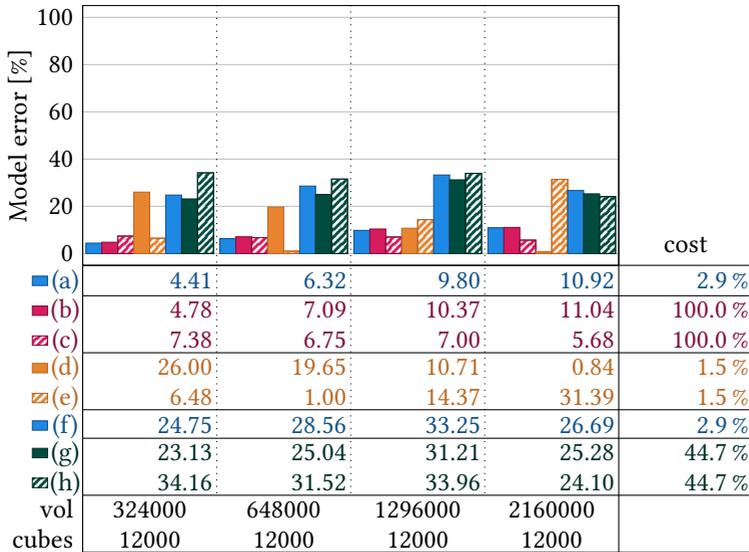
Experiment Design	Modeling approach	Perf-Taint?	Cost Pace3D		Mean error Pace3D	
			core hours	#experiments	test set interpolated	test set extrapolated
Performance-Detective	Extra-P	✓	10.9	25	8.05 %	8.74 %
Full-factorial	Extra-P [Cal16]	✓	367.55	625	9.5 %	10.7 %
Full-factorial	Extra-P [Cal16]	–	367.55	625	6.2 %	6.3 %
Sparse	Extra-P [Rit20]	✓	5.54	80	8.9 %	17.7 %
Sparse	Extra-P [Rit20]	–	5.54	80	15.0 %	31.8 %
Performance-Detective	Decision Trees	✓	10.9	25	22.6 %	47.7 %
Plackett-Burman	Decision Trees [Web21]	✓	164.37	245	20.1 %	45.4 %
Plackett-Burman	Decision Trees [Web21]	–	164.37	245	27.0 %	44.2 %

Table 3.5.: Mean error of different models for Pace3D for interpolated and extrapolated test points.

application run should be enough compared to one iteration per repetition of the experiment. While the total time measured in the main calculation stays the same for both variants, we can save the initialization overhead by executing it more often in only one execution.

Experiment Design	Modeling approach	Perf-Taint?	Cost Kripke		Mean error Kripke	
			core hours	#experiments	test set interpolated	test set extrapolated
Performance-Detective	Extra-P	✓	5.3	5	4.3 %	15.2 %
Full-factorial	Extra-P [Cal16]	✓	85.9	125	7.3 %	17.0 %
Full-factorial	Extra-P [Cal16]	–	85.9	125	6.7 %	11.2 %
Sparse	Extra-P [Rit20]	✓	8.8	50	24.6 %	34.2 %
Sparse	Extra-P [Rit20]	–	8.8	50	7.7 %	16.7 %
Performance-Detective	Decision Trees	✓	5.31	5	22.4 %	25.3 %
Plackett-Burman	Decision Trees [Web21]	✓	35.66	50	15.8 %	35.1 %
Plackett-Burman	Decision Trees [Web21]	–	35.66	50	21.8 %	31.2 %

Table 3.6.: Mean error of different models for Kripke for interpolated and extrapolated test points.

					cost
■(a)	4.41	6.32	9.80	10.92	2.9 %
■(b)	4.78	7.09	10.37	11.04	100.0 %
▨(c)	7.38	6.75	7.00	5.68	100.0 %
■(d)	26.00	19.65	10.71	0.84	1.5 %
▨(e)	6.48	1.00	14.37	31.39	1.5 %
■(f)	24.75	28.56	33.25	26.69	2.9 %
■(g)	23.13	25.04	31.21	25.28	44.7 %
▨(h)	34.16	31.52	33.96	24.10	44.7 %
vol	324000	648000	1296000	2160000	
cubes	12000	12000	12000	12000	

■ (a) Performance-Detective (Extra-P) ■ (d) [Rit20], Perf-Taint ■ (g) [Web21], Perf-Taint
■ (b) [Cal16], Perf-Taint ▨ (e) [Rit20] ▨ (h) [Web21]
▨ (c) [Cal16] ■ (f) Performance-Detective (Decision Trees)

Figure 3.9.: Accuracy of models for the Pace3D case study for excerpts of the interpolated data set with 64 processes.

3.4.3. RQ1.2: Varying Independent Parameters Simultaneously

The central question in our evaluation is whether we can maintain accuracy of the model when generating it from a minimal experiment design. To answer this, we compare the accuracy of our *Performance-Detective* model to models created with conventional experiment or sampling designs.

For the Extra-P multi-parameter modeler [29], this means using a full-factorial setup with 125 measuring points for Pace3D and 25 for Kripke. For the Extra-P sparse modeler [123], we use 5 values for each parameter while keeping the others constant and one interaction point for each parameter combination, resulting in 16 points for Pace3D and 10 for Kripke.
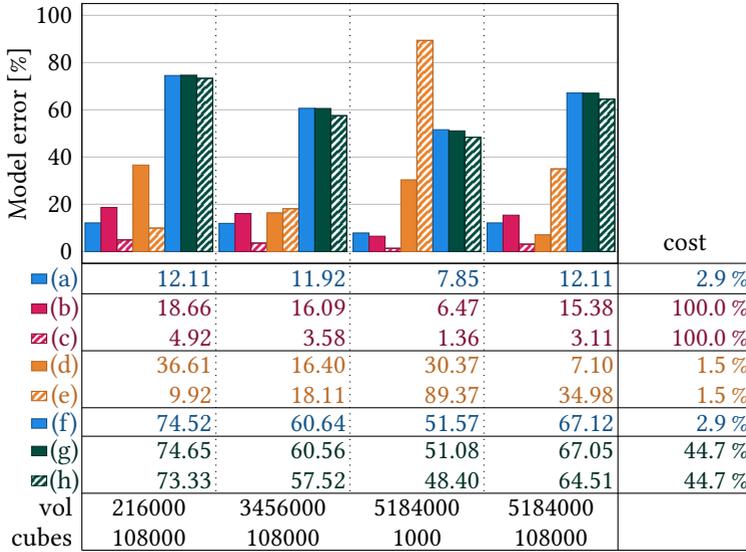
Figure 3.10.: Accuracy of models for the Pace3D case study for excerpts of the extrapolated data set with 64 processes.

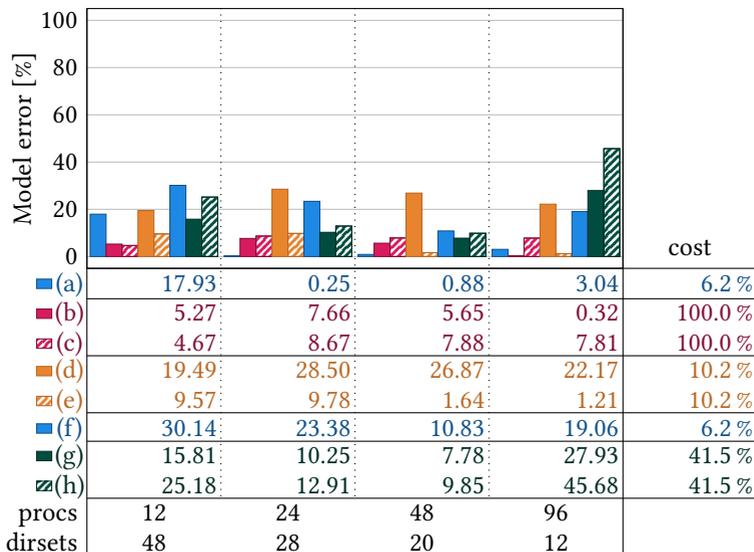|					cost
(a)	12.11	11.92	7.85	12.11	2.9 %
(b)	18.66	16.09	6.47	15.38	100.0 %
(c)	4.92	3.58	1.36	3.11	100.0 %
(d)	36.61	16.40	30.37	7.10	1.5 %
(e)	9.92	18.11	89.37	34.98	1.5 %
(f)	74.52	60.64	51.57	67.12	2.9 %
(g)	74.65	60.56	51.08	67.05	44.7 %
(h)	73.33	57.52	48.40	64.51	44.7 %
vol	216000	3456000	5184000	5184000	
cubes	108000	108000	1000	108000	

- (a) Performance-Detective (Extra-P)
- (b) [Cal16], Perf-Taint
- (c) [Cal16]
- (d) [Rit20], Perf-Taint
- (e) [Rit20]
- (f) Performance-Detective (Decision Trees)
- (g) [Web21], Perf-Taint
- (h) [Web21]

To learn PIMs, we use the extended Plackett-Burman Design [157] with 49 samples (Pace3D) or 10 samples (Kripke) and five levels, effectively being a subset of the full-factorial measuring points. We use the script of Weber et al. [159] to learn PIMs on a function level based on known parameter dependencies. However, we do not use their two-step process but only one profiling step using compiler instrumentation. We think this can even be beneficial for the quality of the resulting models: Because they use sampling in the first step, periodically interrupting system execution, they have a statistical chance of missing function calls. The second step uses tracing to learn more accurate models of performance-relevant functions that could not be learned from the sampling step. However, tracing imposes a significant overhead on the executions, possibly distorting the quality of the performance model learnt from them. In contrast to Weber et al., we use mean values

					cost
■ (a)	17.93	0.25	0.88	3.04	6.2 %
■ (b)	5.27	7.66	5.65	0.32	100.0 %
▨ (c)	4.67	8.67	7.88	7.81	100.0 %
■ (d)	19.49	28.50	26.87	22.17	10.2 %
▨ (e)	9.57	9.78	1.64	1.21	10.2 %
■ (f)	30.14	23.38	10.83	19.06	6.2 %
■ (g)	15.81	10.25	7.78	27.93	41.5 %
▨ (h)	25.18	12.91	9.85	45.68	41.5 %
procs	12	24	48	96	
dirsets	48	28	20	12	

■ (a) Performance-Detective (Extra-P) ■ (d) [Rit20], Perf-Taint ■ (g) [Web21], Perf-Taint

■ (b) [Cal16], Perf-Taint ▨ (e) [Rit20] ▨ (h) [Web21]

▨ (c) [Cal16] ■ (f) Performance-Detective (Decision Trees)

Figure 3.11.: Accuracy of models for the Kripke case study for excerpts of the interpolated data set.

derived from the repetitions of the experiment for modeling. They build a separate model for each repetition of the experiment.

As we showed in Section 3.4.2 that the accuracy of the models generated from a single application execution per measurement point is comparable to the accuracy of the model generated from repeating measurements five times for each point, we will continue with these models for *Performance-Detective*.

Table 3.5 and Table 3.6 show the required number of measurements of each approach and the cost in total core hours for Pace3D and Kripke, respectively. The cost of individual samples are not uniform across the configuration space, as the impact the parameters have on performance can mean that some samples are orders of magnitude more expensive than others. We therefore consider the total core hours to be the more important metric. To evaluate

					cost
■(a)	6.89	9.90	12.19	15.02	6.2 %
■(b)	4.04	0.25	24.57	27.71	100.0 %
▨(c)	1.10	3.96	11.95	12.09	100.0 %
■(d)	28.38	25.81	44.40	47.13	10.2 %
▨(e)	9.28	4.81	27.84	39.44	10.2 %
■(f)	26.02	41.11	40.12	56.11	6.2 %
■(g)	37.12	49.95	52.00	64.82	41.5 %
▨(h)	31.77	45.69	45.07	59.74	41.5 %
procs	16	16	256	512	
dirsets	96	128	96	128	

■ (a) Performance-Detective (Extra-P) ■ (d) [Rit20], Perf-Taint ■ (g) [Web21], Perf-Taint
■ (b) [Cal16], Perf-Taint ▨ (e) [Rit20] ▨ (h) [Web21]
▨ (c) [Cal16] ■ (f) Performance-Detective (Decision Trees)

Figure 3.12.: Accuracy of models for the Kripke case study for excerpts of the extrapolated data set.

the predictions' accuracy, we use the same test configurations as in Section 3.4.2, inter- and extrapolating the training data points. Table 3.5 depicts the mean error of the models from the different approaches for the Pace3D case study, and Table 3.6 for the Kripke case study.

3.4.3.1. Interpolated Test Data

Figure 3.9 and Figure 3.11 show the comparison of prediction error for an excerpt of the test data among models learned using Extra-P (depicted as [Cal16] and [Rit20]) and Decision Trees (depicted as [Web21]) both with and without the information about dependencies from Perf-Taint. We compare our approach to the full-factorial setup and the sparse modeling approach,

and show configurations where the parameters identified as independent by *Performance-Detective* are varied independent of another. Table 3.5 and Table 3.6 show the mean prediction error for each model.

One would expect all approaches to be reasonably effective at predicting configuration options within the range of measurements already available. In the case of Extra-P, using the full set of measurements leads to accurate models, with model errors seldom passing 10%, and the use of Perf-Taint has a relatively small impact on results. The results of the sparse modeler are worse overall, but for Pace3D, they are overall improved by the use of Perf-Taint, decreasing the average prediction error from 15.0% to 8.9%. *Performance-Detective* achieves results comparable to the full factorial experiment design of Extra-P, while requiring only a fraction (2.9% and 6.18%, respectively) of the cost.

The performance-influence models, however, show a consistently higher model error rate of over 20%. When using *Performance-Detective*, the quality of the models remains approximately the same, while the cost is reduced by a factor of 15 for Pace3D and 7 for Kripke.

3.4.3.2. Extrapolated Test Data

Figure 3.10 and Figure 3.12 show an overview of the accuracy of the different Extra-P (depicted as [Cal16] and [Rit20]) and PIM models ([Web21]) for an excerpt of the extrapolated test data points, both using information about dependencies from Perf-Taint and without it. Table 3.5 and Table 3.6 show the mean prediction error for each model.

When extrapolating, predicting the runtime is more challenging as any errors are quickly magnified. Overall, we observe the same trends as for interpolated evaluation points, with a couple of differences. For Pace3D, the sampling approach of Extra-P generates significantly higher errors when not used in conjunction with Perf-Taint, increasing the average prediction error from 17.7% to 31.8%. However, for Kripke, the errors of the sampling approach are increased from on average 16.7% to 34.2% when used together with Perf-Taint.

The performance-influence models show a larger model error rate, this time of over 40% for Pace3D and over 30% for Kripke when using the Plackett-Burman sampling design. However, the approach for learning performance-influence

models does not extrapolate, as we can see in our data: It only predicts the time measured for the highest training value of the respective extrapolated parameter.

While providing information about the dependencies derived by Perf-Taint can be beneficial to the accuracy of the sparse Extra-P modeler, it decreases accuracy for the full-factorial experiment design. This is because with a full-factorial design, there is a lot of measurement data and Perf-Taint only removes wrong dependencies that model noise. Therefore, while the models learned without using the dependencies derived by Perf-Taint provide a better prediction, they may not be best suited for users to understand the performance behavior of an application, as they achieve these predictions by modeling measurement noise. The accuracy of the sparse modeler increases because it has less measurement data and very few information about parameter interplay.

3.4.3.3. Discussion

For both case studies, we can maintain the accuracy of the model generated with the experiment design deduced by *Performance-Detective* (25 and 5 experiment executions) as compared to a full-factorial design (625 and 125 experiment executions) and Plackett-Burman sampling (245 and 50 experiment executions). While for Pace3D, the sparse modeler is even less expensive than *Performance-Detective*, it has a lower accuracy than *Performance-Detective* (mean accuracy of 85.9% and 74.9% with and without Perf-Taint as compared to 91.5%) with especially worse predictions for extrapolated test points (mean accuracy of 17.7% and 31.8% with and without Perf-Taint compared to 9.3%).

For the PIMs, we observe a generally significant prediction error in both evaluation sets, which does not change much depending on using our minimal experiment design or sampling, and usage of Perf-Taint. This indicates that decision trees are not well-suited to model the performance of these applications.

Across evaluation scenarios, we observe that Performance-Detective *always considerably decreases the cost of experiments, while not meaningfully degrading model quality.*

3.4.4. Threats to Validity

A threat to external validity is that we ran the taint analysis on a small problem size. This was possible for the scenarios shown because the calculations are the same and still representative. However, this is not guaranteed in the general case, as sometimes different branches can be active depending on problem size. We can detect this by using control-flow tainting, provided by Perf-Taint. The only impact this scenario would have is that the taint analysis will have a higher cost.

Regarding internal validity, not inlining functions defined as inline is a potential issue. Functions are not inlined if we detect them as being performance-relevant. This could distort the measurements, as not inlining might incur a performance penalty on the applications as a whole. The overhead introduced by the profiling itself is a further source of distortion. We mitigate this by only instrumenting functions detected as performance-relevant, and keep the overhead as low as possible.

3.5. Related Work

In the following, we discuss related research in the areas of performance-influence modeling in Section 3.5.1, HPC performance modeling in Section 3.5.2, and auto-tuning in Section 3.5.3.

3.5.1. Performance-Influence Models

The difficulty of modeling modern software systems with a high degree of configurability has been addressed by Performance-Influence Models (PIMs) [141]. There, the machine learning and heuristic methods are used to iteratively learn models representing the influences and interactions of various application parameters. To overcome the difficult trade-off between measurement costs and accuracy, the models have been extended with a white-box approach [153, 154]. Unfortunately, the improved accuracy of white-box models comes with the limitation of modeling only binary and binary-encoded parameters. This limits the applicability of their approach, as for example modeling the scalability of a software is not possible. In contrast, our approach supports numerical parameters only. Function-level PIMs by Weber et al. [159]

support modeling numerical parameters, but they require expensive trace measurements. Other methods of learning performance models efficiently for highly configurable systems use Fourier transformations to reduce the number of samples and parameter combinations [72, 168]. While they rely on heuristics for sampling, we use the results obtained by the system analysis to deduce an experiment design. An extended discussion of PIMs can be found in Section 2.2.1.2 and Section 2.2.2.2.

3.5.2. HPC Performance Modeling

Analytical performance models of an application can be created manually through source code inspection and guidance by performance engineers [78, 90]. Unfortunately, such models require significant time effort and expert guidance. Furthermore, the exclusion of empirical data makes the models prone to underestimate the effects of hardware congestion and network performance.

Extra-P [32] is the state-of-the-art tool for empirical and parametric performance modeling. Extra-P supports multi-parameter modeling [29] and uses learned heuristics to reduce the experiment design [123]. The work has been extended with dedicated modeling approaches validating high-performance libraries [140] and prototyping hardware requirements for HPC applications [31]. Perf-Taint [37, 39] improves the black-box instrumentation and modeling of Extra-P with program information. We present a wider discussion of capabilities and limitations of Extra-P and Perf-Taint in Sections 2.2.1.1 and 2.2.2.1, respectively.

Hoisie et al. [148] proposed PALM, a divide-and-conquer approach for constructing performance models from annotated application source code, and enhances them with user-provided insights, program analysis, and measurement data. Vetter et al. [144] introduced Aspen, a domain-specific language for the manual specification of program operations. With an abstract machine model, the tool is capable of generating analytical models based on the provided descriptions. ASPEN has been extended with automatic performance modeling in COMPASS [101] to generate full application models statically, but it requires user intervention in the case of situations that are ambiguous for the compiler, caused by, e.g., non-scalar variables in control structures. While the user intervention is supposed to be an optional step, all benchmarks analyzed in the evaluation have been enriched with manually inserted ASPEN

annotations. In contrast, *Performance-Detective* does not require any user input and manual analysis steps.

Online learning has been used to improve the accuracy of purely static and compiler-based performance models [20, 21]. In addition, machine learning methods have been applied successfully to model performance [81, 100, 147] and to decrease the negative effects of measurement noise [124]. *Performance-Detective* provides a complete and white-box workflow that applies neither heuristics nor approximations to construct performance models, and we provide validated parametric dependencies of models.

3.5.3. Auto-Tuning

Auto-tuning methods apply an optimization method to achieve one or more goals, such as minimizing runtime [2] or floating point operations per second [152]. The scope of applying them can range from specific kernels over libraries to complete applications [13]. While online auto-tuning approaches dynamically execute the application and measure the metrics to be optimized, therefore requiring repeated runs of the software specifically for tuning purposes, offline auto-tuning approaches use data that was collected beforehand. The search space can span from different variants of low-level implementation details to achieve performance portability across different hardware, loop unrolling, and data transformations to different available solvers for a given problem [13].

To find the best configuration among the search space, global or local search methods are employed, such as genetic algorithms [14], differential evolution [86], or the Nelder-Mead algorithm [116]. Solutions are often limited to specific types of applications, system architectures, or programming language [108].

In summary, auto-tuning approaches aim at finding the best configuration for a given optimization goal. In contrast, *Performance-Detective* enables efficient modeling of the whole configuration space and interactions among configuration options. The resulting performance models can be used not only for optimizing performance, but also for exploring the performance of other configurations and guiding performance optimizations by unveiling performance bottlenecks.

3.6. Summary

We have shown that we can significantly lower the cost of automatic performance modeling of applications with multiple configuration parameters. We deduce a minimal experiment design with *Performance-Detective* by exploiting automatically derived insights about parameter interplay and main loops and thus reduce the number and cost of required measurements while achieving comparable accuracy to methods costing more than an order of magnitude more compute hours. With *Performance-Detective*, we model the Pace3D real-world multi-physics solver using 25 rather than 3125 measurements, require 34 times fewer core hours and achieve and still maintain a model accuracy of 91.5% compared to 93.8% when all measurements are used. Furthermore, we model Kripke, reducing needed measurements from 125 to 5, leading to 16 times fewer core hours needed, while maintaining an accuracy of 89.2% compared to 90.6% using all measurements.

While we validated *Performance-Detective* with software from the scientific domain, we expect that the methodology is applicable to software from other domains as well. The key requirement for applying our approach is the availability of a system analysis that can output the dependencies of configuration options on specific functions. This generality suggests that *Performance-Detective* can be applied to other types of software beyond scientific applications. However, this broader applicability still needs to be validated, and the effectiveness of *Performance-Detective* in other domains may depend on the characteristics and nature of the system being analyzed: While scientific software usually has a main loop driving the computation, this is not necessarily the case in other domains, potentially lowering the possible savings in measurement cost. Moreover, the assumption imposed by Perf-Taint that the performance-relevant behavior of a system is located in computational loops and MPI communication routines may not hold true for software from other domains. However, we could mitigate that by using another system analysis that can take other performance-relevant behavior into regard.

4. Identification of Performance-Irrelevant Parameters

Performance models for configurable software show the influence of configu-
ration options on software performance. However, as discussed in Section 2.1,
not all configuration options are performance-relevant. Therefore, the chal-
lenge to select which parameters to include in the performance modeling
process arises. While for some parameters, such as the problem size, it is
usually clear that they are performance-relevant, others are more difficult
to categorize: The impact of each option can vary significantly depending
on the scenario computed, and the interplay between options can create
complex dependencies. Tools for system analysis such as Perf-Taint [37], as
presented in Section 2.2.2.1 and utilized in Chapter 3, and Comprex [154] ex-
tract performance influences of options based on static and dynamic analysis
automatically but cannot quantify their influence. As *Performance-Detective*
deduces the experiment design based on the results of the system analysis,
pruning of experiments is only possible if options have a linear performance

impact or no impact at all [134]. In other cases, it is necessary to rely on heuristics to reduce the experiments, potentially compromising model accuracy, or to use an expensive full-factorial experiment design – taking 368 core hours for an application with only three options already, as we showed in Section 3.4.2.

To further guide domain scientists with selecting parameters for performance modeling, we introduce a novel approach to ease the modeling process by introducing a pre-processing step that automatically determines performance-irrelevant configuration options and removes them from the remaining modeling process. This chapter thus describes our contribution C2.

As previously in Chapter 3, we focus on modeling systems that operate as batch jobs and run without the possibility for user intervention during their execution.

4.1.   Limitations of Existing Modeling Frameworks

As presented in Section 2.2, there are many different black-box and white-box modeling workflows for performance modeling of configurable software. Black-box approaches treat the application as black-box, only learning the performance models from the correlation between measurement data and configurations executed. White-Box approaches leverage knowledge about the application and how the configuration options impact performance for performance modeling.

Figure 4.1 shows the modeling workflows of black-box performance-influence models and *Performance-Detective* for an exemplary configuration space with parameters *param*1 to *paramN*, with some parameters being performance-relevant and others performance-irrelevant (cf. Section 2.1). Both modeling workflows have to consider all options selected during the system analysis for their experiment design. For black-box modeling, an expert selects these options based on intuition and their understanding of the software [67]. However, this can lead to selecting more parameters than necessary [89] for modeling, as illustrated in the example: The expert selected not only the performance-relevant parameters *param*1 and *param*3 for modeling, but also *param*2, *param*7, and *paramN*, necessitating sampling over all of them to detect potential performance influences, even though the final model does not contain the performance-irrelevant parameters.
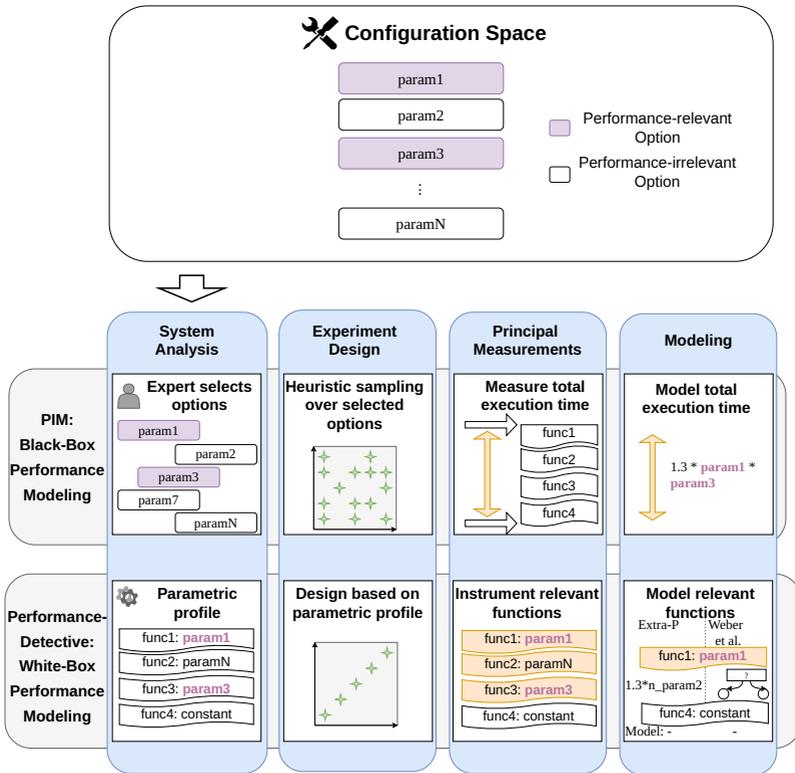
Figure 4.1.: Modeling workflows for black-box *Performance-Influence Models* (PIMs) and white-box models using *Performance-Detective*.

Tools such as Perf-Taint [37] and Comprex [154] extract possible performance impacts of options automatically based on static and dynamic analysis in white-box modeling. However, these analyses cannot quantify the performance impact of configuration options. An option influencing only small parts of the execution could have a critical impact, while it could also just change minor things. In the latter case, again, samples not necessary to model the system performance have to be taken.

We give two examples that are inspired by real-world occurrences in the multi-physics solver Pace3D. The first example is shown in Listing 4.1: The configuration option y influences the control-flow of the program by determining the result that is returned by the function branch. However, the only change in computation is a single addition, making the expected performance impact of y minimal. This can occur, for example, when physical constants are added or changed, depending on another configuration option y. Listing 4.2 shows another example: While the variable global_offset does impact the number of iterations via the local_offset variable in the loop, it only changes the iteration count by one. Scenarios like this can occur when the local area of material that each process gets contains overlaps with other processes, which should or should not be taken into account for the computation. On the other hand, x directly impacts the number of iterations. Given a high value for x, we can assume that global_offset does not significantly impact the performance. Therefore, we do not have to model the impact of the global_offset configuration option on performance. Nevertheless, state-of-the-art tooling does not provide a way to gain and utilize this knowledge.

As the tooling does not provide a strategy to guide the user in selecting the parameters that significantly impact performance, users have to collect samples that consider all configuration options, resulting in high costs for performance modeling.

```
int branch(int result, bool y) {
    if (y == true) {
        return result + 1;
    }
    else {
        return result;
    }
}
```

Listing 4.1: Example computation using the configuration option y: While y influences the control-flow of the program, its performance impact is expected to be minimal.

```
void calculateAll(int global_offset, int x) {
    int local_offset;

    if (global_offset % 2 == 0) {
        local_offset = 0;
    } else {
        local_offset = 1;
    }

    for (int i = local_offset; i < x; i++) {
        calculate();
    }
}
```

Listing 4.2: Example computation using configuration options global_offset and x: While x directly influences the amount of loop iterations, global_offset changes the amount of loop iterations by one only.

4.2. Approach

The general idea for improving the empirical performance modeling processes for high-performance software applications is to use small-scale experiments in order to identify performance-irrelevant configuration options. We therefore improve the parameter selection for the system analysis phase with our approach.

Figure 4.2 shows our addition to the performance modeling process: Instead of directly starting the modeling workflow, we first employ our optimization process that identifies and removes performance-irrelevant configuration options. After that, the usual modeling workflow can be employed with a reduced parameter set, starting with further system analysis for white-box approaches to trace the influences of options to functions more precisely or directly with the experiment design for black-box approaches.

Our optimization process involves three additional steps: Considering all available configuration options, we first collect samples in a cheap way by conducting small-scale experiments. We then build a preliminary performance prediction model from the collected samples using a preexisting performance

modeling method. Based on this performance model, we can classify all configuration options as either *performance-relevant* or *performance-irrelevant*. We create the filtered configuration space containing only performance-relevant configuration options based on this classification. This filtered configuration space then serves as input configuration space for the further modeling process.

Our approach allows users to select options from a reduced set of only performance-relevant configuration options without losing predictive power in the resulting performance model. We identify two key benefits: First, knowledge about the internal structure of the examined application or expert knowledge about performance engineering is not required anymore, as our approach classifies the options into performance-relevant and -irrelevant. Second, in addition to saving a significant amount of time by modeling fewer parameters during the actual performance modeling, we argue that we can execute the parameter identification experiments on cheaper compute infrastructures, such as consumer desktop computers or workstations, even if the final performance model utilizes high-performance systems for measurement acquisition. We hereby build on results of previous studies [82] that have shown that if a configuration option or interaction between configuration options is measured to have an influence on performance on one hardware, this property is typically preserved across differing environments.

In the following, we elaborate on our approach in detail: First, Section 4.2.1 explains how we keep the sampling process for our small-scale experiments cheap yet extensive enough to collect meaningful samples. Section 4.2.2 details our requirements for a performance modeling method used to identify the performance-irrelevant options based on the collected samples. Finally, Section 4.2.3 presents how configuration options can be filtered based on the preliminary performance model.

4.2.1. Small-Scale Measurements

As our identification process adds three pre-processing steps to the performance modeling pipeline, we must ensure that it decreases cost in the later stages of performance modeling, outweighing the additional cost incurred. To reduce the number of measurements for the optimization step, we only measure two different values for numeric configuration options because, for the identification of relevant options, we only need to detect a leap in the
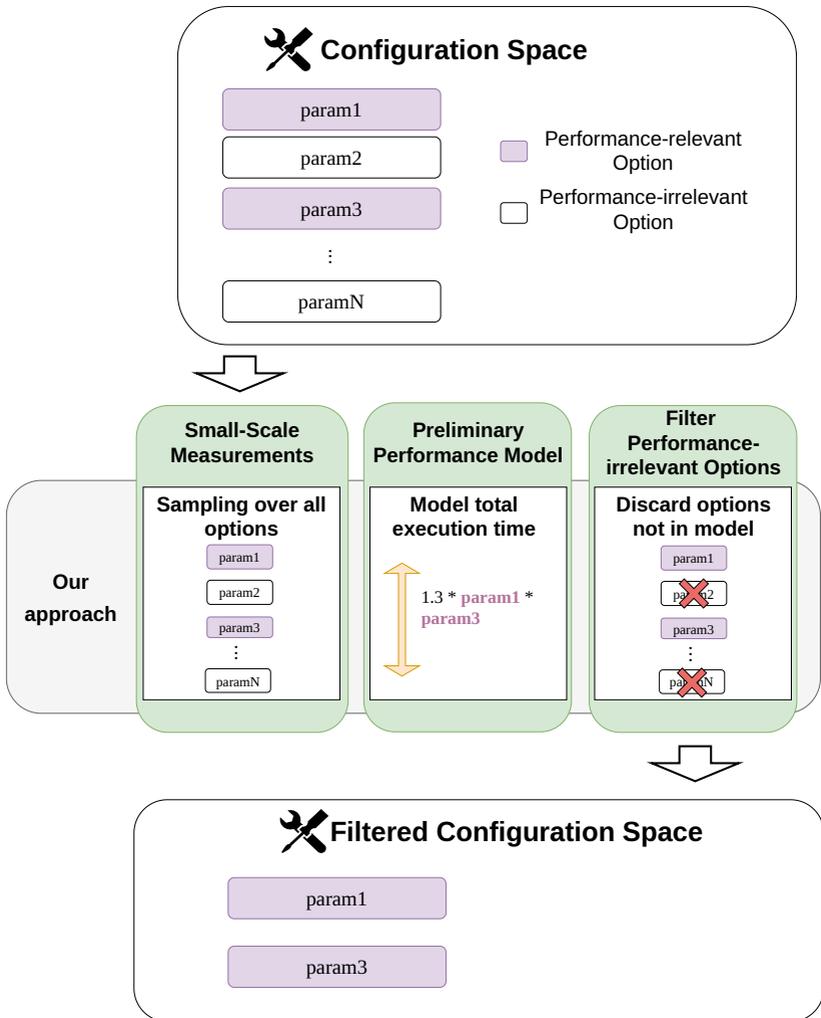
Figure 4.2.: Our pre-processing determines performance-irrelevant configuration options from the modeling process, enabling the exclusion of them in the further modeling process.

runtime when changing option values. For non-binary and non-numeric configuration options, such as selection options, however, we have to analyze every possible configuration value as we cannot assume a (partial) order.

However, not only the number of samples but also their cost is important. To keep the cost of the individual samples low, we use small, yet realistic problem sizes and value ranges of configuration options. It is the responsibility of the user to select these values carefully using domain knowledge, ensuring that the computation to be modeled is still representative of the computation at larger scale. Note that we do not require the user of our approach to have knowledge about the internals of the application or expertise in performance engineering. However, we do assume them to be familiar with the domain of the application. That means that they can configure the application according to the problem they want to compute using functional options.

4.2.2. Create Preliminary Performance Model

Our proposed process can leverage any performance modeling method. The sole prerequisite is the availability of an empirical performance model derived from runtime measurements. Ideally, it should swiftly produce preliminary models that offer a realistic representation of application performance based on small-scale measurements. As our process is a pre-processing step, we prefer black-box models learned from measuring the end-to-end runtime of the application, as white-box models include instrumentation overhead.

We select DECART [70], as introduced in Section 2.2.1.3, as the exemplary performance modeling method to build the preliminary performance models and illustrate our approach. We choose DECART as it promises to build performance models from random samples with the number of samples only linear in the number of options. The resulting performance models have a decent prediction accuracy in the 90% range [70] and take only a few seconds to learn.

4.2.3. Filter Performance-Irrelevant Options

In this last step, we examine the created performance model regarding the options it uses for modeling the system performance and creating its performance prediction. In the case of DECART, the models contain a list of configuration options used within the model. Inherently, only the configuration options integrated into the models can impact its performance prediction.

Therefore, we classify every option in that list as performance-relevant and all other options, which do not appear in the model, as performance-irrelevant.

4.2.4. Assumptions and Limitations

Our approach relies on several assumptions and has certain limitations. We aim at detecting the performance influence of an option with a limited set of training measurements. Thereby, we risk to not capture all possible scenarios in which an interaction between options may lead to a performance impact of one of them, particularly those involving rare or corner-case interactions. This is a principal limitation of all black-box modeling approaches. Systematically exploring these interactions and their performance impact for a given system will require further research and could integrate other techniques like SafeTune [76] that aim at extracting performance influences from software documentation.

Moreover, for the benefit of using cheaper compute infrastructures for the parameter identification experiments, we rely on the assumption that the impact of configuration options on performance remains consistent across different environments, as shown by Jamshidi et al. [82]. While they use four subject systems from different domains for their study, none of them is a scientific application and thus may exhibit different characteristics. Therefore, the applicability of the findings from the study has yet to be validated for scientific software, e.g., by repeating their study with subject systems from this domain.

Additionally, we assume that insights on the performance-irrelevance of options gathered from small-scale experiments are representative of large-scale computations. We thereby rely on the domain knowledge of users of our approach and their ability to downscale computations.

4.3. Case Study

We illustrate our approach presented in Section 4.2 with a real-world case study based on Pace3D [80], a multi-physics framework for digital material research that we also used for the evaluation of *Performance-Detective* in Chapter 3. Pace3D is highly configurable, offering more than 170 tools for

pre- and post-processing of computations alone. Consequently, the flexibility offered by the software system introduces many configuration options. This makes it challenging for the domain scientists using the software to understand the performance impacts of the many options and consequently choose a configuration that will lead to good performance. However, the immense number of configuration options and their interactions make building performance models for the whole application with current approaches impractical due to the high costs associated.

For our case study, we set fixed values for options influencing the physics of the material to simulate to keep the workload and resulting computations constant, and consider only non-functional options that do not change the final result of the simulation. It is common practice to consider the values of some configuration options as fixed [154], with mostly options influencing the workload being considered fixed. This is because modeling them not only requires significantly more measurements, but also a deep knowledge of the domain in order to characterize them [111]: For example, one property of a material may be described by several options, making modeling them in isolation less meaningful. Moreover, changing this property may result in performance changes that can not necessarily be modeled mathematically [111].

We consider 34 non-functional configuration options that can be set for the chosen computation scenario, a phase-field computation. These options consist of 9 binary options, two binary vector options with two elements each, three selection options, 19 numeric options, and one numeric vector with three elements. The reduced set includes, among others, the simulation volume, number of preprocessing steps, simulation coefficients, number of time steps, random generator settings, numeric scaling factors, and the number of MPI processes.

4.3.1. Small-Scale Measurements

As DECART only supports binary options, we map every non-binary option to a binary representation. We employ two predefined values (low/high) for numeric configuration options to reduce the number of required measurements. For example, we measure only the values 10 and 100 for the `writeTimesteps` option instead of multiple more values in-between, mapping the numeric `writeTimesteps` option to two binary options `writeTimesteps_100` and `writeTimesteps_1000`. Thus, the 34 considered options are represented as

66 binary options. DECART uses a feature-size heuristic to prescribe the number of required samples. Therefore, our simple model will have $N = 66$ options. We generate samples randomly.

As mentioned in Section 2.2.1.3, the DECART modeling process is iterative. This means that the sampling and modeling process should be repeated until the learned model has a validation error below 10%. However, this recommendation is given for models to reach high accuracy for being able to predict the performance of the whole configuration space of the software. In contrast, we want to identify performance-irrelevant options. As we do not know how many samples and resulting model prediction accuracy we need to identify performance-irrelevant options, we repeat the sampling process ten times. With these measurements, we can create models with an increasing number of samples from N to $10N$ that we can evaluate separately.

Performing Measurements. We run our experiments on an on-premise cluster with nodes with an AMD Opteron 2378 8-Core processor @ 2.4 GHz and 16 GB memory. This cluster is regularly used for simulation runs of Pace3D, thus a realistic execution environment for such a simulation. We repeat each measurement five times, observing a mean coefficient of variation of 4.84% between repetitions of the same configuration.

4.3.2. Create Preliminary Performance Model

In the small-scale measurement step, we repeated the sampling process ten times (see Section 4.3.1). Using these small-scale measurements, we create ten inputs for DECART, each with 66 samples more than the one before. We supply each measurement individually to DECART, meaning that the first input file contains $66 * 5 = 330$ measurements, as we repeated each measurement five times. We choose 10-fold-cross-validation as resampling and grid search as a parameter optimization algorithm as these values proved best [70].

DECART generates multiple performance models for every input with an increasing number of samples used in the training set, requiring the user to review and select the models to accept. Guo et al. [70] recommend selecting a model with a validation error below 10%. If there is no such model, the sample size should be increased, and new sample measurements added. However, as

Model No.	No. of Experiments	Validation Error (%)	Generalization Error (%)	Model Time (s)
1	330	6.17	6.43	5.26
2	660	7.21	6.67	7.13
3	990	6.34	8.05	8.74
4	1320	8.06	8.93	10.10
5	1650	8.31	7.80	13.62
6	1980	8.23	9.53	12.14
7	2310	9.09	9.87	12.43
8	2640	8.63	9.35	15.24
9	2970	9.11	9.95	14.83
10	3300	7.82	10.51	13.86

Table 4.1.: Our preliminary performance models for PACE3D, each with an increasing number of experiments used. Validation and generalization error are calculated based on the respectively supplied experiments.

our identification process is only a preprocessing step, we will still proceed with a model that has a higher validation error.

Table 4.1 shows an overview of our models. It lists the number of measurements used for creating each model, the time for creating it, and validation and generalization errors. While validation and generalization errors are below 10% for nine out of ten models, with only the generalization error of model ten being slightly above 10%, their explanatory power is inherently limited to the number of experiments conducted for the respective model.

4.4. Evaluation

We evaluate our approach based on the case study from Section 4.3. To evaluate our approach with the previously detailed case study, we assess if our approach accurately classifies options as performance-irrelevant. Second, we evaluate if our approach can save costs for creating an exhaustive performance model despite the additional pre-processing steps introduced. We formulate the following sub-research questions to RQ 2 as stated in Section 1.3:

RQ2 How can performance-irrelevant configuration options be identified automatically?

    RQ2.1 Can we identify performance-irrelevant configuration options by reusing a performance modeling method with small-scale samples?

    RQ2.2 Can we reduce the cost of performance modeling by introducing our identification process of performance-irrelevant configuration options as a pre-processing step?

We address these questions by first evaluating if our approach can accurately classify configuration options as performance-irrelevant. Second, we evaluate if applying our approach can save costs during creation of the principal performance model, despite the additional pre-processing step introduced.

Option Name	Samples Measured During Pre-Processing										Runtime Deviation
	66	132	198	264	330	396	462	528	594	660	
Φ dynamic memory	✓	–	–	–	–	–	–	–	–	–	0.46%
Φ block data	✓	–	–	–	–	–	–	–	–	–	0.47%
Φ avg RDTIC	–	✓	–	–	–	–	–	–	–	–	2.39%
Φ driving force nl.	–	–	–	–	–	✓	–	–	–	–	2.33%
Φ # Active Phases	✓	–	–	–	–	–	–	–	–	–	0.60%
Φ phi index	✓	–	–	–	–	–	–	–	–	–	0.60%
Φ avg driving force	✓	–	–	–	✓	–	–	–	–	–	3.71%
Φ # LROPB locksize	–	✓	✓	–	–	–	–	–	–	–	2.03%
prec. smear iterations	✓	–	–	–	–	–	–	–	–	–	4.88%
IO buffer scale factor	✓	–	–	–	–	–	–	–	–	–	0.12%
RNG manual seed	✓	–	–	–	–	–	–	–	–	–	2.29%
SI scal. factor ampere	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.48%
SI scal. factor candela	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.02%
SI scal. factor kelvin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.21%
SI scal. factor kg	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.08%
SI scal. factor meter	✓	✓	✓	✓	✓	✓	✓	–	✓	✓	0.22%
SI scal. factor mol	✓	✓	✓	–	✓	✓	✓	–	✓	–	0.89%
SI scal. factor second	✓	✓	✓	✓	✓	✓	✓	✓	–	✓	1.74%
control output	✓	–	–	–	–	–	–	–	–	–	1.89%

Table 4.2.: Identified irrelevant options and the runtime deviation (%) when changing them. Checkmarks indicate that the respective option is identified as irrelevant, while Dashes indicate that the respective option is identified as relevant.

Option Name	Samples Measured During Pre-Processing										Expert Assessment
	66	132	198	264	330	396	462	528	594	660	
Φ dynamic memory	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	Major
Φ block data	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	Minor
Φ avg RDTIC	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	Minor
Φ driving force nl.	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	Minor
Φ # Active Phases	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	Minor
Φ phi index	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	Minor
Φ avg driving force	×	✓	✓	✓	×	✓	✓	✓	✓	✓	Minor
Φ # LROPB locksize	✓	×	×	✓	✓	✓	✓	✓	✓	✓	Minor
prec. smear iterations	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	Minor
IO buffer scale factor	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	Minor
RNG manual seed	✓	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	Irrelevant
SI scal. factor ampere	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Irrelevant
SI scal. factor candela	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Irrelevant
SI scal. factor kelvin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Irrelevant
SI scal. factor kg	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Irrelevant
SI scal. factor meter	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Irrelevant
SI scal. factor mol	✓	⊗	✓	⊗	⊗	⊗	✓	⊗	✓	⊗	Irrelevant
SI scal. factor second	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Irrelevant
control output	×	✓	✓	×	✓	✓	✓	×	✓	✓	Major/Minor

Table 4.3.: Expert assessment of the identified irrelevant options: Assessments matching the models' classification are marked with a Checkmark (✓), those not matching with a cross (×) if relevant according to the expert and with a circled cross (⊗) if irrelevant according to the expert.

4.4.1. RQ2.1: Accurate Identification

We created ten performance prediction models with increasing sampling sizes (see Section 4.3.2) and analyzed the models by inspecting which configuration options they use. Table 4.2 and Table 4.3 show the results: Our model analysis classifies six to sixteen options out of 34 as irrelevant to the performance. We evaluate whether this classification is correct by first measuring the relative runtime deviation when changing the considered option. Second, we interview the lead developer of Pace3D to state his assessment on the performance relevance of all considered options.

4.4.1.1. Measuring Performance Impact

To measure the performance impact of options identified as irrelevant, we measure the relative runtime deviation when changing the considered option. All other options are activated or set to a default value. We repeat each measurement five times.

Table 4.2 shows our results, indicating that the identified options are indeed performance-irrelevant. We show the deviation of runtime in percent when changing the respective option. Cells with dashes indicate that the respective option has been identified as relevant by the model. The biggest deviation caused by one of the performance-irrelevant options in runtime is 5.1%. In contrast, options that strongly impact performance often cause a performance difference of over 100%. Notably, some options are identified as performance-irrelevant by all models, while others are flagged by only some or even only one model.

Model one, which uses the smallest number of samples, identifies most options as performance-irrelevant. While this could also be due to the samples not catching the performance impact of some of the options, our evaluation shows that only one of the options, `precond.smear.iterations`, could be judged as performance-relevant with an impact on the runtime of 5.12%.

4.4.1.2. Developer Statement

We asked one of the main developers of Pace3D who has been working on the software for a long time to fill out a questionnaire, indicating if they

think that a specific configuration option has no, minor, or a major impact on performance. We also gave the possibility to indicate that they were unsure about the impact of the option.

Table 4.3 shows how our expert classified the options and what options the respective model classified as irrelevant. The expert identified eleven options to be performance-irrelevant. Six of them were classified as irrelevant by all models. Model one further identified `RNG.ManualSeed` and model one, three, seven, and nine identified `Settings.SIscalingfactor.mol` as being performance-irrelevant, while the other three (`FunctionH`, `Phasefield.Eps`, and `Settings.RandomGenerator.Type`) were not identified as irrelevant by any model.

The expert further identified nine options as having a major performance impact. Seven of them were classified as relevant by all models, with the other two being misclassified by model one (`DynamicMemory`) and model one, four, and eight (`ControlOutput`), respectively. However, he further stated that `ControlOutput` would have a major impact only for configurations that have a short runtime which will become minor for long-running simulations. As simulations used to simulate real-world scenarios are usually longer-running, we deem this misclassification to be tolerable. Ten other options identified as irrelevant by some models are classified as having a minor impact on performance by our expert. Five of them were misclassified exclusively by model one.

Overall, model one has the most misclassifications according to the expert statement. This is to be expected as it had the fewest training data.

Few models misclassified options with a major performance impact as being performance-irrelevant. Moreover, up to including model eight, some options are being classified as irrelevant while having a minor impact on the performance according to our expert.

4.4.1.3. Discussion

The results in Section 4.4.1.1 imply that using model one, which is the cheapest preliminary performance model, is already sufficient for pruning performance-irrelevant configuration options. However, according to our expert statement presented in Section 4.4.1.2, it misclassifies some options with a minor performance impact as being irrelevant. Crucially, it misclassifies an option that

has a major impact on performance according to our expert. As model one uses only a very limited number of samples, it likely did not see enough data to capture their performance influence.

Models nine and ten have the least misclassifications according to our measurements and the expert statement: They identify seven and six, respectively, out of eleven performance-irrelevant options. Moreover, the do not misclassify any performance-relevant options as irrelevant, which would compromise the accuracy of the principal performance model, as critical performance influences could not be modeled.

No model identified all eleven performance-irrelevant options pointed out by the expert. While not identifying all performance-irrelevant reduces potential savings by not being able to exclude them from the principal modeling, it does not impact the quality of the principal performance model. Moreover, as only model one classified an option having a major impact also in large-scale settings as irrelevant, we can conclude that *our approach can save costs while not compromising model quality*.

However, as we only evaluate our approach with one case study, we can not give a general recommendation on how many samples to use based on our data. For this, further experimentation is needed.

4.4.2. RQ2.2: Cost of Performance Modeling

We estimate the costs saved for creating principal performance models using black-box and white-box approaches separately, as their strategies for selecting samples differ and are, subsequently, affected in different ways by filtering performance-irrelevant options.

4.4.2.1. Black-Box Modeling

Black-box performance modeling techniques do not require a set number of samples, but instead rely on heuristics to estimate the required number of samples. This results in a hard-to-quantify trade-off between number of samples and model accuracy [66, 94]. Some techniques iteratively use more samples, as proposed also by DECART [70]: In each step, a model is trained with part of the data and validated with the rest. If the error

exceeds a given threshold, more samples need to be added to the training data. Thus, it is hard to quantify the costs saved by our approach. However, it will still prove helpful to prune options from the configuration space, as we can ensure that the remaining options have a performance impact that is worth modeling. Samples that only vary configuration options not relevant to performance cannot give insight into the performance behavior of the system. By eliminating such samples and collecting meaningful samples that vary options relevant to performance instead, we expect the model to converge faster. In other words, the same amount of samples will result in better model quality.

4.4.2.2. White-Box Modeling

White-Box Performance-Influence Models [153, 154] are limited to modeling only binary and binary-encoded parameters. However, numerical parameters, such as the number of processes or the problem size, are common in scientific applications and would be tedious to encode.

The white-box modeling workflow of Extra-P with Perf-Taint [37] requires a full-factorial experiment design with 5 measured values per configuration option. *Performance-Detective* can only prune configurations to measure if options do not interact at all or if they linearly affect system runtime [134]. However, *Performance-Detective* cannot quantify the impact of a configuration option, requiring the user to either model all configuration options or exclude them solely based on the number of functions they are influencing, not their actual impact. Moreover, the modeling of Extra-P does not support binary configuration options. In order to capture the influence of a binary configuration option, we would thus need to create two separate performance models and compare them to assess the performance impact of the binary option. Capturing performance impacts of all binary options would thus mean creating performance models for any combination of binary configuration options.

In our case, model 10 identified six numerical configuration options as performance-irrelevant. Following a full-factorial experiment design, measuring these six options would require 78125 experiments. For each performance model, *we save 78125 experiments by pruning the six numerical configuration options identified by model 10, which translates to 30482 core hours*, assuming an average runtime of 0.39 core hours we observed during our small-scale

sampling. This estimate does not even consider that users would likely opt for measuring larger problem sizes and value ranges during the principal measurements. Gathering all of our samples took 1288 core hours.

4.4.3. Threats to Validity

After discussing our findings, we discuss threats to validity regarding the internal and external validity of our evaluation.

Internal Validity. Our evaluation indicates that the identified configuration options in later models indeed do not have a performance impact. However, the potential impact of filtering them on the prediction accuracy of resulting performance models remains uncertain. Nevertheless, the consistency observed in the measured scenarios and the expert statement reinforces our confidence in the overall findings.

External Validity. Our evaluation is currently limited to a single case study application. While our selected application is a real-world HPC application, an assessment of the generalizability of the approach requires further research. However, we are confident in the generalizability of our approach, as we designed it to be applicable to any batch application.

4.5. Related Work

In the following, we discuss related research regarding performance modeling of configurable systems in Section 4.5.1, experiment design strategies in Section 4.5.2, and selection of performance-relevant parameters in Section 4.5.3.

4.5.1. Performance Modeling of Configurable Software

There exist many approaches for creating performance models of configurable software [29, 94, 114, 141]. CoMSA [166] is an iterative modeling approach where the sample selection process and the performance model

influence each other: By explicitly modeling the uncertainty in the prediction of configurations, the next sample to take is determined by the configuration with the highest uncertainty. Instead of reducing the amount of samples to take, they focus on achieving better model quality with the same number of samples. Ha et al. [71] create models using deep neural networks. Shu et al. [139] present PERF-AL that uses neural networks with adversarial learning. Other approaches [72, 168] use Fourier transformations to reduce the number of samples and parameter combinations, but are limited to binary configuration options. Han et al. [74] focus on finding performance bugs by ranking configuration options regarding their performance impact. They all require sampling across the whole configuration space and do not consider systematically decreasing the configuration space before creating the experiment design by filtering performance-irrelevant options. Our approach is orthogonal and can be used as pre-processing to any performance modeling approach for creating the principal performance model.

4.5.2. Strategies for Experiment Design

Sarkar et al. [129] propose heuristic strategies for the cost-effective sampling of configurations but do not consider reducing the configuration space by pruning options. Velez et al. [154] use program analysis to identify which configuration option influences control-flow statements in a code region. They use this knowledge to select option values for configurations that allow the exploration of all paths. However, this considers only binary configuration options. Nair et al. [112] use dimensionality reduction to reduce the configuration space and the number of required performance measurements. This approach assumes that all the options are equally important and only works for numeric options. With *Performance-Detective* (cf. Chapter 3), we reduce the number of required experiments by using parameter interaction knowledge gained through a taint analysis. This approach considers only numeric configuration options. In contrast, our approach in this chapter works for any option type. Dominguez-Trujillo et al. [45] aim at reducing the amount of data needed to analyze performance variation in HPC applications (as caused by, e.g., operating system management activites or inconsistent system cooling patterns) by focusing on maxima distributions.

4.5.3. Selection of Performance-Relevant Parameters

SafeTune [76] extracts the potential performance relevance of parameters from software documentation, hence requiring the presence of extensive and accurate documentation. DiagConfig [36] uses taint tracking to compute the so-called performance property of an option by counting the number of performance-related operations. They train a Random Forest model to predict the performance-relevance of options with labeled data from systems, and then apply the model to predict the performance-relevance of options in other systems. This limits the applicability of the approach to domains within the training systems. Moreover, relying on counting performance-related operations causes misclassifications, as a count is not necessarily an accurate reflection of runtime performance behavior. Cao et al. [33] identify the three parameters most important to the performance of storage systems to optimize performance tuning. They do not aim at identifying all performance-relevant parameters and only consider the amount of samples required and not their cost. Kanellis et al. [89] studied how many parameters need to be tuned to achieve well-performing configurations in database systems. While they show that only some parameters are important to performance and can thereby accelerate auto-tuning, they manually pre-selected the tested parameters they believed to be performance-relevant out of all parameters. Moreover, they do not propose an approach as to how to identify the performance-relevant parameters at a low cost out of the complete configuration space.

4.6. Summary

In this chapter, we presented an approach to systematically identify performance-irrelevant configuration options. By excluding these options from the performance modeling process, costs for constructing performance models can be reduced. Domain scientists can use our approach to filter performance-irrelevant options specific to a certain problem they would like to simulate. Our approach is easily applicable by domain scientists, as users are neither required to have knowledge about the internal structure of the application nor performance engineering expertise.

Our evaluation shows that we can identify performance-irrelevant options while not incorrectly filtering performance-relevant options. The cost saved

by filtering the performance-irrelevant options from the principal modeling is expected to be much higher than the cost incurred by the additional small-scale measurements needed for our approach.

While we validated our approach with an application from the scientific domain, we believe that our methodology can be applied to software from other domains as well. However, the effort required to create a detailed performance model may not always be justified. Scientific software is typically long-running, meaning that optimizations can lead to significant performance gains over time. In other domains, where applications may have shorter runtimes or less sensitivity to configuration, the benefits of performance modeling may not outweigh the costs of building such a model.

5. Benchmarking of Serverless Workflows

The emergence of workflows led to significant research activity, with many examples of optimizations and new systems developed to improve the performance and efficiency of stateless and stateful workflows. We examined 72 different research contributions to determine the similarity of their evaluation baselines and present the results in Table 5.1. We found that publications use different applications to benchmark the performance of new ideas, do not cover the same classes of workloads, and do not always compare against the same subset of platforms. Without a consistent baseline, comparing research results and establishing the most promising ideas becomes impossible. While established benchmark suites exist for CPU [145], databases [149], and specific workloads like machine learning [18] and microservices [55], serverless is a relatively new paradigm. Benchmarking suites and systems have been proposed for FaaS [40, 91, 104, 143], but as of today, a benchmarking suite

Papers	Total	Benchmarks						Platforms					Artifact?
		Micro	WebApp	Multimedia	Data Proc.	ML	Scientific	AWS	Azure	GCP	Other	Research	
Analysis	14	7	1	4	2	4	2	8	4	3	3	3	5
Optimization	17	8	3	4	4	5	6	9	0	2	2	7	4
Application	18	1	4	1	4	1	7	15	5	5	2	3	9
Prog. Model	23	10	6	5	8	11	8	10	3	1	2	16	11

Table 5.1.: Analysis of 72 research papers on serverless workflows with benchmarks. For each category, we show how many papers use which types of benchmarks and platforms. Note that one paper can use multiple benchmarks and platforms.

for serverless workflows has remained an open problem. A comprehensive, consistent, platform-independent, and portable benchmarking suite will support the ongoing research work [113, 131], enable software developers to differentiate between alternative solutions, and determine the future road to improved workflows.

We propose the first serverless workflows benchmarking suite to support developers in choosing a platform as well as the quickly growing research activity in serverless workflows. Our work provides a baseline and benchmarking methodology that can be used to evaluate and compare the optimizations and systems developed to improve the performance and efficiency of workflows.

As highlighted in Section 2.5, serverless workflows platforms differ in their programming models and implementation. We therefore begin by establishing a unified and portable workflow model to abstract away the differences between different commercial, open-source, and research platforms (Section 5.1). We base our formal definition on Petri Nets [115] since they highlight control-flow dependencies and have already defined semantics. We extend Petri Nets with components needed to represent the full spectrum of workflows. With a cloud-agnostic workflow definition, we design the benchmarking suite (Section 5.2) and implement it on top of SeBS, an established serverless benchmarking suite [40]. Then, we implement six workflow benchmarks based on solutions common in research and industry (Section 5.3). Selected benchmarks represent different computational patterns and cloud services, covering machine learning, multimedia processing, and scientific applications.

Applications are implemented in our unified workflow model, providing an identical benchmark structure for each platform. We evaluate *SeBS-Flow* in Section 5.4 by evaluating the expressiveness of our model and the overhead of transcribing it to the native platform representations.

Then, we comprehensively evaluate the three major cloud workflow services (Section 5.5) using *SeBS-Flow*. We measure the performance of each application and analyze the potential overhead sources. We examine the cost difference between platforms and investigate cold startups and parallel scalability as well as suitability for scientific workflows. Finally, we compare the results obtained across several months to analyze long-term performance stability trends of serverless workflows in commercial settings. We follow the FAIR principle [164] and release our benchmark suite on an open-source license. Our experiments can be repeated automatically in the cloud, allowing for reproducible results and measuring performance changes in clouds over time.

This chapter thus constitutes our contribution C 3, *SeBS-Flow*, composed of the following contributions:

- We introduce a platform-agnostic workflow definition to model control- and data-flow, automatically transcribe the application into a cloud's proprietary presentations, and enable developers to run near identical workloads on different systems.

- We propose a benchmark suite with six real-world application benchmarks and four microbenchmarks.

- We evaluate the expressiveness and overhead of our model by reviewing the literature on serverless workflows.

- We extensively analyze performance, cost, scaling, and stability of our benchmarks by deploying them to three major cloud platforms.

A first version of the serverless workflows benchmarking suite has been described in a Master thesis [24]. We extend this version by expanding and formalizing the proposed workflow model, extending and improving the implementation, collecting measurement data, conducting a literature study to evaluate the workflow model, and publish the resulting version of *SeBS-Flow* [133]. For reasons of readability, we only describe *SeBS-Flow* in this work and shortly outline relevant differences to prior work in Sections 5.2

(Implementation of Workflows Benchmark Suite) and 5.3 (Benchmark Applications).

5.1. Serverless Workflows Model

We define a model for serverless workflows that allows us to implement and analyze a workflow application independent of the platform it will run on. The model should encode the control flow and task parallelism. Furthermore, it should clearly display the flow of data between functions, which aids the user in detecting scalability bottlenecks and errors, such as inconsistent or missing data. Therefore, we opt for defining our model on top of workflow nets with data (WFD-nets) [150] (see Section 2.6) and extend them to be able to express the orchestration by the platform and how data is passed between functions.

While basing the model on WFD-nets is only one possibility among alternatives such as state machines, we opt for WFD-nets due to their advantages as modeling formalism, such as their graphical nature, formal semantics, and analysis defined. WFD-nets already have different analyses defined on them related to control-flow correctness and verification of the flow of data through the workflow. This allows analyzing, e.g., whether a workflow terminates, whether a specific task will ever be executed, whether data to be read is missing, and whether data is strongly redundant, i.e., never read after it is written. Analyzing this aids users in the process of defining correct workflows. However, serverless workflows are orchestrated by the platforms which impose time limits on execution and schedule tasks. Moreover, it is important to model how in- and output data is passed between tasks. Modeling both of this is not supported by WFD-nets currently.

5.1.1. Transitions

The set of transitions T is composed of two types, the coordinators C and serverless functions SF, $T = C \cup SF$. A *function* transition $sf \in SF$ represents the execution of a serverless function. All function transitions that can run in parallel without any precedence dependencies and their immediate predecessor and successor places make up a workflow phase S. There are different possible token routing constructs within one phase of the workflow:
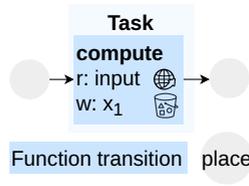
Figure 5.1.: *Task* function transition.
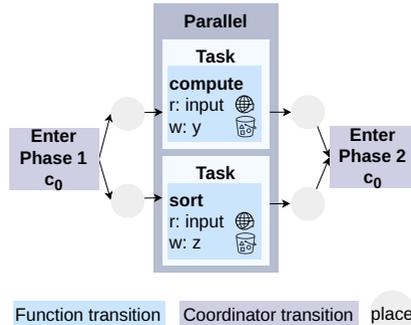
Figure 5.2.: *Parallel* function transition.

Sequential Routing. A *task* phase is a sequential routing consisting of one function transition only. Figure 5.1 shows an example: Once a token is in its input place, the task transition will fire, executing the *generate* function.

Parallel Routing. For parallel routing, there are two alternatives: First, a *parallel* phase can consist of any number of sub-phases that will be executed concurrently. For example, Figure 5.2 shows a parallel phase with the Task transitions *compute* and *sort*. Second, the *map* phase. Similar to the parallel phase, it can consist of any sub-phases, but each sub-phase is executed concurrently on different elements of an input array. Figure 5.3 shows an example: The map functions compute $y_i = \text{map}(x_i)$ simultaneously for all i, i.e., all transitions inside the bounding box fire simultaneously.

Conditional Routing. A *switch* phase uses conditional routing based on data values by annotating guarding functions to transitions. Figure 5.5 shows
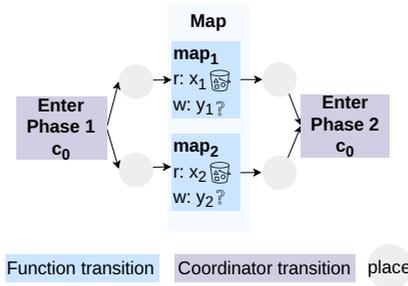
Figure 5.3.: *Map* function transition.

an example: add is invoked only if [pred(x)] evaluates to true; otherwise, subtract will be invoked. This represents an XOR-split depending on the data in x.

The first transition of a workflow in our model is always a *coordinator $c \in C$* that initializes the workflow and schedules functions for execution. Additional coordinator transitions take place between phases, meaning that the coordinator awaits the termination of the currently running functions and then schedules the functions of the next phase, explicitly modeling the orchestration of the workflow by the platform.

Figure 5.4 shows an example with C and SF as follows:

$$C = \langle c_0, EnterPhase1c_0, EnterPhase2c_1 \rangle$$

$$SF = \langle generate, map_1, map_2, process \rangle$$

For readability, we do not show the coordinator transitions when they can be skipped while preserving the control flow between function transitions, i.e., whenever a sequential phase is the next phase. This is because the sequential function already serves the purpose of the AND-join, otherwise realized by the coordinator transition. In Figure 5.4, this means we can leave out all coordinator transitions after the initial c_0 transition.

5.1.2. Resource Annotations

Data labeling functions indicate the required inputs and provided outputs of a transition. However, for the performance of serverless workflows, it
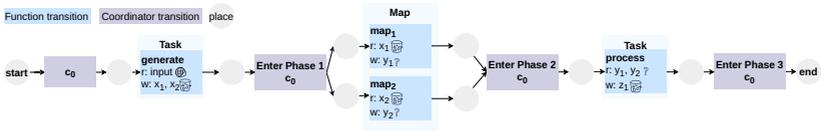
Figure 5.4.: Example workflow using our model based on WFD-nets.
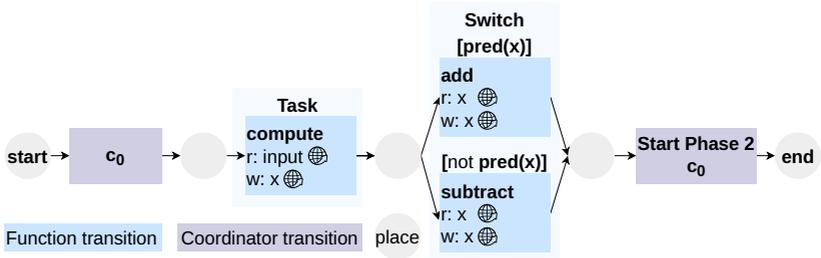
Figure 5.5.: Switch phase: If `pred(x)` is `true`, the `add` transition fires. Otherwise, `subtract` fires.

is important to know where the data resides and how it is provided, as performance could vary greatly depending on that. Therefore, we extend the notation of WFD-nets by annotating how read and write operations on data are performed using the following resource annotations:

- 🗄 **Object storage.** Data is saved in cloud storage in the same region. While this provides high capacity, cloud storage suffers from limited I/O bandwidth and high latency.

- 🗄 **NoSQL.** Data stored in NoSQL key-value storage provides low-latency data storage.

- 🌐 **Invocation Payload.** Small input data can be transferred by protocols such as HTTP and gRPC. However, the exact size limit is subject to the protocol and platform.

- ? **Transparent.** The type of transmission used when returning a payload is up to the provider and can change given the payload size.

- 💭 **Reference.** Some functions only need the reference to an object in the object storage rather than the object itself.

Formally, we define the set of resource annotations $A = \{o, n, p, t, r\}$ as additional element of the tuple of a WFD-net, with o representing data passing via the object storage, n via NoSQL, p via the invocation payload, t transparently, and r via reference. We define the corresponding resource annotation functions for reading and writing data as ra and rw as follows and also add them to the tuple of a WFD-net:

$$ra : \{(t, d) \in T \times D \mid d \in r(t)\} \rightarrow A$$

$$rw : \{(t, d) \in T \times D \mid d \in w(t)\} \rightarrow A$$

This means that each pair of a transition and a data element (t, d), with d being read or written by t, respectively, is assigned a resource annotation $a \in A$. By adding resource annotations, we do not change the behavior of the WFD-net. However, we enable checking the consistency of data accesses, for example, if the same data object is written and read using the same resource annotation.

We annotate the data passing in workflows using the respective icon and show an example in Figure 5.4. The function `generate` reads the data element `input` received via an invocation payload and writes its output to the object storage. The `map` functions each receive and read an array element containing x_1 and x_2, respectively, process it, and return their resulting elements y_1 and y_2 through a protocol the cloud provider decides. Once both `map` functions have finished execution, the `process` function receives and reads y_1 and y_2 as input and, finally, uploads the final result z of the workflow to the object storage.

5.2. Workflows Benchmark Suite

We now present the design and implementation of *SeBS-Flow*. To provide a reliable and fair comparison of various workflow technologies, we need to execute the same benchmark implementation on many platforms. However, the platforms exhibit vast differences in the programming model and API of their workflow services (Section 2.5). Thus, we define a platform-agnostic workflow definition (Section 5.2.1) based on our workflow model (Section 5.1). Then, we propose platform-specific generators that transcribe workflows to the respective proprietary definition of the desired platform (Section 5.2.2).

```
"phase_name": {
  "type": "type_name",
  "func_name": "serverless_function_name",
  "next": "name_of_next_phase"
}
```

Listing 5.1: General format of our JSON definition for a workflow phase.

We add the workflow representation and implementation to a serverless benchmark suite (Section 5.2.3).

While the preceding Master thesis [24] already proposed a platform-agnostic workflow definition, we extend the definition and transcription by adding the *Parallel* state and establishing the *Map* state as a higher-level phase able to contain multiple states and passing common parameters to them.

5.2.1. Platform-Agnostic Workflow Definition

Our workflow model encodes the application as a Petri Net, as defined in Section 5.1. To define workflows in *SeBS-Flow* conforming to our model, we use a JSON syntax and show its general format for defining a single workflow phase in Listing 5.1. The phase object is named with a phase_name that can be any string. Coordinator transitions encode the order of phases, represented by the next field of phases describing the consecutive workflow step. The next field refers to the phase name of the phase to be executed next. The workflow will terminate if the next field is not set. Each phase receives the output payload of the previous function as input. This means that the implementations of the functions need to conform to the resource annotations as defined in the workflow model and download and upload data as needed

```
"compute_phase": {
  "type": "task",
  "func_name": "compute"
}
```

Listing 5.2: *Task* Statement.

accordingly. The property `func_name` specifies the filename of the function to be called. Every phase has a `type`, relating to one of the available routing constructs introduced in Section 5.1.1. The `type` of a phase can refer to one of the following supported types:

Task. A `task` executes a single serverless function, constituting a sequential routing. Listing 5.2 shows an example with the *compute_phase* executing the function *compute*. This encodes the task function transition as shown in Figure 5.1.

Repeat. A `repeat` phase executes a function a given number of times. This syntactic sugar eases the modeling of a chain of tasks. The additional field `count` specifies how many times the functions should be executed. Listing 5.4 presents an example where the function `process` is invoked ten times, and the return payload of the ith invocation is passed onto the $i+1^{th}$ execution.

Map. The map phase is a parallel routing construct and concurrently executes the phases given as `states` one after another, starting with the phase given as `root`, on each element of the given `array` and returns an array again. While by default, only the respective array element is passed to the states as input, the phase can optionally define `common_parameters` that will be passed additionally. Listing 5.3 shows an example with the `process_names` phase: for each element of `customers`, the function `normalize` is executed concurrently.

```
"process_names": {
  "type": "map",
  "array": "customers",
  "root": "normalize",
  "next": "accumulate_emails",
  "states": {
    "normalize": {
      "type": "task",
      "func_name": "normalize"
    }
  }
}
```

Listing 5.3: *Map* Statement.

```
"process_10": {
  "type": "repeat",
  "func_name": "process",
  "count": 10
}
```

Listing 5.4: *Repeat* Statement.

```
"send_if_enough_data": {
  "type": "switch",
  "cases": [
    {
      "var": "data.length",
      "op": ">=",
      "val": 1048576,
      "next": "send_truncated"
    },
    "Break - each element is a separate case"
    { "var": "data.length",
      "op": ">=",
      "val": 1024,
      "next": "send"
    }
  ],
  "default": "log"
}
```

Listing 5.5: *Switch* Statement.

Only after all functions have terminated, the coordinator will transition to the next phase, which in this case is named `accumulate_emails`.

Loop. The `loop` phase is similar to `map` but traverses the given input array sequentially. Thus, loop encodes tasks that cannot be parallelized due to existing dependencies.

Switch. The `switch` phase is a conditional routing as the next phase is decided dynamically at runtime depending on the given condition. Listing 5.5 presents

```
"compute_and_sort": {
    "type": "parallel",
    "parallel_functions": [
      {
        "root": "compute",
        "states": {
          "compute": {
            "type": "task",
            "func_name": "compute"
          }
        }
      },
      {
        "root": "sort",
        "states": {
          "sort": {
              "type": "task",
              "func_name": "sort"
          }
        }
      }
    ],
    "next": "frequency_and_overlap"
}
```

Listing 5.6: *Parallel* Statement.

a simple `switch` phase where different functions are executed depending on the running variable `data.length`. The different `cases` are evaluated one after another, with the first one fulfilling the condition being executed.

Parallel. This higher-level phase corresponds to a parallel routing and executes independent sub-workflows concurrently. The sub-workflows can consist of any of the phases presented. All sub-workflows receive the complete output of the previous phase as input. The outputs of the sub-workflows are merged after all functions have completed execution. Listing 5.6 shows an example where the sub-workflows `compute` and `sort` are executed in parallel, both consisting of one function each. After they both have finished, the next phase

```json
{
  "root": "generate_phase",
  "states": {
    "generate_phase": {
      "type": "task",
      "func_name": "generate",
      "next": "map_phase"
    },
    "map_phase": {
      "type": "map",
      "array": "x",
      "root": "map",
      "next": "process_phase",
      "states": {
        "map": {
          "type": "task",
          "func_name": "map"
        }
      }
    },
    "process_phase": {
      "type": "task",
      "func_name": "process",
    }
  }
}
```

Listing 5.7: Example workflow from Figure 5.4 encoded using our JSON syntax.

frequency_and_overlap will be executed. This encodes the workflow phase as shown in Figure 5.2.

We show an example of a complete workflow definition in Listing 5.7, encoding the same workflow as shown in Figure 5.4. The root entry specifies the name of the phase that should be executed first, in this case the generate_phase. The states entry then contains all phases of the workflow. As mentioned above, each phase receives the output payload of the previous function as input. It is therefore up to the implementation of the functions

to download and upload data as needed, and construct their output payload accordingly. Only in the case of the `map` phase, we explicitly specify which array is used for distributing its elements to single functions. The level of parallelism is then decided dynamically at runtime depending on the size of the given array.

5.2.2. Platform-Specific Transcription

We map the six phases building a serverless workflow to different modeling language features on each platform, and we select different cloud language components to represent our states. As we see, the mapping of our identified workflow constructs to platform capabilities is not straightforward. This is why we believe a common workflow definition will be helpful for developers as they do not have to account for these differences when experimenting with multiple platforms.

5.2.2.1. AWS

The most notable difficulty when transcribing our platform-agnostic definition to the state machine definition of AWS Step Functions is the `loop` phase. Step Functions do not inherently support sequential array iteration. Their official documentation suggests using an additional serverless function that iterates over a given range [4], which is inefficient. Thus, we use the AWS `map` state and configure it to traverse the given array sequentially, yielding the semantics of a `loop`. A downside of this approach is that the input to each function is the same, i.e., consecutively executed functions can observe the results of computations of their predecessors only if uploaded to the object storage.

5.2.2.2. Google Cloud

Google Cloud Workflows do not natively support a `task` type. Instead, the recommended approach for invoking Cloud Functions [61] is to create a state performing a POST request and providing the trigger URL of the desired function as input. However, this requires additional states for each `task` and `map` to parse the HTTP response of a function and assign results. Moreover,

the parallel `map` execution accepts only other workflows and not states, which requires creating another sub-workflow, even if it contains only a single function to be invoked. Finally, there is no mechanism for passing additional arguments to a `map` function, which is necessary for us to track measurements. As a workaround, the input array is zipped together with an array consisting of the additional parameter passed by the benchmarking infrastructure.

5.2.2.3. Azure

Azure uses the dynamic model of Durable Functions instead of state machines. This means that users have to write the orchestrator themselves as a separate function. Therefore, we upload our workflow definition together with the function code and an orchestrator function. Our orchestrator then parses the definition as input, decodes our definition, and executes it by spawning new function executions. Sub-workflows within `map` and `parallel` phases are executed by invoking sub-orchestrators.

5.2.3. Benchmark Suite

We follow standard design practices to build a new benchmark suite: it should be relevant, extensible, easy to use, and reproducible [22, 40, 77, 92]. Our suite is relevant as we include applications representing a variety of workloads in the industry and academia (Section 5.3). The implementation is based on an abstract workflow definition and can be extended to new platforms and programming models by implementing a single interface that transcribes our model definition to the new platform. To ensure that we fulfill the two remaining criteria, we build our implementation upon SeBS [40], an established benchmark suite for FaaS:

Easy to use. To ensure self-validation of benchmarks, they must be easy to deploy and execute [92]. In the last few years, serverless platforms have been continuously updated. In addition to deprecating old language runtimes and introducing new ones, cloud operators upgrade their internal infrastructure. For example, AWS Lambda introduced ARM functions based on Graviton CPUs in 2021 [5], Azure Functions migrated to the fourth version of their serverless runtime in 2021 [11], while Google Cloud introduced a second

generation of serverless functions in 2022 [64]. The changes are not only an opportunity to extend benchmarking experiments, but they often replace older solutions, making old benchmark implementations no longer deployable. Thus, integration into a maintained and up-to-date platform with an active community – such as SeBS – helps to integrate new developments continuously and avoids pushing this task to the end user, which would inhibit reproducibility.

Reproducible. *SeBS-Flow* is multi-platform, supports automatic deployment of functions to the cloud, and integrates with services like storage and cloud logging, allowing developers to focus on the actual implementation rather than specifics of cloud providers, which can be time-consuming [35, 121]. These features are critical to reproducing measurements, as the user intervention is minimized, and the experiments are executed automatically from a provided configuration.

Serverless functions need cloud-managed storage to access data and retain state across invocations. To that end, SeBS automatically manages object storage instances and provides functions with a multi-cloud API. To create realistic workflow representations of web applications, we need to support low-latency data stores other than object storage. We chose NoSQL key-value storage for this task and extended SeBS with a high-level interface for creating, modifying, retrieving, and deleting items. The interface supports a partition and an optional sorting key. Each benchmark function can use multiple tables managed by the benchmark suite. We map the tables to DynamoDB on AWS, CosmosDB on Microsoft Azure, and Firestore in Datastore mode on Google Cloud.

To measure runtime and performance, we collect the following data points for each function invoked during a workflow execution:

1. **start:** timestamp T_{start}^{sf} at the start of function sf's execution

2. **end:** timestamp T_{end}^{sf} at the end of function sf's execution

3. **requestID:** provider-internal identifier used to locate billing information

4. **containerID:** in all cloud providers, we detect container reuse and assign unique container identifiers by using the temporary filesystem and global variables.

To reason about the runtime of phases, we use the following notations:

$$T_{start}^S = \min_{sf \in S}(T_{start}^{sf}) \qquad T_{end}^S = \max_{sf \in S}(T_{end}^{sf})$$

$sf \in S$ is defined as all functions sf executed within a single workflow phase S. For example, the map phase invokes multiple functions simultaneously, and its beginning and end are defined by the earliest and latest functions, respectively. These values are sent to a Redis [118] instance deployed in the same cloud region. We chose an in-memory cache as it provides sub-millisecond latencies, reducing the risk of distorting the performance measurements.

5.3. Benchmark Applications

In *SeBS-Flow*, we implement six benchmarks covering various real-life workloads where a distributed computation is a natural fit. In addition, we implement four microbenchmarks used in the evaluation: function chain, object storage performance, parallel invocations (Section 5.5.3.1), and selfish detour (Section 5.5.3.2). The selected benchmarks cover various domains that use workflows (Table 5.2) and correspond to previous findings on the characterization of workflow use cases [48, 49] regarding control-flow, number of functions, parallel invocations of the same functions, and longer runtimes: While 33% of workflows include complex control flow, 50% are sequential, which we cover with Trip Booking and the function chain microbenchmark. 72% of workflows use fewer than ten different functions, 52% involve parallel invocations of the same function, and 25% contain functions with an estimated runtime of over one minute, which is also included in our benchmark suite.

In addition to the benchmarks introduced in the preceding Master thesis [24], we add the scientific benchmark *1000Genome* and the web application *Trip Booking*. Moreover, we improve the implementation of the *Video* and *Ex-Camera* benchmarks by fixing the data-flow between functions and ensuring different files are used per workflow invocation, respectively.

Video Analysis   In this analysis benchmark, based on a parallelized version of the sequential benchmark in vSwarm [156] (Figure 5.6), we detect all specified objects in a video. Functions decode video frames and perform detection with
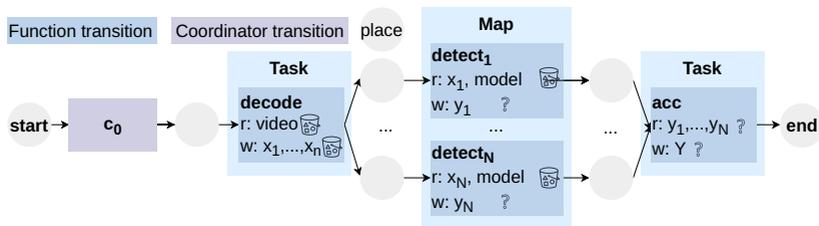
Figure 5.6.: The Video Analysis benchmark.

the Faster R-CNN model [120]. The video and the model are uploaded to the object storage before. The `decode` function first downloads the video, decodes F frames, and then uploads $N = \lceil \frac{F}{B} \rceil$ batches of size B. N parallel `detect` functions compute Y_i, all detections with the model's confidence $p > 0.5$. Finally, all detections are accumulated in `acc`, which returns the final payload Y. Throughout our experiments, we fixed the number of frames $F = 10$ and batch size $B = 5$, yielding two parallel functions in the map phase.

Trip Booking The benchmark represents web applications, and it mocks a common example of reserving a hotel, car rental, and flight [107, 146]. We show the workflow in Figure 5.7. The workflow is a pipeline of functions mocking the reservation system by storing trip data in a shared NoSQL database. It implements the SAGA pattern of long-running transactions [57] where a failure triggers the reversal of prior changes. For testing, we simulate failure in the last *confirm* function, which is followed by three consecutive functions to reverse the booking.

Benchmark	#functions	Parallelism	Critical path	Download [MB]	Upload [MB]
Video	4	2	3	238.83	7.48
Trip Booking	7	1	4/7	0.0	0.0
MapReduce	9	5	4	0.02	0.04
ExCamera	16	5	6	302.07	17.49
ML	3	2	2	7.82	3.91
1000Genome	19	12	4	273.54	3.47

Table 5.2.: Key features of different benchmarks.
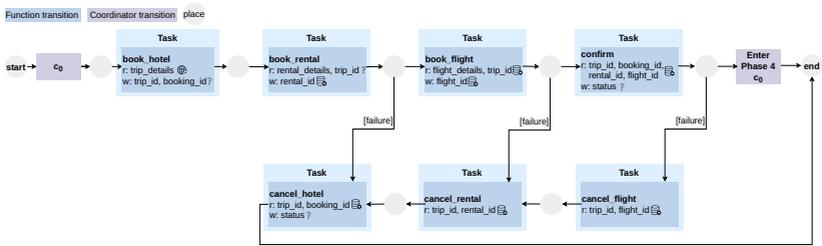
Figure 5.7.: The Trip Booking benchmark.

MapReduce   Figure 5.9 shows our example of a MapReduce job, which is based on prior implementations [106, 156] and performs the standard problem of word counting. First, the split function partitions the input text containing M different words into N batches. N parallel map functions count how often each word occurs in their text chunk next and return these counts separately for each word, denoted as $Y_{M,i}$ in Figure 5.9. Next, shuffle flattens the resulting array $Y_{M,i}|i < M$. Finally, M reducers count the total number of occurrences for their respective word in parallel, yielding Z_i. The benchmark has two parameters that influence the amount of work: the number of mapping functions N, and the total number of words W. We set $N = 3$ and $W = 5000$, containing $M = 5$ different words. MapReduce frameworks typically execute fully in parallel. However, the available workflow primitives of the platforms necessitate the shuffle function, not relying on the array Y_i itself but flattening it to ensure that reduce achieves the desired level of parallelism.

ExCamera   ExCamera [54] uses interdependent video-processing tasks to encode videos efficiently in parallel. A video with M total frames is processed in chunks of N frames by $\frac{M}{N} = T$ parallel functions. First, each frame is encoded, yielding one key frame and $N - 1$ interframes. Decode decodes all N frames again, calculating the final state. The final state from the first frame of the chunk is used for reencoding the other frames, resulting in one final state and $N - 2$ interframes. Lastly, the interframes are rebased sequentially based on the previous result. We derive our implementation from the original description of ExCamera [54] and the available implementation [53] (Figure 5.10). We use $M = 30$ total frames and a chunk size of $N = 6$, resulting in five parallel functions.
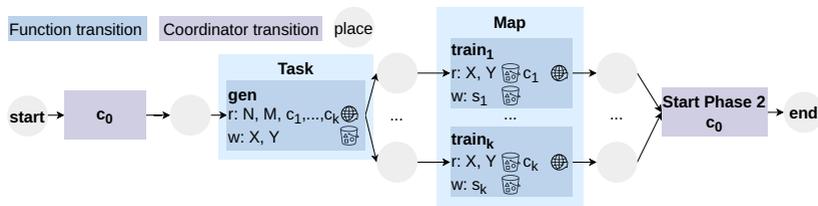
Figure 5.8.: The Machine Learning benchmark.

Machine Learning   This workload represents a typical training pipeline (Figure 5.8). The workflow starts with `gen` generating a dataset, accepting the number of samples N and the number of features M as input. Then, we `train` K different classifiers C_i in parallel. In our tests, we generate $N = 500$ samples and $M = 1024$ features. We train $K = 2$ classifiers: a Support Vector Machine [117], and a Random Forest [25], creating two concurrent functions.

1000Genomes   This scientific workflow identifies mutational overlaps using data from the 1000 Genomes project to provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. It consists of five tasks and three phases (Figure 5.11): First, N `individuals` functions parse the data for their chunk of the input file of size M and then upload their results to the cloud storage. While `individuals_merge` merges the results into one, `sifting` computes the SIFT scores using the `SNP_variants` file uploaded to the cloud storage prior to workflow execution. In the last phase, `mutation_overlap` measures the overlap in SNP variants among pairs of individuals, and `frequency` measures the frequency of mutation overlapping. Both functions are executed once per population P. The benchmark has the number of lines in the input file M, number of parallel `individuals` functions N, and number of populations P as input variables. Each `individuals` function receives $\frac{M}{N}$ lines as input. We change the original workflow, written for a cluster environment, to let the `individuals` functions download only their respective chunk of the input file from cloud storage. We use $M = 1250$ lines in the input file, $N = 5$ parallel individuals function, and $P = 6$ populations.
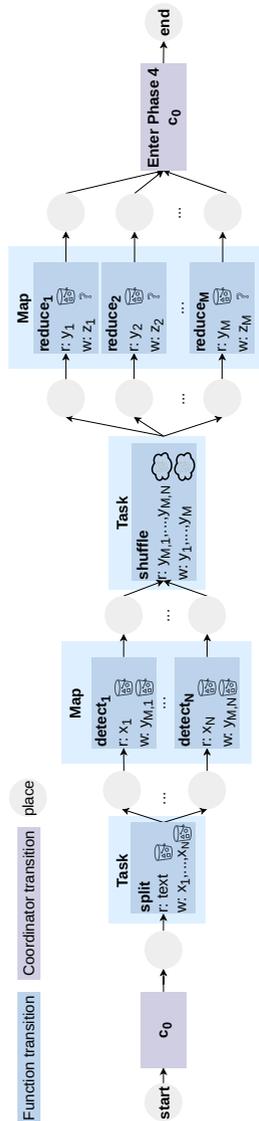
Figure 5.9.: The MapReduce benchmark.
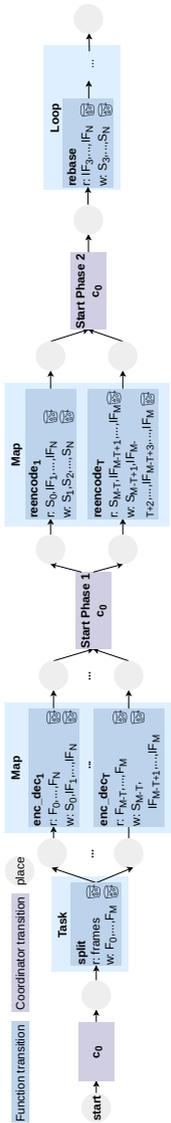
Figure 5.10.: The ExCamera benchmark.
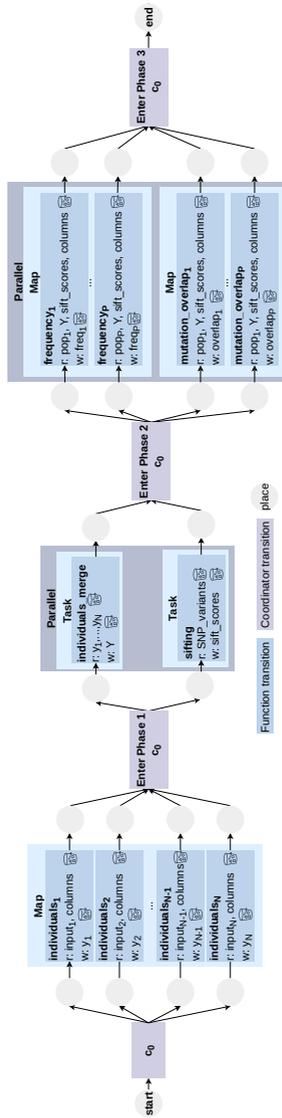
Figure 5.11.: The 1000 Genomes benchmark.

5.4. Evaluation of Workflow Model

We want to ensure that software engineers and researchers can use our model and workflows. For this, it needs to be capable of expressing the workflows used. Furthermore, our transcription should not add overhead to the workflow, obstructing from finding the fastest platform for a given workflow. We therefore review existing literature on serverless workflows to evaluate whether our model is general enough to express applications of workflows in academia, and if the transcription from our platform-independent representation to the platform-specific representations of workflows can add overhead compared to their native implementation. We do so by using the meta-search engine Google Scholar to find peer-reviewed publications containing the keywords *cloud, orchestration*, and *serverless workflow* or *serverless DAG*. We exclude papers that are not in English, do not use a workflow benchmark, or were published before 2017, the year of the first serverless workflows in the cloud. This results in 72 papers analyzed papers (see Table 5.1, p. 94 for their categorization). We provide the complete list of papers and analysis results as part of the replication package.

5.4.1. Expressiveness of our Model

We analyze the workflow benchmarks used in the literature and evaluate whether our model can represent the control flow within the workflows without adding unnecessary dependencies between their tasks. We distinguish between the capabilities of our model (Section 5.1) and our implementation (Section 5.2).

Out of the 72 papers, 14 did not provide sufficient detail on the workflows used and their dependencies to judge if we can express them. Benchmarks used in two papers are not presentable by our model (Section 5.1), as they introduce new programming models to support communication between functions during their execution and orchestration based on the current load of the system, which is out of scope for us. Benchmarks used in three more papers can be modeled but not transcribed to platform-specific representations (Section 5.2). For two of them, cloud platforms are the limitations, such as ending the workflow as a result of a switch state (not possible on AWS) and using multi-stage inputs, i.e., using the output of a previously executed function as input without passing it to the functions invoked in-between.

The third one uses a *switch* state requiring two conditions to be true. While we do not support transcribing this currently, transcription can easily be added to the implementation. We fully support modeling and transcribing the workflows described in 53 of the 58 analyzed papers. Therefore, we conclude that our model does not have general limitations within the scope of programming models not allowing for communication between functions and using orchestration based on dynamic characteristics of the system and can be used to model and execute workflows on serverless platforms.

5.4.2. Overhead of our Model

To check if our model and transcription create overhead compared to a native implementation, we evaluate available benchmark implementations used in the analyzed papers and compare them to our transcription of their workflows. Only 10 of the 72 papers include an artifact containing workflow implementations or show their implementation as part of the paper for any of the platforms we support. None of them uses Google Cloud Workflows.

In total, we find eleven AWS Step Functions state machines. One of them uses the *AND* choice type. We currently do not transcribe this choice type and are therefore not able to generate the same state machine. However, if we would add the transcription, the resulting state machine would look similar. Another state machine adds *fail* and *success* states before ending the workflow, which only introduces overhead as compared to just ending the workflow. The other nine state machines use the same states with the same parameters in the same order as the state machines we transcribe, except for the fact that they specify each parameter explicitly as part of the state machine while we wrap them within a single *payload* entry, which does not affect the overhead.

Four of the papers provide implementations for a total of six workflows using Azure Durable Functions. While one paper only provides an implementation using stateful entity functions, as opposed to the stateless activity functions targeted by us, the other five workflow implementations use activities to orchestrate tasks similar to our transcription. Since we parse our platform-independent representation within the orchestrator, this could introduce an overhead during the execution of the orchestrator. However, the evaluation of the 1000Genome benchmark, the benchmark with the most functions,

shows that the average duration of the orchestrator function is only 13.6 milliseconds.

We conclude that *SeBS-Flow* does not introduce noteworthy overhead in the workflows compared to their native implementation.

5.4.3. Threats to Validity

We only used one search query to find relevant works to include in our evaluation, bearing the risk of missing some results. We mitigated this by evaluating different queries beforehand, evaluating the relevance of papers found, and checking if the results included relevant papers we knew as a gold standard [44]. Moreover, the analyzed papers already show a variety of workflows from different domains, which makes us confident that our evaluation of the expressiveness of our model can generalize to workflows used in other works. For the evaluation of the overhead, a threat to the external validity of our evaluation is the limited number of artifacts available, with none available that uses GC Workflows. While our transcription follows best practices and tutorials as provided by the cloud providers and matches the artifacts we found, usage in other projects may be different.

5.5. Evaluation of Cloud Services

We use *SeBS-Flow* to evaluate and compare three major cloud workflow services – AWS Step Functions, Google Cloud Workflows, and Azure Durable functions. We first evaluate the overall runtime of the benchmarks from Section 5.3. Next, we break the runtime into the runtime of the *critical path* and *overhead* between function invocations, investigating different causes for their variation. This enables detailed insights into how different characteristics, such as level of parallelism throughout the workflow, impact the runtime of a workflow. Using these insights, developers can estimate which platform may be well-suited for their workflow based on its characteristics.

Moreover, we are interested in how the execution of scientific workflows on cloud workflow services compares to execution in a cluster environment: There is rising interest in the scientific community to use serverless solutions [49], accompanied by experimentation with serverless offerings of the

platforms [105] and management systems for serverless execution of scientific workflows [83, 85, 126, 127]. However, they do not consider the workflow orchestration systems the cloud platforms offer.

Next, we compare the pricing of the workflows on the different cloud workflow platforms. Finally, we look at the long-term stability and evolution of performance by comparing the overall runtime of benchmarks from 2022 and 2024. To sum up, we investigate the following research questions:

RQ3 How can workflows be modeled and transcribed to different platforms to enable comparative evaluation of their performance?

   RQ3.1 What are the major runtime differences between platforms?

   RQ3.2 What causes runtime and stability differences?

   RQ3.2.1 What are overhead sources between function invocations?

   RQ3.2.2 What causes variations in the critical path?

   RQ3.3 How well can serverless workflow orchestration support scientific workflows?

   RQ3.4 How does the pricing compare between platforms?

   RQ3.5 How did the performance and stability of the platforms evolve over time?

5.5.1. Methodology

We deploy benchmarks on Azure to the *europe-west* region, on AWS to *us-east-1*, and on Google Cloud to *us-east1*. We use the lowest common memory configuration that successfully executes the workflow on AWS and Google Cloud, at least 256 MB for computational functions and 128 MB for simple web applications. We invoke the application benchmarks in *burst* mode, triggering 30 executions at once and accepting all successful workflow executions, as other work suggests that most serverless applications have potentially bursty workloads [48].

We check how often we should repeat experiments by computing non-parametric confidence intervals on the results for the MapReduce benchmark. We aim for being within a 5% interval of the median with a 95% confidence

interval. For the burst mode with 30 executions triggered at once, this results in 1, 2, and 6 repetitions on AWS, GCP, and Azure, respectively. We opt to execute and repeat all experiments 6 times, resulting in 180 executions. However, we could only obtain 30 executions of the 1000Genome benchmark on Azure due to frequent timeout issues.

5.5.2. RQ3.1: Runtime Differences among Platforms

We first compare the total runtime of each benchmark on selected platforms. We calculate the runtime by subtracting the first *start* timestamp from the last *end* timestamp, $S \in W$ denoting all phases S in a workflow W:

$$T_R = T_{end}^W - T_{start}^W = \max_{S \in W}(T_{end}^S) - \min_{S \in W}(T_{start}^S)$$

The runtime results presented in Figure 5.12 do not yield a single fastest platform among all our benchmarks. AWS is the fastest platform for three of the six benchmarks – Video Analysis, ExCamera, and 1000Genome – while performing relatively well for the other three. While Google Cloud's performance is comparable to AWS, it is 1.55-1.97x slower on the three benchmarks and the slowest platform for MapReduce and Machine Learning. Azure Durable functions, however, perform very well on the MapReduce and Machine Learning benchmarks, showing the shortest median runtime, but are the slowest platform for the Video Analysis, the ExCamera, and the 1000Genome benchmark. For Trip Booking, Azure also achieves the best median performance but suffers from large outliers. We investigate the potential causes of slowdown in the next section. All platforms demonstrate variable performance, with Azure showing the largest variance.

5.5.3. RQ3.2: Causes for Runtime and Stability Differences.

According to our benchmarking results, AWS and Google Cloud provide a mostly performance-reliable workflow service, whereas Azure has considerably higher runtime variability across all benchmarks. To investigate the causes for the slowdown, we split the runtime into two components: the critical path T_C computed as the sum of all states' maximum runtime within one phase, and the overhead T_O caused by the scheduling and data movement

(a) Video Analysis, 2048MB.

(b) ExCamera, 256MB.

(c) MapReduce, 256MB.

(d) TripBooking, 128MB.

(e) Machine Learning, 1024MB.

(f) 1000Genome, 2048MB.
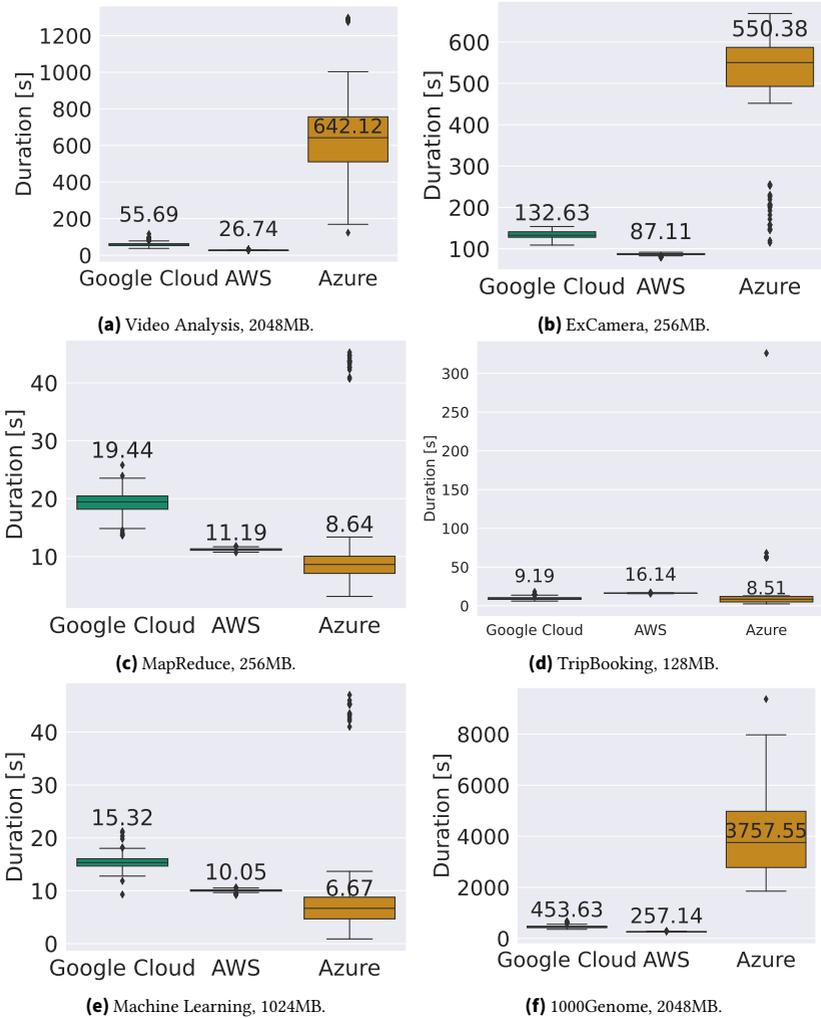
Figure 5.12.: Runtime of benchmark applications on AWS Step Functions, GC Workflows, and Azure Durable, *burst* invocations.

conducted by the cloud workflow service. We calculate the absolute overhead by subtracting the critical path from the total runtime as follows:

$$T_O = T_R - T_C = (T_{end}^{W} - T_{start}^{W}) - \underbrace{\sum_{S \in W} (T_{end}^{S} - T_{start}^{S})}_{\text{critical path}}$$

Figure 5.13 presents the duration of the critical path opaque and the overhead shaded for all benchmarks. We summarize the results as follows: Azure's runtime is significantly impacted by scheduling overhead. For instance, in the ExCamera benchmark, the scheduling overhead averages 495.5 seconds, which is more than 36× the length of its critical path of 13.5 seconds. The Machine Learning benchmark exhibits the least overhead, at 5× the length of its critical path. For the 1000Genomes benchmark, the scheduling overhead on Azure averages 3754.94 seconds, which is over 10 times the critical path duration of 371.47 seconds. Despite the high overheads, Azure achieves very low critical path durations across all benchmarks, recording the fastest critical paths for ExCamera, MapReduce, and Machine Learning. Google Cloud demonstrates the slowest critical path throughout the entire benchmark suite. While it experiences less overhead than Azure, the overhead is bigger than on AWS for all benchmarks. AWS shows the smallest overheads throughout the benchmarks and the shortest critical path for the Video Analysis and 1000Genome benchmarks.

To summarize, orchestration overhead causes long runtimes and performance variances on Azure. For AWS and Google Cloud, however, the runtime of the critical path varies.

5.5.3.1. RQ3.2.1 Sources of Overhead

To further analyze the reasons for the different amounts of overhead between the platforms and benchmarks, we analyze three common sources of overhead: object storage I/O, parallel schedule, and function return payload.

Cloud Storage I/O.   The size of data downloaded from the object storage differs between benchmarks (Table 5.2, p. 110), with hundreds of megabytes in ExCamera, 1000Genomes, and Video Analysis. These benchmarks experience

(a) Video Analysis, 2048MB.

(b) ExCamera, 256MB.

(c) MapReduce, 256MB.

(d) Trip Booking, 128MB.

(e) Machine Learning, 1024MB.

(f) 1000Genome, 2048MB.
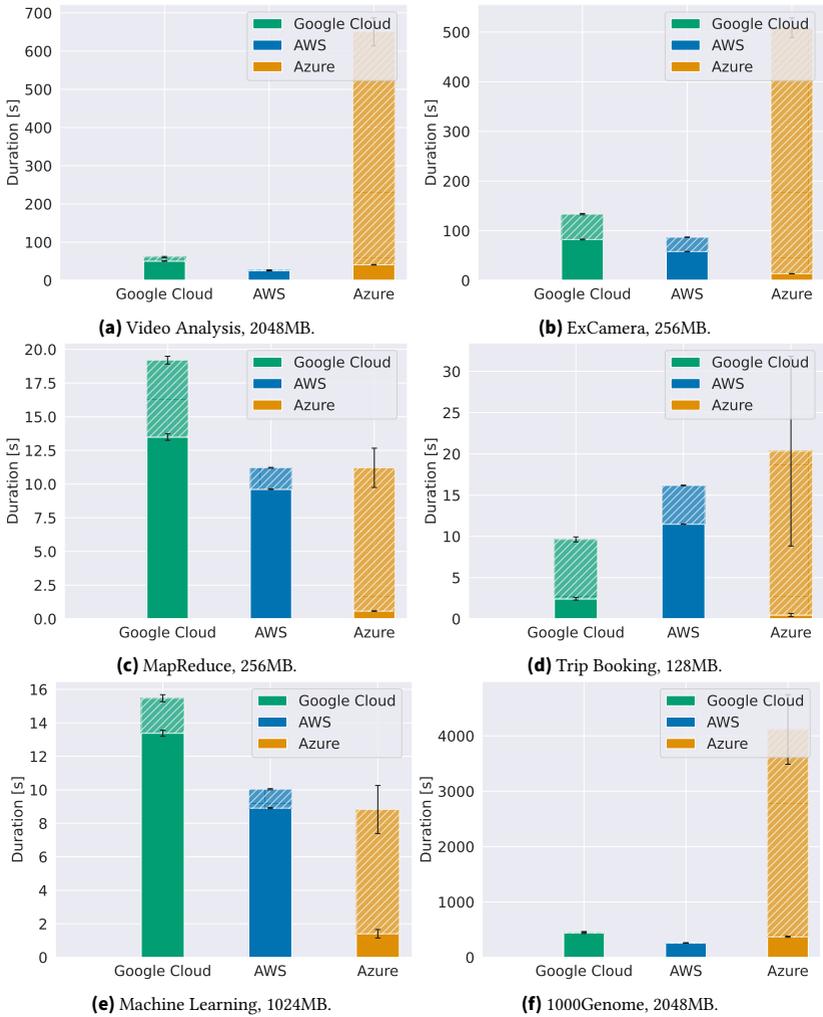
Figure 5.13.: Mean duration of critical path (opaque) and overhead (shaded) of different benchmarks on considered platforms, *burst* invocations.
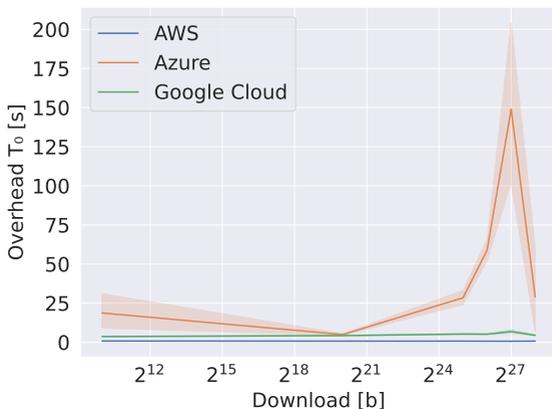
Figure 5.14.: Overhead T_O of storage I/O, 20 functions, $2^{10} \leq D \leq 2^{28}$, 512MB, *burst* invocations. Light area shows the 95% confidence interval.

the highest relative and absolute overheads on Azure, with a duration of 495.5s, 3743.8s, and 609.7s, respectively. To verify that this correlation is indeed causation, we execute a microbenchmark evaluating the cloud storage I/O performance. We invoke 20 functions in parallel, each attempting to download a file of size D from the storage. Figure 5.14 shows the results. The overhead T_O remains stagnant for AWS at around one second and nearly stagnant on Google Cloud at around five seconds, increasing a bit for downloads larger than 1MB. On Azure, the overhead shows the largest variance and increases for downloads larger than 1MB, from 4.9 seconds for 1 MB files up to 148.9s for 128 MB. However, the overhead decreases to 29.01s for 256 MB, possibly due to optimizations. Still, the overhead is 6.5× and 40.2× higher than for Google Cloud and AWS for 256 MB, respectively. We conclude that the data downloads can account for a significant part of the large overhead measured on Azure Durable.

Parallel Scheduling. Another potential source of overhead are parallel invocations within a benchmark. Benchmarks with the highest degree of parallel computation – ExCamera and 1000Genomes – show the largest overheads of Azure. We test that hypothesis by executing a microbenchmark that spawns a `generate` function that spawns N `sleep` functions in parallel in the following

workflow phase, each one sleeping for T seconds. We execute 30 invocations of the microbenchmark concurrently, once for each possible combination of $T = \{1, 5, 10, 15, 20\}$ and $N = \{2, 4, 816\}$. Figure 5.15 shows the relative overhead of the actual runtime of the benchmark compared to the execution time of the generate and sleep functions. We calculate the relative overhead T_{rO} as $T_{rO} = T_R \div T_C$. This means that a value of one indicates no overhead, and a value of two already indicates that the overhead doubles the total runtime of the workflow. Figure 5.16 shows the absolute overhead T_O in seconds.

AWS functions (see Figure 5.15b) demonstrate modest relative overhead T_{rO}, with largest values of 1.6 and 1.5 for the shortest duration. The relative overhead decreases with increasing duration, resulting in very low relative overheads of 1.02 to 1.04 for the durations of 15 and 20 seconds. However, the absolute overhead T_O (see Figure 5.16b) incurred for a certain number of parallel functions remains relatively constant, with the highest overhead of around 0.63 seconds for 16 parallel functions and lowest of around 0.52s for two parallel functions.

GC functions (see Figure 5.15c) present a more considerable relative slowdown that increases with the number of parallel tasks. There, the system puts a cap on scaling up and reuses containers, as 30 invocations with $N = 2, T = 1$ start 60 different function containers on AWS, but only 30 on Google Cloud. As for AWS, the relative overhead on Google Cloud also increases for shorter durations, ranging from 3.2 to 5 for one second and only from 1.1 to 1.2 for 20 seconds. The absolute overhead (see Figure 5.16c) incurred for a certain number of parallel functions varies a bit more than for AWS, not consistently decreasing or increasing with the duration. The highest absolute overhead is 5.0s for $N = 16, T = 5$, lowest 1.3s for $N = 2, T = 20$.

Azure (see Figure 5.15a) experiences an order of magnitude larger relative overhead that increases with the parallelism factor, but does not seem to be correlated to the function runtime. Azure shows the highest relative overhead of 42 for $N = 16, T = 5$, and the lowest relative overhead of 8 for $N = 2, T = 20s$. Also, the absolute overhead (see Figure 5.16a) is not consistently correlated to the number of parallel functions: While the absolute overhead increases with the number of parallel functions for $T = 2$, $T = 4$, and $T = 20$, this does not hold true for $T = 10$ and $T = 15$. However, we can correlate the absolute overhead with the function runtime as it increases for longer runtimes.
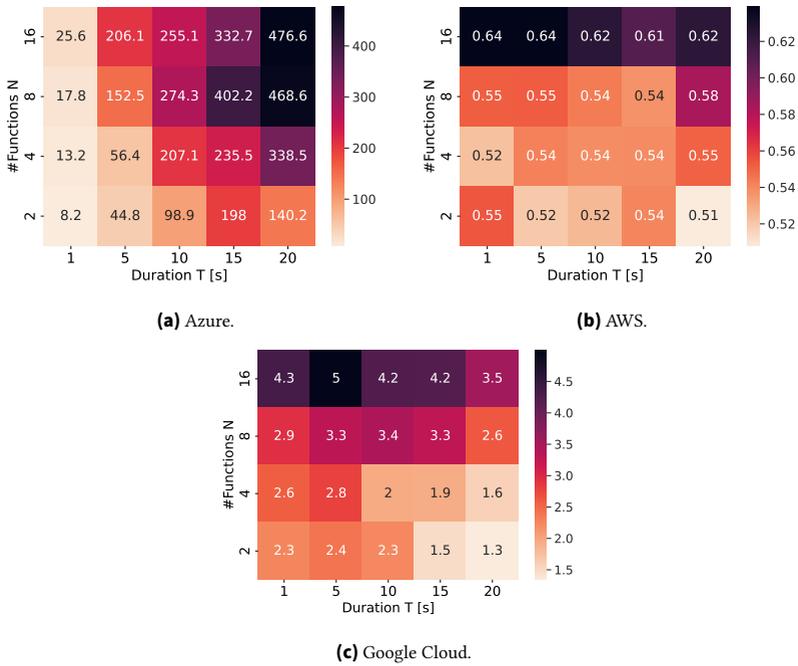
(a) Azure.

(b) AWS.

(c) Google Cloud.
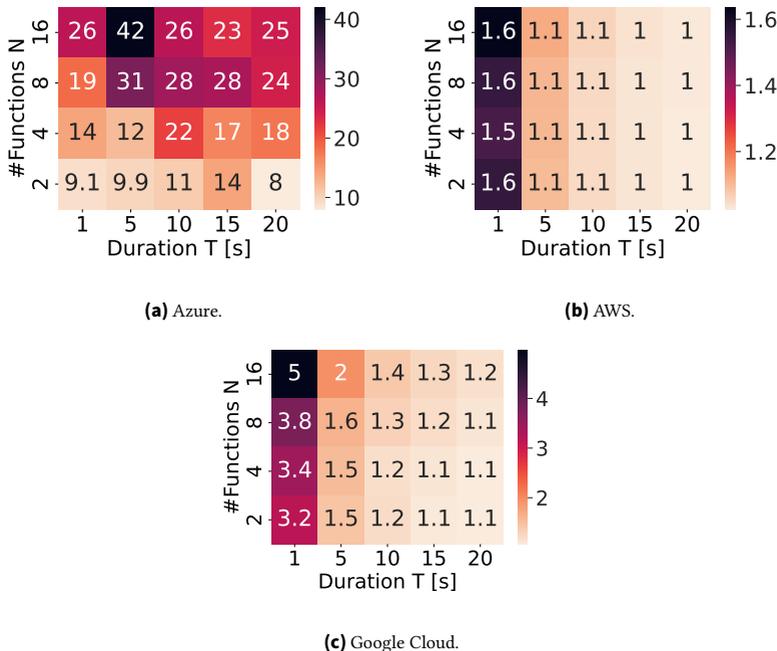
Figure 5.15.: The relative overhead of the parallel sleep microbenchmark, $2 \leq N \leq 16$, $1 \leq T \leq 20$, 256MB, *burst* invocations.

To better understand the impact of limited parallel scalability on our benchmarks, we measure the number of distinct sandboxes allocated at any given time until the last function execution has terminated. We invoke 30 concurrent executions of our workflow benchmarks and display the scaling behavior in Figure 5.17. We do not show results for the 1000Genomes benchmark, as we were not able to obtain 30 concurrent successful executions on Azure due to frequent timeout issues.

Throughout the benchmarks, AWS spins up most containers for each of the benchmarks, putting almost every function execution in a new container and reusing very few containers. The transitions between the different phases of the workflows are clearly visible. Google Cloud exhibits similar scaling behavior, showing the same local maxima for the Video Analysis, ExCamera, and Machine Learning benchmark. However, it also spins up new containers

(a) Azure.

(b) AWS.

(c) Google Cloud.

Figure 5.16.: The absolute overhead of the parallel sleep microbenchmark, $2 \leq N \leq 16$, $1 \leq T \leq 20$, 256MB, *burst* invocations.

more slowly, showing a less steep curve, and reuses containers more, as for example for the MapReduce benchmark. Azure produces a much more constant curve that remains similar throughout the benchmarks, never allocating more than ten containers simultaneously and even scaling down to zero containers during workflow executions. The transitions between the phases of the workflows are not visible.

In conclusion, parallel function executions cause high overheads on Azure. In contrast, AWS and GC Functions experience less overhead and spin up more containers to execute functions in parallel, making them better suited for workflows with a high level of parallelism.

(a) Video Analysis

(b) ExCamera

(c) MapReduce

(d) TripBooking

(e) Machine Learning
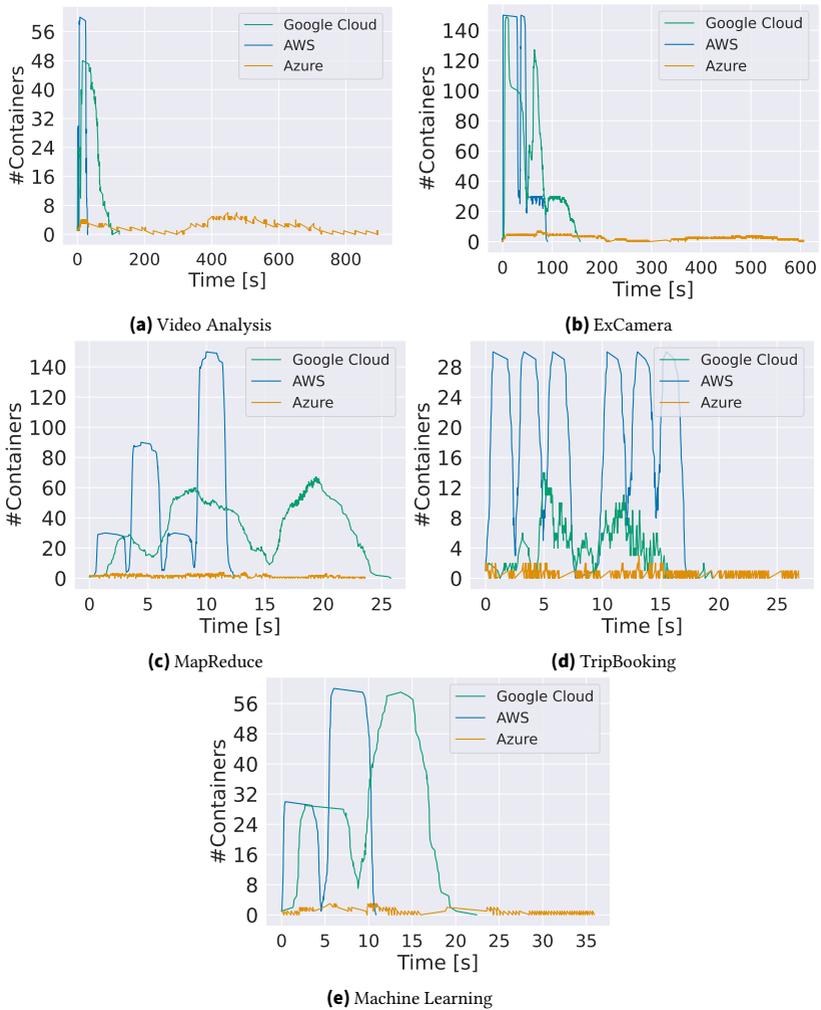
Figure 5.17.: Scaling profiles: the number of distinct containers used for 30 consecutive workflow invocations.
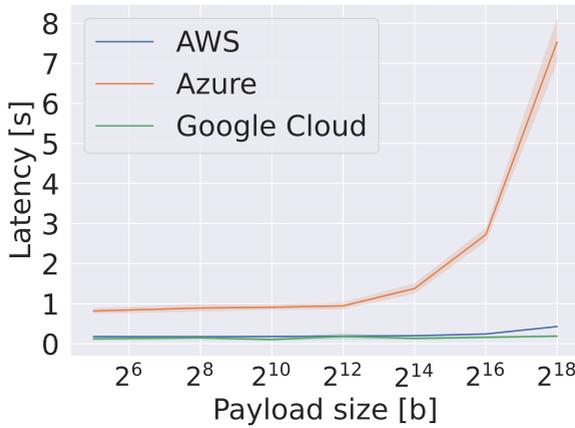
Figure 5.18.: Invocation latency, $2^5 \leq M < 2^{18}$, 256MB, *warm* invocations.

Return Payload. How data is transmitted when returning a payload is transparent to the user (cf. Section 5.1.2), and may change depending on the payload size. Therefore, we evaluate the overhead resulting from the function return payload size. We use a microbenchmark consisting of a function chain, where functions return M bytes of result sent to the consecutive function. Each function returns a unique payload. We deploy the chain with ten functions and test the varying input size until the limit of $M = 2^{18} - 1000b$ on Google Cloud, invoking the chain 30 times simultaneously and using results from warm invocations only. Figure 5.18 shows that Google Cloud exposes the shortest latency of the three platforms: The latency on Google Cloud remains relatively constant at around 0.15s, showing no consistent increase for larger payloads. The latency on AWS is a bit higher and remains almost constant for payloads from 2^5 bytes (0.17s) to 2^{16} bytes (0.24s) only. For $M = 2^{18} - 1000b$, it increases up to 0.43s, almost doubling the latency. On Azure, however, the latency is considerably higher for $M = 2^5b$ already with 0.82s, increasing dramatically from 1.38s for $M = 2^{14}b$ (16 kB) to up to 7.53s for $M = 2^{18} - 1000b$, suggesting an influence of remote storage or queue. While this may present a significant source of overhead in certain applications, our benchmarks do not return payloads larger than 1MB. Therefore, this overhead can only account for a part of the slowdown.

Conclusions. The microbenchmarks demonstrate that a significant part of the overhead observed on Azure originates from the parallel schedules and object storage I/O. Another potential source which is out of scope of our study could be dynamic orchestration: A statically scheduled system could optimize function placement, data prefetching, and scalability by using a priori knowledge. However, as Azure is dynamically orchestrated, the scheduler does not know which functions will be executed and is thus unable to execute any of the optimizations outlined before.

5.5.3.2. RQ3.2.2 Critical Path Discrepancy

The runtime of benchmarks across platforms shows that in addition to varying overhead, the critical path of computation can be significantly different. To understand the reasons behind this difference, we analyze how the critical path duration is impacted by three factors: data downloads, the varying CPU allocation, and frequency of cold starts.

Data Downloads. We reuse the measurements from the parallel download microbenchmark presented in 5.5.3.1, this time looking at the time spent in the download functions (see Figure 5.19). All platforms show a relatively constant critical path for downloads up to 2^{20} bytes but demonstrate slowdowns for larger downloads up to 2^{27} bytes. All platforms seem to have optimizations in place that reduce download times again for downloading 2^{28} bytes. Azure demonstrates the fastest download time of the three platforms, starting at 0.05s for downloading 2^{10} bytes and showing a maximum critical path of 3.95s. AWS and Google Cloud, on the other hand, already take 1.16s and 1.27s for downloading 2^{10} bytes, respectively. The maximum time of 4.43s on AWS is comparable to Azure. However, Google Cloud takes 11.43s at the longest, 2.58× more than AWS and 2.89× more than Azure. While Google Cloud demonstrates the slowest critical path throughout all benchmarks, only some benchmarks download a considerable amount of data, accounting for part of the slowdown. Thus, the duration of data downloads can only account for part of the overhead.
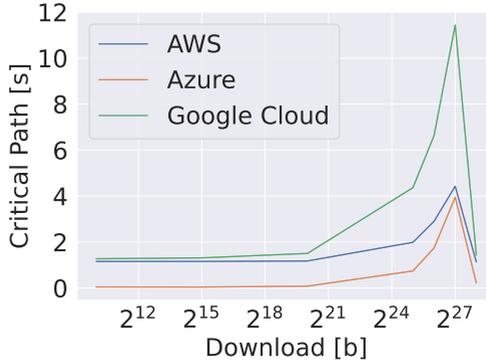
Figure 5.19.: Average time for downloading D bytes per function, 20 parallel functions, $2^{10} \leq D \leq 2^{28}$, 512MB.

OS Noise. The cloud provider controls the CPU allocation to a serverless function, either in relation to the memory configuration on AWS and GCP [60, 98] or in an undisclosed fashion on Azure. We use the selfish detour benchmark to quantify OS noise [79]. This allows us to estimate how long the function is suspended by the OS, which in turn approximates the vCPU timeshare. The benchmark runs a tight loop and records the event that one iteration took significantly more cycles. It stops once it recorded this event N times. The magnitude and frequency of these events characterize the suspension and noise. We deploy a workflow with a single function executing the benchmark, invoke it 30 times concurrently, collect $N = 5000$ events, and sample warm invocations only to obtain consistent results. Figure 5.21 compares the relative to the expected suspension time according to the cloud documentation. We observe generally less noise on Google Cloud when compared to AWS, with more than 20% difference on 1024MB of memory. While the noise on Google Cloud is less than suggested by the documentation except for configurations with 2048MB of memory, the noise on AWS is consistently higher than suggested by the documentation. We normalize the critical path per platform using the following approximation: given a function with memory configuration M, we represent the relative duration of function suspension as S_M and compute the normalized critical path T_C' as follows:

$$T_C' = T_C * (1 - S_M)$$

We observe the largest relative discrepancy on two benchmarks, MapReduce (Figure 5.22a) and Machine Learning (Figure 5.22b). The overall trend observed in Section 5.5.2 remains unchanged: Google Cloud demonstrates the longest critical path duration. The suspension time explains the shorter critical path on Azure as compared to AWS and GCP for benchmarks with low-memory configurations: Azure functions receive larger CPU allocations.

Cold Starts. Cold invocations add significant overhead to the function execution [40]. Table 5.3 presents the frequency of cold starts encountered when executing 30 workflow invocations simultaneously, with cold starts identified using the containerID (see Section 5.2.3). AWS shows the most cold starts, with 100% cold starts for the MapReduce and Machine Learning benchmark and only down to 73.58% cold starts for the ExCamera benchmark. Google Cloud also shows a high percentage of cold starts, ranging from 68.17% for the MapReduce benchmark up to 99.26% for the Machine Learning benchmark. Azure Durable exhibits significantly less cold starts with a maximum of 7.72% for the 1000Genome benchmark, likely because function apps on Azure can hold many invocations concurrently [40]. While the low scalability causes high orchestration overheads on Azure, it benefits the actual computations by putting them in warm containers. Due to the high percentage of cold starts in our measurement data on AWS and Google Cloud, we collected another 60 workflow invocations for AWS and GCP, where at least one function is warm, and plotted the critical path for the resulting completely warm invocations. Figure 5.20 shows the impact of cold starts on the critical path and overhead of the Machine Learning and MapReduce benchmark. Google Cloud and AWS functions perform up to 7.9× and 12.7× better, respectively, achieving almost the same performance measured on Azure. Thus, cold starts are a major factor influencing the slowdown and performance instability observed in many benchmarks.

Conclusions. The microbenchmarks demonstrate that the shorter critical path on Azure for low-memory function configurations is explained by larger CPU allocations, and that the time spent within a function for downloading data explains longer critical paths on Google Cloud for benchmarks with high data downloads. Moreover, we show that cold starts are a major factor influencing the critical path of benchmarks.

| | Cold starts | | |
Benchmark	AWS	GCP	Azure
Video	86.94%	68.61%	3.89%
MapReduce	100%	68.17%	1.0%
Trip Booking	100%	38.24%	0.6%
ExCamera	73.58%	69.34%	0.94%
ML	100%	99.26%	2.60%
1000Genome	98.16%	72.40%	7.72%

Table 5.3.: The relative number of cold starts per benchmark.

(a) Machine Learning, 1024MB.

(b) MapReduce, 256MB.
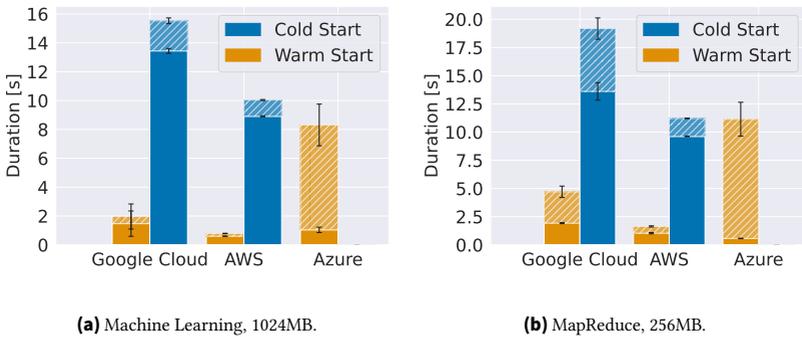
Figure 5.20.: Critical path (opaque) and overhead (shaded) of completely warm and cold invocations.

5.5.4. RQ3.3: Usability for Scientific Workflows

We use the scientific benchmark *1000Genome* to compare cloud services and the HPC system Ault using nodes equipped with Intel(R) 6154@3.00GHz CPU, repeating measurements five times.

First, we compare the runtime of the total workflow, as shown in Figure 5.23a. While the workflow execution time is, on average, 457.7s and 259.8s on GCP and AWS, respectively, the execution takes only 7.7s on Ault. GCP exhibits a coefficient of variation of 12.2%, while AWS has a coefficient of variation of
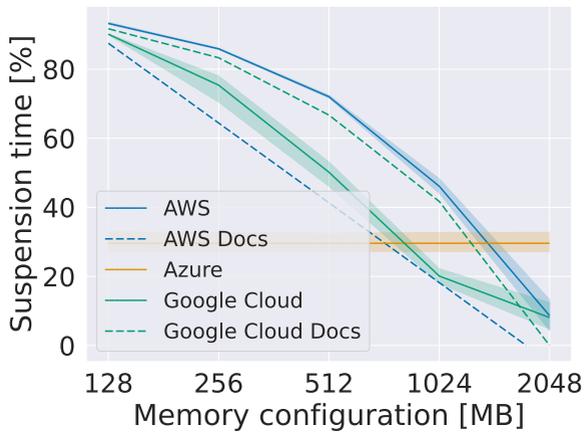
Figure 5.21.: Analysis of OS noise: Relative suspension time of functions, $N = 5000$, *warm* invocations. Light area shows the 95% confidence interval.
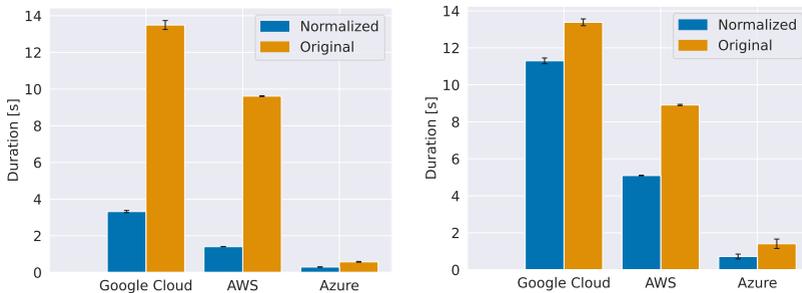
(a) Normalized critical path of MapReduce, 256MB, *burst* invocations.

(b) Normalized critical path of ML, 1024MB, *burst* invocations.

Figure 5.22.: Analysis of OS noise.

only 3.3% - even lower than 4.1% on Ault. Interestingly, I/O takes less than one second on AWS, meaning that the computation is slower in the cloud [1].

---

[1] Note that the parallelism during the data download is less than during our parallel download micro-benchmark, which makes the I/O times not directly comparable.

(a) Complete workflow.
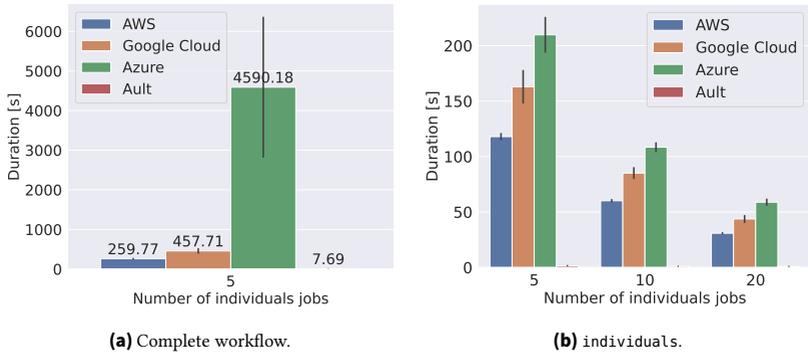
(b) `individuals`.

Figure 5.23.: Scalability of the 1000Genome workflow.

Then, we compare the scaling behavior of the different platforms for the `individuals` task of the workflow. We employ strong scaling, i.e., adding more jobs while keeping the size of the input file the same, resulting in smaller chunks per job. Therefore, optimal scaling would halve the execution time for double the amount of tasks. Figure 5.23b shows the speedup of 1.96 and 1.95 on AWS, 1.91 and 1.95 on GCP, and 1.51 and 1.24 on Ault for 10 and 20 jobs, respectively. The cloud platforms achieve a nearly optimal speedup, which is not surprising given the high overhead for the baseline execution. Still, the execution time of the complete workflow takes at least 33.8× more time than on Ault, showing that the performance of the cloud platforms' workflow orchestration is not competitive with the performance of HPC systems.

While it is not surprising that serverless workflow orchestration is slower than executing the same workflow on an HPC system, this result highlights the need for specialized solutions, such as scientific workflow management systems optimized for serverless environments. Despite the slower execution, serverless computing remains appealing to domain scientists due to its pay-as-you-go billing model, flexibility, and the ability to scale resources infinitely, making it accessible to users who lack on-site computational infrastructure.

5.5.5. RQ3.4: Pricing

We compare the average cost of executing the workflows and estimate the prices, as shown in Table 2.2, p. 30. Functions invoked during the execution of a workflow are billed based on the integral of memory and duration. For state machine workflows, as on AWS and Google Cloud, additional costs per state transition apply. Figure 5.24 visualizes the average cost per 1000 workflow executions split into two groups: function execution (opaque) and the cost of orchestrating the state machine (shaded). Note that, due to Azure's billing and measurement system, we could only retrieve an average cost value over all workflow invocations. Even though the Trip Booking benchmark is a simple pipeline with error catching, running it with workflow orchestration still adds significant state transition costs. Azure is the most expensive service for the 1000Genome benchmark. Google Cloud is the most expensive for MapReduce due to the high number of state transitions required. AWS Step Functions are the most expensive solution for the other four benchmarks because functions cost 6.7× more for computation than Google Cloud Functions. Azure is the cheapest platform for the Trip Booking, Machine Learning, MapReduce, and ExCamera benchmarks, while Google Cloud is the cheapest platform for the remaining two benchmarks, Video Analysis and 1000Genome. The cost for state transitions is nearly identical between AWS and Google Cloud, even though AWS charges 2.5× more per transition: the AWS state language requires fewer states to implement the benchmarks (Table 5.4).

In addition to execution and orchestration costs, workflows can generate charges when accessing the object and NoSQL storage. In all three clouds, the prices of read and write operations on the object storage are exactly the same. However, the billing models for key-value storage differ: DynamoDB charges for operations according to the amount of data read and written in strictly defined size increments; CosmosDB applies the same pricing to request units but does not explicitly define expected consumption; and Datastore has higher costs per operation but makes the cost independent of the item size. To understand the impact of price differences, we analyze the full execution of the Trip Booking benchmark. One workflow invocation requires three insertions and three deletions, with all items taking at most a few hundred bytes. While the estimated storage costs are similar on each platform, between ¢0.68 and ¢1.08 for one thousand executions, they impact the final cost differently. NoSQL operations add only 2.74% and 6.72% of the total price on AWS and GCP, respectively. The total execution cost on Azure is just ¢2.4. There, the
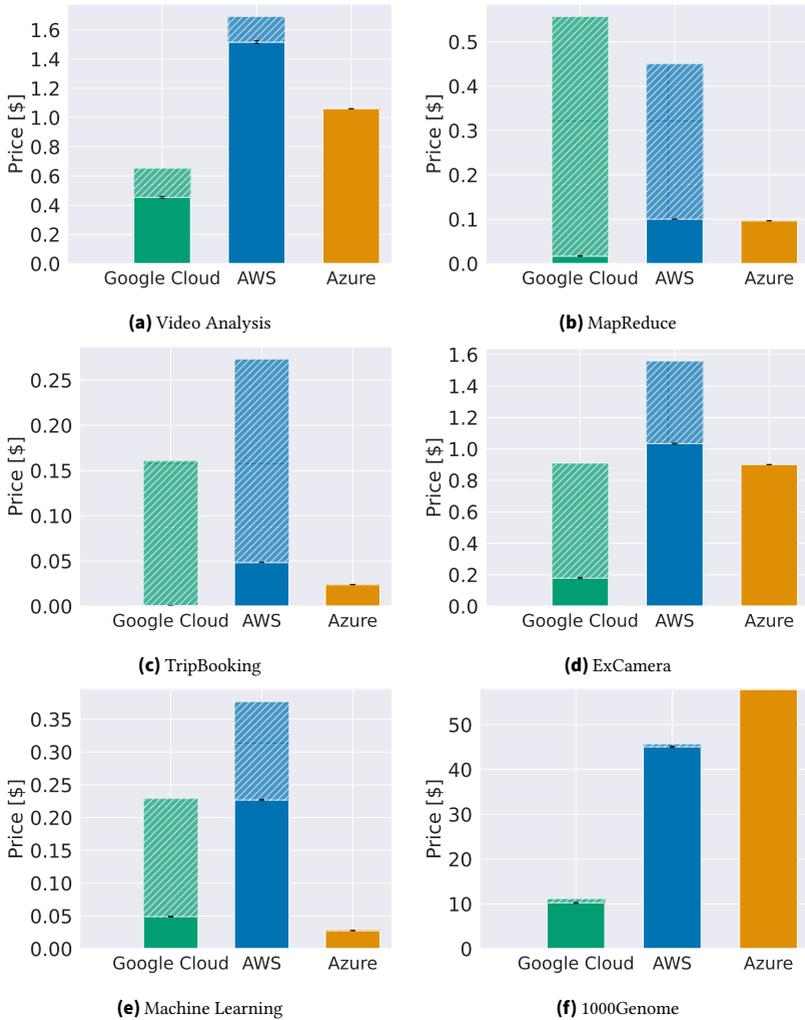
Figure 5.24.: Price per 1000 workflow executions: function costs are opaque and state transition costs are translucent.

estimated cost of CosmosDB request units is equal to ¢0.68 and adds 28.5% of workflow price.

Platform	Video	MapReduce	Trip Booking	ExCamera	ML	1000Genome
AWS	7	14	9	21	6	26
GCP	20	54	16	73	18	96

Table 5.4.: The number of state transitions for AWS and GCP per benchmark.

(a) MapReduce.

(b) ML.
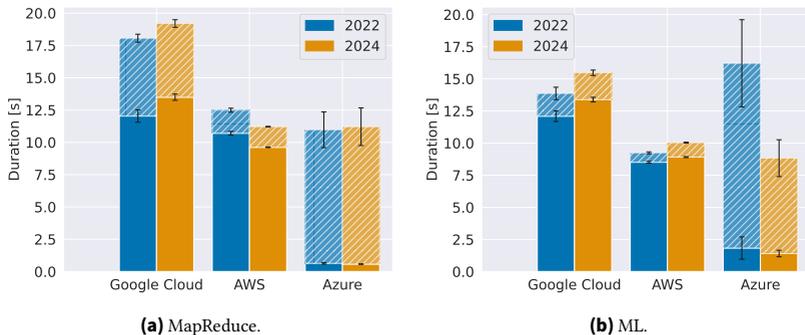
Figure 5.25.: Comparison of critical path (opaque) and overhead (shaded) between 2022 and 2024, *burst* invocations.

5.5.6. RQ3.5: Evolution of Performance

Finally, we assess the performance stability over time by comparing measurements from July 2022 and January 2024. The executions from 2022, conducted as part of the preceding master thesis [24], contain 30 invocations per workflow using Python 3.7, in cloud regions *europe-west* for Azure, *europe-west-1* for GCP, and *us-east-1* for AWS. We run the 2024 invocations in the same regions, except for GCP in *us-east1*, and use Python 3.8, as the platforms already deprecated Python 3.7. Figure 5.25 shows the results. The critical path and overhead of the MapReduce and ML benchmark are approximately the same on Google Cloud. The runtime on AWS is relatively stable without any notable differences between 2022 and 2024. Azure has a stable duration of the critical path. While the overhead for MapReduce on Azure is the same in 2024 as in 2022, the overhead of ML has been approximately halved from 2022 to 2024.

5.5.7. Threats to Validity

A threat to the external validity is our choice of benchmark applications. We mitigate this by using applications from different domains that correspond to previous findings on the characterization of workflow use cases [48, 49].

For the internal validity, the quality of our measurements is important. Here, the different geographical regions and weekdays on which we conducted our measurements could have an impact. First experiments showed that there is performance variability on Google Cloud Functions depending on the time of day [132]. While systematically investigating this is beyond the scope of our work, we are confident that our measurements can provide a reliable base for characterizing the performance of the different platforms: As detailed in Section 5.5.1, we repeat each application measurement six times to obtain stable results, resulting in 180 workflow executions. We conduct the repetitions directly after another and update the functions in-between repetitions to enforce a realistic distribution of cold and warm starts, and avoid order effects between repetitions. Also, the results from RQ3.5 in Section 5.5.6 showed that the measurement results from 2022, conducted on different weekdays and times of days as our 2024 measurements, are mostly comparable and therefore reassure our confidence in the reliability of our measurements.

5.6. Related Work

Multiple benchmark suites have been proposed to cover different aspects of serverless computing, from microarchitecture to the application level [12, 40, 50, 91, 104, 138, 151]. However, all of them consider only the execution of single functions. Das et al. [41] benchmark serverless edge computing platforms. Other performance studies of serverless applications focus on non-workflow orchestration systems, e.g., using cloud storage and queue triggers [65, 75, 130, 143]. Grambow et al. [65] propose BeFaaS, providing an application benchmark modeling an online shop where the functions communicate using synchronous and asynchronous calls. In contrast, we target serverless workflow orchestrations with *SeBS-Flow*.

ServerlessBench [167] considers a function chain microbenchmark orchestrated by AWS Step Functions, but only measures runtime of the workflow and time in between function invocations for varying payload sizes. Kousiouris

et al. [95] use microbenchmarks to estimate the overhead of orchestration in OpenWhisk. López et al. [56] investigate the orchestration overhead with microbenchmarks of function chains and parallel functions. Shahidi et al. [137] evaluate the performance and cost of two stateful workflows on AWS and Azure. Barcelona-Pons et al. [15] use a microbenchmark to test the performance of fork-join parallelism in workflow orchestrators. With *SeBS-Flow*, we provide not only microbenchmarks, but also six applications from different domains that can automatically be deployed to different cloud platforms. Based on these benchmarks, we present a broader evaluation of the performance of cloud platforms.

Wen et al. [162] conducts a performance investigation of serverless workflows using two applications and microbenchmarks with varying numbers of functions, payload size, and parallelism. They measure the execution time and estimate overhead. They show that while AWS Step Functions achieves a shorter total runtime for their applications, Azure Durable Functions has a shorter critical path and its runtime is dominated by overhead. This is consistent to our findings. However, for their micro-benchmarks, they find that Azure Durable Functions has a lower total runtime for parallel function executions. In our experiments, Azure Durable Functions experiences an order of magnitude higher overhead when executing functions in parallel. As they conducted their measurements in 2020, this difference in observations could be caused by evolution of the platforms. Wen et al. do not investigate different sources of overhead, and do not evaluate scalability nor billing. In contrast to them, we focus on a wider collection of applications and propose a unifying model that allows us to deploy and evaluate a single implementation across many cloud platforms. Furthermore, we make all benchmark codes available and integrated into an established benchmarking suite for easy deployment and reproducibility. Finally, we evaluated serverless Google Cloud Workflows instead of the non-serverless Google Cloud Composer. XF-Bench [96] provides chaining of different functions and deploying them to AWS Step Functions and Azure Durable Functions, while we focus on realistic and complete applications. Moreover, they do not consider cloud-native data movement between functions via cloud storage, do not evaluate the overhead of their platform transcription, and can not compare pricing between platforms.

Other authors analyzed the productivity of workflow languages and proposed alternative models. AFCL [122] is a custom and provider-independent orchestration language for serverless workflows. Their goal is to speed up

the definition of workflows and to replace the workflow orchestration provided by the platforms by scheduling the functions themselves, while our workflow definition transcribes to the workflow representations of different platforms. Burckhardt et al. explore the semantics of Durable Functions [27] and propose Netherite [26], a new engine to replace Azure Durable Functions. Their goal is to increase the efficiency of workflow engines, while we aim to benchmark their performance. Versluis et al. [155] investigate the support for annotating non-functional requirements in different workflow formalisms. Moreover, they compare the different formalisms regarding their suitability for extension to support non-functional requirements. Different to them, we are not interested in annotating non-functional requirements but want to express control-flow and data-flow of workflows.

5.7. Summary

We propose *SeBS-Flow*, the first benchmark suite for serverless workflows. We follow the established benchmark design principles: introduce a platform-agnostic workflow model, propose a collection of six representative applications, and integrate them into an existing benchmark suite to ensure reproducibility and ease of use. We support the three major cloud providers AWS Step Functions, Google Cloud Workflows, and Azure Durable Functions, and offer extension possibilities: Benchmarks can be ported to other services by implementing a single interface transcribing our model to the cloud-specific interface.

We first evaluated our workflow model regarding its expressiveness and the introduced overhead when transcribing it to different cloud providers by reviewing existing literature on serverless workflows. We found that our model does not have general limitations in expressiveness and does not introduce noteworthy overhead compared to native implementations, both with regard to the workflows used in literature.

Then, we use *SeBS-Flow* to conduct a comprehensive and long-term evaluation of the performance and cost of the proposed benchmark applications, investigating factors influencing the runtime and variance: cold startups, noise, scheduling overhead, and the storage I/O. With the new benchmark suite, we enable benchmarking of the same workflow on different platforms, providing software developers and researchers with valuable insights regarding their different behaviors and properties.

Part III.

Epilogue

6. Conclusions

In this chapter, we conclude the thesis. We summarize our contributions and their validation in Section 6.1. Section 6.2 presents the benefits of our contributions for various stakeholders. We then recap the most important assumptions and limitations in Section 6.3.

6.1. Summary

In this thesis, we presented three validated contributions that answer the research questions defined in Section 1.3. In the following, we summarize the contributions, the corresponding research questions, and the evaluation.

Experiment Design for Automatic Performance Modeling We introduce *Performance-Detective*, a new white-box measurement method that uses deductive analysis based on the results of a taint analysis to find an optimized, minimal set of required measurements from a complex configuration space (C1). Unlike traditional methods that use heuristics, *Performance-Detective* uses program information to precisely remove unnecessary measurement points. We use two central insights to reduce measurement points, answering RQ1: First, if parameters influence distinct sets of functions, we do not need to measure configurations aimed at finding interactions between them. Second, we not only do not need to include parameters in the experiment design that influence the computation linearly, but can also skip repetitions of experiments. This reduces the dimensionality of experiments, making performance modeling more affordable without losing accuracy. Our evaluation with the Pace3D multi-physics solver and the particle transport application Kripke show that *Performance-Detective* can reduce the cost of required measurements significantly while maintaining the model accuracy, making it a powerful tool for efficient performance modeling in scientific applications.

Identification of Performance-Irrelevant Options   Contribution C2 addresses the complexity of performance modeling by introducing a new pre-processing step that automatically finds and removes performance-irrelevant configuration options, answering RQ2. This is a three-step process: After conducting small-scale measurements, we build a preliminary performance model. Examining this model, we can identify which options it uses to make predictions and which options are not used and therefore do not have a performance impact. By filtering out performance-irrelevant options, our method reduces the cost and effort needed to create performance models. Our evaluation shows that this approach can effectively distinguish into relevant and irrelevant options without mistakenly excluding important ones, thus simplifying the modeling process by making it more accessible and less costly.

Benchmarking of Serverless Workflows   Contribution C3 offers *SeBS-Flow*, the first benchmark suite designed specifically for serverless workflows. To answer RQ3, we propose a workflow model based on the Petri Net formalism that shows control- and data-flow explicitly and can be transcribed to the interfaces of multiple cloud platforms. This allows users to seamlessly run nearly the same workloads across different systems. Reviewing existing publications on serverless workflows with benchmarks, we show that our transcription to the provider-specific implementations adds no substantial overhead. Our benchmarking suite includes six real-world application benchmarks and four microbenchmarks, covering a range of computational tasks such as machine learning, multimedia processing, and scientific applications. With *SeBS-Flow*, we provide a consistent and reproducible method for evaluating and comparing the performance, cost, scalability, and stability of serverless workflows platforms. Our evaluation highlights key factors that affect runtime and variability, such as cold startups, noise, scheduling delays, and storage I/O, offering valuable insights into the performance of serverless programming models and cloud systems.

6.2. Benefits

The contributions presented in this thesis offer numerous benefits tailored to the needs of domain scientists and software engineers, easing performance

modeling, optimization, and benchmarking across diverse computing platforms.

Increased Efficiency and Reduced Costs. The automatic identification and exclusion of performance-irrelevant configuration options streamline the performance modeling process, reducing both time and computational costs. This makes advanced performance modeling techniques more accessible to domain scientists, even those without deep expertise in performance engineering. By utilizing deductive analysis to minimize the number of required measurements, *Performance-Detective* considerably reduces experimentation costs. This efficiency enables domain scientists to allocate more resources to their core research activities, making the performance modeling process less resource-intensive and more cost-effective.

Ease of Use and Accessibility. Building on existing tools for automatic performance modeling, the contributions of this thesis simplify the performance modeling process, therefore making it more user-friendly for domain scientists. By removing the need for performance engineering knowledge, the use of advanced performance modeling techniques is democratized, allowing a broader range of scientists to benefit from these tools. Moreover, we built a ready-to-use benchmarking platform for the performance analysis of serverless workflows, hiding the complexity associated with benchmarking from developers and scientists.

Comprehensive Benchmarking. By abstracting differences between various cloud platforms away, our workflow model allows for seamless benchmarking and comparison of workflow performance across different serverless platforms. Our benchmark suite for serverless workflows provides a standardized and reproducible way to evaluate and compare serverless platforms.

6.3. Assumptions and Limitations

Drawing from the discussions about assumptions and limitations in Section 3.2.5 and Section 4.2.4, as well as about the respective threats to validity

in Section 3.4.4, Section 4.4.3, Section 5.4.3, and Section 5.5.7, we recap the most important assumptions and limitations of our contributions.

Representative computation for small problem instances. For both contributions C1 and C2, we assume that the problem size can be scaled down while the computation is still representative. For C1, the cost of the taint analysis depends on the input, as the dynamic part of the analysis executes the software to derive the parametric profile. If it is not possible to scale down the problem size without affecting the computation, the taint analysis will have a higher cost. For C2, the cost of our small-scale measurements depend on the input size, therefore affecting the total cost of our pre-processing step. However, if measurements are taken with bigger problem sizes for our pre-processing step, they could also be re-used for the principal performance modeling, therefore balancing the additional cost incurred.

Location of Performance-Relevant Behavior. A limitation of *Performance-Detective* is introduced through its reliance on Perf-Taint for the system analysis. As Perf-Taint assumes that performance-relevant behavior is located in computational loops and MPI communication routines, the parametric profile created only contains parameter dependencies for these cases. Moreover, this means that we do not regard binary parameters, such as the choice of algorithm.

Selection of Benchmarks. One limitation of *SeBS-Flow* is the choice of benchmarks. Even though we offer six benchmarks from different domains and with different characteristics, this selection can affect the results. It may not fully reflect the variety and complexity of real-world workflows, which could limit how general our findings are. Therefore, while our benchmarks aim to provide a broad evaluation, they may not cover all the details and performance traits found in actual serverless workflows. A limitation of the evaluation of the overhead of our workflow model is the overall low number of available implementations of workflows and the absence of implementations using Google Cloud Workflows in the reviewed literature. While we follow best practices as provided by the cloud providers, we can not rule out the possibility of projects orchestrating their workflows differently.

7. Future Work

In this chapter, we discuss possible directions for future research regarding each of our contributions.

Experiment Design for Automatic Performance Modeling. An interesting direction for future work is to demonstrate how more than three parameters can be modeled effectively by using taint analysis and our experiment design deduction. Previous studies have indicated that modeling four or more parameters can be difficult due to noise, which makes it hard to model parameter dependencies accurately except for the most important effects. This modeling of noise can lead to misrepresentations about how different parameters interact. By isolating the influence and interactions of parameters, we can improve our ability to model more parameters and gain a clearer understanding of their relationships.

Additionally, an interesting extension of *Performance-Detective* could be the support of binary parameters. As binary parameters usually determine if a specific piece of code is executed and switching between them results in performance jumps. This means that different performance models need to be created for the numerical options influencing the performance of the (de-)activated piece of code. Nevertheless, by using an analysis that can work with binary parameters, such as the analysis used by Velez et al. [154], we could locate the area influenced by a specific binary parameter and deduce a minimal experiment design for creating performance models incorporating the effect of the binary parameters.

Identification of Performance-Irrelevant Options. In future work, the evaluation of our approach with more case studies will provide more insights about its general applicability. Moreover, different performance modeling methods can be compared to see how well they are suited to identify performance-irrelevant options using small samples. For example, DeepPerf [71] and PERF-AL [139] use deep neural networks. These methods may need fewer samples even though they take longer to train than DECART.

Additionally, building black-box principal performance models iteratively with and without filtering the options identified as performance-irrelevant by our approach for the sampling designs can provide further insight on the advantages of our approach. By building performance models from both sets of samples, we can quantify how much faster the predictive quality of the model improves by ignoring irrelevant options. Moreover, in conjunction with the extension of *Performance-Detective* to handle binary parameters, we can also evaluate how much we can save by filtering performance-irrelevant options as compared to the experiment design deduced by *Performance-Detective*.

Another direction is leveraging monitoring data from production runs of the system to identify performance-irrelevant options instead of our dedicated small-scale experiments. This would require the monitoring data to contain not only measurement data, but also the provided configurations and hardware used for different runs. Also, the problem of missing repetitions of the same configuration could arise, which are important to ensure the reliability of the end-to-end measurements. Combining this approach with results derived by documentation-based approaches like SafeTune [76] to specifically evaluate options identified as important by them can further reduce the cost of performance modeling.

Benchmarking of Serverless Workflows.   Future work can add transcription of our workflow model to more serverless platforms. This will help developers to understand the strengths and weaknesses of additional platforms better and enable comparison between them.

Also, including more benchmarks could help in generalizing our findings. Especially benchmarks from different scientific fields would be interesting to further investigate differences in platform suitability for these types of workloads. Additionally, our workflow model could be used for transcription to the representation of different scientific workflow management systems to enable fair comparisons between them.

Moreover, to further evaluate our workflow model, repositories on GitHub and gray literature could be searched to analyze more existing implementations and unpublished studies to compare our model and transcribed workflows with.

Bibliography

[1]   I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. "SAND: Towards High-Performance Serverless Computing". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference.* USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 923–935.

[2]   J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. "OpenTuner: An extensible framework for program autotuning". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT).* 2014, pp. 303–315. DOI: 10.1145/2628071.2628092.

[3]   S. Apel, D. Batory, C. Kästner, and G. Saake. "Basic Concepts, Classification, and Quality Criteria". In: *Feature-Oriented Software Product Lines: Concepts and Implementation.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 47–63. DOI: 10.1007/978-3-642-37521-7_3.

[4]   *Iterating a Loop Using Lambda.* https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html. Accessed 25-01-2024.

[5]   *AWS Lambda Functions Powered by AWS Graviton2 Processor.* https://aws.amazon.com/blogs/aws/aws-lambda-functions-powered-by-aws-graviton2-processor-run-your-functions-on-arm-and-get-up-to-34-better-price-performance/. Accessed 25-01-2024.

[6]   *AWS Lambda Pricing.* https://aws.amazon.com/lambda/pricing/. Accessed 25-01-2024.

[7]   *AWS Step Functions.* https://aws.amazon.com/step-functions/. Accessed 25-01-2024.

[8]   *AWS Step Functions Pricing.* https://aws.amazon.com/step-functions/pricing/. Accessed 25-01-2024.

[9]   *Azure Durable Functions.* https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview. Accessed 25-01-2024. 2019.

[10]   *Azure Functions Pricing.* https://azure.microsoft.com/en-us/pricing/details/functions/. Accessed 25-01-2024.

[11]   *Azure: Migrate apps from Azure Functions version 3.x to version 4.x.* https://learn.microsoft.com/en-us/azure/azure-functions/migrate-version-3-version-4. Accessed 25-01-2024.

[12]   T. Back and V. Andrikopoulos. "Using a Microbenchmark to Compare Function as a Service Solutions". In: *Service-Oriented and Cloud Computing*. Springer International Publishing, 2018, pp. 146–160. DOI: 10.1007/978-3-319-99819-0_11.

[13]   P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. "Autotuning in High-Performance Computing Applications". In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083. DOI: 10.1109/JPROC.2018.2841200.

[14]   P. Balaprakash, S. M. Wild, and P. D. Hovland. "An Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning". In: *High Performance Computing for Computational Science - VECPAR 2012*. Ed. by M. Daydé, O. Marques, and K. Nakajima. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 261–269.

[15]   D. Barcelona-Pons, P. García-López, Á. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas. "FaaS Orchestration of Parallel Workloads". In: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 25–30. DOI: 10.1145/3366623.3368137.

[16]   A. Barker and J. Van Hemert. "Scientific workflow: a survey and research directions". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer. 2007, pp. 746–753.

[17]   B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. "A Regression-Based Approach to Scalability Prediction". In: *Proceedings of the 22nd Annual International Conference on Supercomputing*. ICS '08. Island of Kos, Greece: Association for Computing Machinery, 2008, pp. 368–377. DOI: 10.1145/1375527.1375580.

[18]   T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler. "A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 66–77. DOI: `10.1109/IPDPS.2019.00018`.

[19]   S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. "Characterization of scientific workflows". In: *2008 third workshop on workflows in support of large-scale science*. IEEE. 2008, pp. 1–10.

[20]   A. Bhattacharyya and T. Hoefler. "PEMOGEN: Automatic Adaptive Performance Modeling During Program Runtime". In: *Proc. of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14. Edmonton, AB, Canada: ACM, 2014, pp. 393–404. DOI: `10.1145/2628071.2628100`.

[21]   A. Bhattacharyya, G. Kwasniewski, and T. Hoefler. "Using Compiler Techniques to Improve Automatic Performance Modeling". In: *Proc. of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT'15)*. San Francisco, CA, USA, 2015, pp. 1–12.

[22]   C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. "How is the Weather Tomorrow?: Towards a Benchmark for the Cloud". In: *Proceedings of the Second International Workshop on Testing Database Systems*. DBTest '09. Providence, Rhode Island: ACM, 2009, 9:1–9:6. DOI: `10.1145/1594156.1594168`.

[23]   C. Bischof, D. an Mey, and C. Iwainsky. "Brainware for green HPC". In: *Computer Science - Research and Development* 27.4 (Nov. 1, 2012), pp. 227–233. DOI: `10.1007/s00450-011-0198-5`.

[24]   L. Brandner. "A Platform-Agnostic Model and Benchmark Suite for Serverless Workflows". en. Master Thesis. Zurich: ETH Zurich, 2022. DOI: `10.3929/ethz-b-000574821`.

[25]   L. Breiman. "Random Forests". English. In: *Machine Learning* 45.1 (2001), pp. 5–32. DOI: `10.1023/A:1010933404324`.

[26]   S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu. "Netherite: Efficient Execution of Serverless Workflows". In: *Proc. VLDB Endow.* 15.8 (Apr. 2022), pp. 1591–1604. DOI: `10.14778/3529337.3529344`.

[27] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn. "Durable Functions: Semantics for Stateful Serverless". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: `10.1145/3485510`.

[28] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn. "Serverless Workflows with Durable Functions and Netherite". In: *CoRR* abs/2103.00033 (2021). arXiv: `2103.00033`.

[29] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. "Fast Multi-parameter Performance Modeling". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 2016 IEEE International Conference on Cluster Computing (CLUSTER). Taipei, Taiwan: IEEE, Sept. 2016, pp. 172–181. DOI: `10.1109/CLUSTER.2016.57`.

[30] A. Calotoiu, M. Copik, T. Hoefler, M. Ritter, S. Shudler, and F. Wolf. "ExtraPeak: Advanced Automatic Performance Modeling for HPC Applications". In: *Software for Exascale Computing - SPPEXA 2016-2019*. Ed. by H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel. Lecture Notes in Computational Science and Engineering. Cham: Springer International Publishing, 2020, pp. 453–482. DOI: `10.1007/978-3-030-47956-5_15`.

[31] A. Calotoiu, A. Graf, T. Hoefler, D. Lorenz, S. Rinke, and F. Wolf. "Lightweight Requirements Engineering for Exascale Co-design". In: *Proc. of the 2018 IEEE International Conference on Cluster Computing (CLUSTER), Belfast, UK*. IEEE, Sept. 2018, pp. 201–211. DOI: `10.1109/CLUSTER.2018.00038`.

[32] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. "Using automated performance modeling to find scalability bugs in complex codes". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ISSN: 2167-4337. Nov. 2013, pp. 1–12. DOI: `10.1145/2503210.2503277`.

[33] Z. Cao, G. Kuenning, and E. Zadok. "Carver: Finding Important Parameters for Storage System Tuning". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 43–57.

[34] M. Cashman, M. B. Cohen, P. Ranjan, and R. W. Cottingham. "Navigating the maze: the impact of configurability in bioinformatics software". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Montpellier, France: Association for Computing Machinery, 2018, pp. 757–767. DOI: `10.1145/3238147.3240466`.

[35] R. Chatley and T. Allerton. "Nimbus: improving the developer experience for serverless applications". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 85–88. DOI: `10.1145/3377812.3382135`.

[36] Z. Chen, P. Chen, P. Wang, G. Yu, Z. He, and G. Mai. "DiagConfig: Configuration Diagnosis of Performance Violations in Configurable Software Systems". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 566–578. DOI: `10.1145/3611643.3616300`.

[37] M. Copik, A. Calotoiu, T. Grosser, N. Wicki, F. Wolf, and T. Hoefler. "Extracting Clean Performance Models from Tainted Programs". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 403–417. DOI: `10.1145/3437801.3441613`.

[38] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler. "Software Resource Disaggregation for HPC with Serverless Computing". In: *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2024, pp. 139–156. DOI: `10.1109/IPDPS57955.2024.00021`.

[39] M. Copik and T. Hoefler. "perf-taint: Taint Analysis for Automatic Many-Parameter Performance Modeling". In: *ACM Student Research Competition at ACM/IEEE Supercomputing* (2019).

[40] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler. "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing". In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 64–78. DOI: `10.1145/3464298.3476133`.

[41]   A. Das, S. Patterson, and M. Wittie. "EdgeBench: Benchmarking Edge Computing Platforms". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, Dec. 2018. DOI: `10.1109/ucc-companion.2018.00053`.

[42]   Datadog. *The State of Serverless*. `https://www.datadoghq.com/state-of-serverless/`. Accessed: 2024-01-28. 2024.

[43]   *Clang 9 Documentation - DataFlowSanitizer*. `https://clang.llvm.org/docs/DataFlowSanitizer.html`. Accessed 2024-10-06.

[44]   O. Dieste, A. Grimán, and N. Juristo. "Developing search strategies for detecting relevant experiments". In: *Empirical Software Engineering* 14 (2009), pp. 513–539.

[45]   J. Dominguez-Trujillo, K. Haskins, S. J. Khouzani, C. Leap, S. Tashakkori, Q. Wofford, T. Estrada, P. G. Bridges, and P. M. Widener. "Lightweight Measurement and Analysis of HPC Performance Variability". In: *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2020, pp. 50–60. DOI: `10.1109/PMBS51919.2020.00011`.

[46]   D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 467–481. DOI: `10.1145/3373376.3378512`.

[47]   S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1 (2021), pp. 32–39. DOI: `10.1109/MS.2020.3023302`.

[48]   S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". In: *IEEE Transactions on Software Engineering* 48.10 (2022), pp. 4152–4166. DOI: `10.1109/TSE.2021.3113940`.

[49]   S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. "A review of serverless use cases and their characteristics". In: *arXiv preprint arXiv:2008.11110* (2020).

[50]   *FaaSTest*. `https://github.com/nuweba/faasbenchmark`. Accessed: 2020-08-01.

[51]   D. Fernandez-Amoros, R. Heradio, C. Mayr-Dorn, and A. Egyed. "Scalable Sampling of Highly-Configurable Systems: Generating Random Instances of the Linux Kernel". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. DOI: `10.1145/3551349.3556899`.

[52]   R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman. "A characterization of workflow management systems for extreme-scale applications". In: *Future Generation Computer Systems* 75 (2017), pp. 228–238. DOI: `https://doi.org/10.1016/j.future.2017.02.026`.

[53]   S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488.

[54]   S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376.

[55]   Y. Gan, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, Y. Zhang, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, and B. Ritchken. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*. ACM Press, 2019. DOI: `10.1145/3297858.3304013`.

[56]   P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, and D. Arroyo Pinto. "Comparison of FaaS Orchestration Systems". In: *2018 IEEE/ACM International Conference*

on Utility and Cloud Computing Companion (UCC Companion). 2018, pp. 148–153. DOI: `10.1109/UCC-Companion.2018.00049`.

[57] H. Garcia-Molina and K. Salem. "Sagas". In: *SIGMOD Rec.* 16.3 (Dec. 1987), pp. 249–259. DOI: `10.1145/38714.38742`.

[58] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. "Examining the challenges of scientific workflows". In: *Computer* 40.12 (2007), pp. 24–32.

[59] D. Gohman. *ScalarEvolution and Loop Optimization.* Talk at LLVM Developer's Meeting. 2009.

[60] *Cloud Functions Pricing.* `https://cloud.google.com/functions/pricing`. Accessed 25-01-2024.

[61] *Invoke Cloud Functions or Cloud Run.* `https://cloud.google.com/workflows/docs/calling-run-functions`. Accessed 25-01-2024.

[62] *Google Cloud Workflows.* `https://cloud.google.com/workflows`. Accessed 25-01-2024.

[63] *Google Cloud Workflows Pricing.* `https://cloud.google.com/workflows/pricing`. Accessed 25-01-2024.

[64] *Google Cloud: Cloud Functions 2nd gen is GA.* `https://cloud.google.com/blog/products/serverless/cloud-functions-2nd-generation-now-generally-available`. Accessed 25-01-2024.

[65] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach. "BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms". In: *2021 IEEE International Conference on Cloud Engineering (IC2E).* 2021, pp. 1–8. DOI: `10.1109/IC2E52221.2021.00014`.

[66] A. Grebhahn, N. Siegmund, and S. Apel. "Predicting Performance of Software Configurations: There is no Silver Bullet". In: *arXiv:1911.12643 [cs]* (Nov. 28, 2019). arXiv: `1911.12643`.

[67] A. Grebhahn, N. Siegmund, H. Köstler, and S. Apel. "Performance Prediction of Multigrid-Solver Configurations". In: *Software for Exascale Computing - SPPEXA 2013-2015.* Ed. by H.-J. Bungartz, P. Neumann, and W. E. Nagel. Lecture Notes in Computational Science and Engineering. Cham: Springer International Publishing, 2016, pp. 69–88. DOI: `10.1007/978-3-319-40528-5_4`.

[68]   S. Gulwani, K. K. Mehra, and T. Chilimbi. "SPEED: precise and efficient static estimation of program computational complexity". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 127–139. DOI: `10.1145/1480881.1480898`.

[69]   J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wąsowski. "Variability-aware performance prediction: A statistical learning approach". In: *ASE'13*. IEEE, Nov. 2013, pp. 301–311. DOI: `10.1109/ASE.2013.6693089`.

[70]   J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu. "Data-Efficient Performance Learning for Configurable Systems". In: *Empirical Softw. Engg.* 23.3 (June 2018), pp. 1826–1867. DOI: `10.1007/s10664-017-9573-6`.

[71]   H. Ha and H. Zhang. "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network". In: *ICSE'19*. Ed. by G. Mussbacher, J. M. Atlee, and T. Bultan. IEEE, May 2019, pp. 1095–1106. DOI: `10.1109/icse.2019.00113`.

[72]   H. Ha and H. Zhang. "Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 470–480. DOI: `10.1109/ICSME.2019.00080`.

[73]   X. Han and T. Yu. "An Empirical Study on Performance Bugs for Highly Configurable Software Systems". In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '16. Ciudad Real, Spain: Association for Computing Machinery, 2016. DOI: `10.1145/2961111.2962602`.

[74]   X. Han, T. Yu, and M. Pradel. "ConfProf: White-Box Performance Profiling of Configuration Options". In: *ICPE*. Ed. by J. Bourcier, Z. M. (Jiang, C.-P. Bezemer, V. Cortellessa, D. D. Pompeo, and A. L. Varbanescu. France: ACM, Apr. 2021, pp. 1–8. DOI: `10.1145/3427921.3450255`.

[75]   R. Hancock, S. Udayashankar, A. J. Mashtizadeh, and S. Al-Kiswany. "OrcBench: A Representative Serverless Benchmark". In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 2022, pp. 103–108. DOI: `10.1109/CLOUD55607.2022.00028`.

[76] H. He, Z. Jia, S. Li, Y. Yu, C. Zhou, Q. Liao, J. Wang, and X. Liao. "Multi-intention-aware configuration selection for performance tuning". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1431–1442. DOI: 10.1145/3510003.3510094.

[77] T. Hoefler and R. Belli. "Scientific Benchmarking of Parallel Computing Systems". In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15). Austin, TX, USA: ACM, Nov. 2015, 73:1–73:12.

[78] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. "Performance modeling for systematic performance tuning". In: *SC'11*. Ed. by S. Lathrop, J. Costa, and W. Kramer. ACM, Nov. 2011, pp. 1–12. DOI: 10.1145/2063348.2063356.

[79] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. "Netgauge: A Network Performance Measurement Framework". In: *Proceedings of High Performance Computing and Communications, HPCC'07*. Vol. 4782. Houston, USA: Springer, Sept. 2007, pp. 659–671.

[80] J. Hötzer, A. Reiter, H. Hierl, P. Steinmetz, M. Selzer, and B. Nestler. "The parallel multi-physics phase-field framework Pace3D". In: *Journal of Computational Science* 26 (2018), pp. 1–12. DOI: https://doi.org/10.1016/j.jocs.2018.02.011.

[81] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee. "An Approach to Performance Prediction for Parallel Applications". In: *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*. Euro-Par'05. Lisbon, Portugal: Springer-Verlag, 2005, pp. 196–205.

[82] P. Jamshidi, N. Siegmund, M. Velez, C. Kästner, A. Patel, and Y. Agarwal. "Transfer learning for performance modeling of configurable systems: An exploratory analysis". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 497–508. DOI: 10.1109/ASE.2017.8115661.

[83] Q. Jiang, Y. C. Lee, and A. Y. Zomaya. "Serverless Execution of Scientific Workflows". In: *Service-Oriented Computing*. Ed. by M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol. Cham: Springer International Publishing, 2017, pp. 706–721.

[84]   D. Jin, M. B. Cohen, X. Qu, and B. Robinson. "PrefFinder: Getting the Right Preference in Configurable Software Systems". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 151–162. DOI: 10.1145/2642937.2643009.

[85]   A. John, K. Ausmees, K. Muenzen, C. Kuhn, and A. Tan. "SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '19 Companion. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 43–50. DOI: 10.1145/3368235.3368839.

[86]   H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. "A multi-objective auto-tuning framework for parallel codes". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–12. DOI: 10.1109/SC.2012.7.

[87]   G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. "Characterizing and profiling scientific workflows". In: *Future generation computer systems* 29.3 (2013), pp. 682–692.

[88]   C. Kaltenecker, A. Grebhahn, N. Siegmund, and S. Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction". In: *IEEE Software* 37.4 (2020), pp. 58–66. DOI: 10.1109/MS.2020.2987024.

[89]   K. Kanellis, R. Alagappan, and S. Venkataraman. "Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs". In: *HotStorage'20*. Ed. by A. Badam and V. Chidambaram. USENIX Association, July 2020.

[90]   D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. "Predictive performance and scalability modeling of a large-scale application". In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. 2001, pp. 37–37.

[91]   J. Kim and K. Lee. "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, July 2019. DOI: 10.1109/cloud.2019.00091.

[92]   J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. "How to Build a Benchmark". In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: ACM, 2015, pp. 333–336. DOI: `10 . 1145 / 2668930.2688819`.

[93]   A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Ed. by H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[94]   S. Kolesnikov, N. Siegmund, C. Kästner, A. Grebhahn, and S. Apel. "Tradeoffs in modeling performance of highly configurable software systems". In: *Software & Systems Modeling* 18.3 (June 2019), pp. 2265–2283. DOI: `10.1007/s10270-018-0662-9`.

[95]   G. Kousiouris, C. Giannakos, K. Tserpes, and T. Stamati. "Measuring Baseline Overheads in Different Orchestration Mechanisms for Large FaaS Workflows". In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. ICPE '22. Bejing, China: Association for Computing Machinery, 2022, pp. 61–68. DOI: `10.1145/3491204.3527467`.

[96]   V. Kulkarni, N. Reddy, T. Khare, H. Mohan, J. Murali, M. A, R. B, S. Balajee, S. S, S. D, V. S, Y. V, C. Babu, A. S. Prasad, and Y. Simmhan. "XFBench: A Cross-Cloud Benchmark Suite for Evaluating FaaS Workflow Platforms". In: *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2024, pp. 543–556. DOI: `10.1109/CCGrid59990.2024.00067`.

[97]   A. J. Kunen, T. S. Bailey, and P. N. Brown. "Kripke - a massively parallel transport mini-app". In: *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (ANS MC '15)*. Nashville, Tennessee, Apr. 2015.

[98]   *Memory and Computing Power*. `https://docs.aws.amazon.com/ lambda/latest/operatorguide/computing-power.html`. Accessed 25-01-2024.

[99]   C. Lattner and V. Adve. "LLVM: A Compilation Framework for Life-long Program Analysis & Transformation". In: *Proc. of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004.

[100] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. "Methods of inference and learning for performance modeling of parallel applications". In: *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. (PPoPP '07). San Jose, California, USA: ACM, 2007, pp. 249–258.

[101] S. Lee, J. S. Meredith, and J. S. Vetter. "Compass: A framework for automated performance modeling and prediction". In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015, pp. 405–414.

[102] P. Leitner, E. Wittern, J. Spillner, and W. Hummer. "A mixed-method empirical study of Function-as-a-Service software development in industrial practice". In: *Journal of Systems and Software* 149 (2019), pp. 340–359. DOI: https://doi.org/10.1016/j.jss.2018.12.013.

[103] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang, et al. "{SONIC}: Application-aware Data Passing for Chained Serverless Applications". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 285–301.

[104] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni. "FaaSdom: a benchmark suite for serverless computing". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*. DEBS '20. Montreal, Quebec, Canada: Association for Computing Machinery, 2020, pp. 73–84. DOI: 10.1145/3401025.3401738.

[105] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela. "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions". In: *Future Generation Computer Systems* 110 (2020), pp. 502–514. DOI: https://doi.org/10.1016/j.future.2017.10.029.

[106] *A MapReduce Overview*. https://towardsdatascience.com/a-mapreduce-overview-6f2d64d8d0e6. Accessed 25-01-2024.

[107] C. McCaffrey. *Applying the Saga Pattern*. https://www.youtube.com/watch?v=xDuwrtwYHu8. Accessed: 2024-08-02.

[108]   S. Memeti, S. Pllana, A. Binotto, J. Kołodziej, and I. Brandic. "Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review". In: *Computing* 101.8 (2019), pp. 893–936.

[109]   Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1.* Nov. 2023.

[110]   A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. "Agile Cold Starts for Scalable Serverless". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19).* Renton, WA: USENIX Association, July 2019.

[111]   S. Mühlbauer, F. Sattler, C. Kaltenecker, J. Dorn, S. Apel, and N. Siegmund. "Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* 2023, pp. 2085–2097. DOI: 10.1109/ICSE48619.2023.00176.

[112]   V. Nair, T. Menzies, N. Siegmund, and S. Apel. "Faster discovery of faster system configurations with spectral learning". In: *Automated Software Engg.* 25.2 (June 2018), pp. 247–277. DOI: 10.1007/s10515-017-0225-2.

[113]   A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* 47.8 (2021), pp. 1528–1543. DOI: 10.1109/TSE.2019.2927908.

[114]   J. A. Pereira, M. Acher, H. Martin, J.-M. Jézéquel, G. Botterweck, and A. Ventresque. "Learning software configuration spaces: A systematic literature review". In: *Journal of Systems and Software* 182 (2021), p. 111044. DOI: https://doi.org/10.1016/j.jss.2021.111044.

[115]   J. L. Peterson. "Petri Nets". In: *ACM Comput. Surv.* 9.3 (Sept. 1977), pp. 223–252. DOI: 10.1145/356698.356702.

[116]   P. Pfaffe, M. Tillmann, S. Walter, and W. F. Tichy. "Online-Autotuning in the Presence of Algorithmic Choice". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2017, pp. 1379–1388. DOI: 10.1109/IPDPSW.2017.28.

[117]   J. C. Platt. "Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods". In: *ADVANCES IN LARGE MARGIN CLASSIFIERS*. MIT Press, 1999, pp. 61–74.

[118]   *Redis*. https://redis.io/. Accessed 2024-10-06.

[119]   E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. "Using symbolic evaluation to understand behavior in configurable software systems". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. the 32nd ACM/IEEE International Conference. Vol. 1. Cape Town, South Africa: ACM Press, 2010, p. 445. DOI: 10.1145/1806799.1806864.

[120]   S. Ren, K. He, R. Girshick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 91–99.

[121]   S. Ristov, P. Gritsch, D. Meyer, and M. Felderer. "GoSpeechLess: Interoperable Serverless ML-based Cloud Services". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ICSE-Companion '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 394–395. DOI: 10.1145/3639478.3643123.

[122]   S. Ristov, S. Pedratscher, and T. Fahringer. "AFCL: An Abstract Function Choreography Language for serverless workflow specification". In: *Future Generation Computer Systems* 114 (2021), pp. 368–382. DOI: https://doi.org/10.1016/j.future.2020.08.012.

[123]   M. Ritter, A. Calotoiu, S. Rinke, T. Reimann, T. Hoefler, and F. Wolf. "Learning Cost-Effective Sampling Strategies for Empirical Performance Modeling". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). New Orleans, LA, USA: IEEE, May 2020, pp. 884–895. DOI: 10.1109/IPDPS47924.2020.00095.

[124]   M. Ritter, A. Geiß, J. Wehrstein, A. Calotoiu, T. Reimann, T. Hoefler, and F. Wolf. "Noise-Resilient Empirical Performance Modeling with Deep Neural Networks". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 23–34. DOI: 10.1109/IPDPS49936.2021.00012.

[125] M. Rosendahl. "Automatic complexity analysis". In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, pp. 144–156.

[126] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari. "Mashup: Making Serverless Computing Useful for HPC Workflows via Hybrid Execution". In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 46–60. DOI: 10.1145/3503221.3508407.

[127] R. B. Roy, T. Patel, and D. Tiwari. "DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts". In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2022, pp. 1–18. DOI: 10.1109/SC41404.2022.00027.

[128] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, and A. Arjona. "Toward Multicloud Access Transparency in Serverless Computing". In: *IEEE Software* 38.1 (2021), pp. 68–74. DOI: 10.1109/MS.2020.3029994.

[129] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. "Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)". In: *ASE'15*. Ed. by M. B. Cohen, L. Grunske, and M. Whalen. IEEE, Nov. 2015, pp. 342–352. DOI: 10.1109/ase.2015.45.

[130] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup. *Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications*. 2022. arXiv: 2205.07696 [cs.DC].

[131] J. Scheuner and P. Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". In: *Journal of Systems and Software* 170 (2020), p. 110708. DOI: https://doi.org/10.1016/j.jss.2020.110708.

[132] T. Schirmer, N. Japke, S. Greten, T. Pfandzelter, and D. Bermbach. "The Night Shift: Understanding Performance Variability of Cloud Serverless Platforms". In: *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*. SESAME '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 27–33. DOI: 10.1145/3592533.3592808.

[133]   L. Schmid, M. Copik, A. Calotoiu, L. Brandner, A. Koziolek, and T. Hoefler. "SeBS-Flow: Benchmarking Serverless Cloud Function Workflows". In: *Proceedings of the Twentieth European Conference on Computer Systems*. EuroSys '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 902–920. DOI: `10.1145/3689031.3717465`.

[134]   L. Schmid, M. Copik, A. Calotoiu, D. Werle, A. Reiter, M. Selzer, A. Koziolek, and T. Hoefler. "Performance-Detective: Automatic Deduction of Cheap and Accurate Performance Models". In: *Proceedings of the 36th ACM International Conference on Supercomputing*. ICS '22. Virtual Event: Association for Computing Machinery, 2022. DOI: `10.1145/3524059.3532391`.

[135]   L. Schmid, T. Sağlam, M. Selzer, and A. Koziolek. "Cost-Efficient Construction of Performance Models". In: 4th Workshop on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy (PER-MAVOST '24). Pisa, Italy: Association for Computing Machinery, 2024, p. 1. DOI: `10.1145/3660317.3660322`.

[136]   E. Schoof, T. Mittnacht, M. Seiz, P. Hoffrogge, H. Hierl, and B. Nestler. "High-performance multiphase-field simulations of solid-state phase transformations using Pace3D". In: *High Performance Computing in Science and Engineering '21*. Ed. by W. E. Nagel, D. H. Kröner, and M. M. Resch. Cham: Springer International Publishing, 2023, pp. 167–184.

[137]   N. Shahidi, J. R. Gunasekaran, and M. T. Kandemir. "Cross-Platform Performance Evaluation of Stateful Serverless Workflows". In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2021, pp. 63–73. DOI: `10.1109/IISWC53511.2021.00017`.

[138]   M. Shahrad, J. Balkind, and D. Wentzlaff. "Architectural Implications of Function-as-a-Service Computing". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075. DOI: `10.1145/3352460.3358296`.

[139]   Y. Shu, Y. Sui, H. Zhang, and G. Xu. "Perf-AL: Performance Prediction for Configurable Software through Adversarial Learning". In: *ESEM*. Ed. by M. T. Baldassarre, F. Lanubile, M. Kalinowski, and F. Sarro. Bari, Italy: ACM, Oct. 2020. DOI: `10.1145/3382494.3410677`.

[140] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf. "Exascaling Your Library: Will Your Implementation Meet Your Expectations?" In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. ICS '15. Newport Beach, California, USA: Association for Computing Machinery, 2015, pp. 165–175. DOI: 10.1145/2751205.2751216.

[141] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. "Performance-influence models for highly configurable systems". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. the 2015 10th Joint Meeting. Bergamo, Italy: ACM Press, 2015, pp. 284–294. DOI: 10.1145/2786805.2786845.

[142] C. U. Smith. "Software performance engineering". In: *Performance Evaluation of Computer and Communication Systems*. Ed. by L. Donatiello and R. Nelson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–536.

[143] N. Somu, N. Daw, U. Bellur, and P. Kulkarni. "PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications". In: *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. 2020, pp. 144–151.

[144] K. L. Spafford and J. S. Vetter. "Aspen: A Domain Specific Language for Performance Modeling". In: *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 84:1–84:11.

[145] *Standard Performance Evaluation Corporation (SPEC) Benchmarks*. https://www.spec.org/benchmarks.html. Accessed: 2020-08-01.

[146] *Step Functions Workflow Collection: Saga Pattern*. https://github.com/aws-samples/step-functions-workflows-collection/tree/main/saga-pattern-tf. Accessed: 2024-08-02.

[147] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen. "Automated Performance Modeling of HPC Applications Using Machine Learning". In: *IEEE Transactions on Computers* 69.5 (2020), pp. 749–763. DOI: 10.1109/TC.2020.2964767.

[148] N. R. Tallent and A. Hoisie. "Palm: Easing the Burden of Analytical Performance Modeling". In: *Proc. of the 28th ACM International Conference on Supercomputing*. ICS '14. Munich, Germany: ACM, 2014, pp. 221–230. DOI: 10.1145/2597652.2597683.

[149]   *Transaction Processing Performance Council*. https://www.tpc.org. Accessed: 2024-01-27.

[150]   N. Trcka, W. Aalst, van der, and N. Sidorova. *Analyzing control-flow and data-flow in workflow processes in a unified way*. English. Computer science reports. Technische Universiteit Eindhoven, 2008.

[151]   D. Ustiugov, T. Amariucai, and B. Grot. "Analyzing Tail Latency in Serverless Clouds with STeLLAR". In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2021, pp. 51–62. DOI: 10.1109/IISWC53511.2021.00016.

[152]   S. Varrette, F. Pinel, E. Kieffer, G. Danoy, and P. Bouvry. "Automatic Software Tuning of Parallel Programs for Energy-Aware Executions". In: *Parallel Processing and Applied Mathematics*. Ed. by R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski. Cham: Springer International Publishing, 2020, pp. 144–155.

[153]   M. Velez, P. Jamshidi, F. Sattler, N. Siegmund, S. Apel, and C. Kästner. "ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems". In: *Automated Software Engineering* 27.3 (2020), pp. 265–300. DOI: 10.1007/s10515-020-00273-8.

[154]   M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems". In: *ICSE '21*. Madrid, Spain: IEEE, May 2021, pp. 1072–1084. DOI: 10.1109/icse43902.2021.00100.

[155]   L. Versluis, E. van Eyk, and A. Iosup. "An Analysis of Workflow Formalisms for Workflows with Complex Non-Functional Requirements". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 107–112. DOI: 10.1145/3185768.3186297.

[156]   *vSwarm - Serverless Benchmarking Suite*. https://github.com/ease-lab/vSwarm. Accessed 25-01-2024.

[157]   J. Wang and C. J. Wu. "A hidden projection property of Plackett-Burman and related designs". In: *Statistica Sinica* (1995), pp. 235–250.

[158]   L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. "Peeking behind the Curtains of Serverless Platforms". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 133–145.

[159]   M. Weber, S. Apel, and N. Siegmund. "White-Box Performance-Influence Models: A Profiling and Learning Approach". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). Madrid, Spain: IEEE, May 2021, pp. 1059–1071. DOI: `10.1109/ICSE43902.2021.00099`.

[160]   J. Wen, Z. Chen, and X. Liu. "Software engineering for serverless computing". In: *arXiv preprint arXiv:2207.13263* (2022).

[161]   J. Wen, Z. Chen, Y. Liu, Y. Lou, Y. Ma, G. Huang, X. Jin, and X. Liu. "An empirical study on challenges of application development in serverless computing". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 416–428. DOI: `10.1145/3468264.3468558`.

[162]   J. Wen and Y. Liu. "An Empirical Study on Serverless Workflow Service". In: *CoRR* abs/2101.03513 (2021). arXiv: `2101.03513`.

[163]   N. Wicki. "Control Flow Taint Analysis for Performance Modeling in LLVM". Bachelor's Thesis. Bachelor's Thesis. ETH Zurich, 2020.

[164]   M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific data* 3.1 (2016), pp. 1–9.

[165]   M. Woodside, G. Franks, and D. C. Petriu. "The Future of Software Performance Engineering". In: *Future of Software Engineering (FOSE '07)*. Future of Software Engineering. Minneapolis, MN, USA: IEEE, May 2007, pp. 171–187. DOI: `10.1109/FOSE.2007.32`.

[166]   Y. Xia, Z. Ding, and W. Shang. "CoMSA: A Modeling-Driven Sampling Approach for Configuration Performance Testing". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). Luxembourg, Luxembourg: IEEE, Sept. 11, 2023, pp. 1352–1363. DOI: `10.1109/ASE56229.2023.00091`.

[167]   T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen. "Characterizing Serverless Platforms with Serverlessbench". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 30–44. DOI: 10.1145/3419111.3421280.

[168]   Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. "Performance Prediction of Configurable Software Systems by Fourier Learning (T)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). Nov. 2015, pp. 365–373. DOI: 10.1109/ASE.2015.15.

The Karlsruhe Series on
Software Design and Quality

Band 40   **Sandro Giovanni Koch**
           A Reference Structure for Modular Model-based Analyses.
           ISBN 978-3-7315-1341-4

Band 41   **Maximilian Walter**
           Context-based Access Control and Attack Modelling
           and Analysis.
           ISBN 978-3-7315-1362-9

Band 42   **Larissa Schmid**
           Automatic Performance Modeling and Analysis of
           Configurable Scientific Software and Workflows.
           ISBN 978-3-7315-1444-2
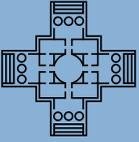
The Karlsruhe Series on Software Design and Quality

Edited by Prof. Dr. Ralf Reussner

This work reduces the cost of automatic performance modeling and enables comparative analysis of workflow orchestration performance. We use parametric profiles to cut measurement costs, a method to identify performance-irrelevant options, and a serverless workflow benchmarking suite based on a Petri Net model. These contributions help developers and scientists better understand, model, and optimize application performance across diverse platforms and configurations.