UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JHONNY MARCOS ACORDI MERTZ

Adaptive Filtering and Sampling in Runtime Software Monitoring

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Ingrid Oliveira de Nunes

Porto Alegre
December 2021

CIP — CATALOGING-IN-PUBLICATION

Mertz, Jhonny Marcos Acordi

Adaptive Filtering and Sampling in Runtime Software Monitoring / Jhonny Marcos Acordi Mertz. – Porto Alegre: PPGC da UFRGS, 2021.

122 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Ingrid Oliveira de Nunes.

1. Execution traces. 2. Monitoring. 3. Sampling. 4. Performance. 5. Logging. 6. Adaptation. 7. Self-adaptation. I. Nunes, Ingrid Oliveira de. II. Título.

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGEMENTS

Countless people supported my effort on this thesis. I would like to express my deepest gratitude to my advisor, Ingrid Nunes. She was a mentor in every sense of the word: supporting, challenging, demanding and encouraging me throughout the entire thesis process. I'm proud of, and grateful for, my time working with Ingrid.

My family deserves endless gratitude. My parents for their guidance, encouragement, and motivation. The sacrifices they've made for me are beyond any description. I hope I made them proud. Above all, I would like to thank my wife, Simone Romero, who has stood by me through all my travails and absences. Thank you for your unconditional love, comprehension and constant support, for all the late nights, holidays and long weekends, and for keeping me sane over the past few years. But most of all, thank you for being my best friend. I cannot imagine my life without you.

I want to thank my friends and colleagues from the Federal University of Rio Grande do Sul (UFRGS) and HP Inc. for their help, support and friendship. A special thank you to my colleagues Daniele, Fernando, Frederico, João, Rômulo, Maurício, Nicole and Vânius, for their suggestions, ideas and our endless conversations.

To those whose names I did not mention, but helped me develop this work in any way, thank you all. Finally, I must acknowledge that this research would not have been possible without the partial financial assistance of CNPq and the infrastructure provided by the Federal University of Rio Grande do Sul.

ABSTRACT

Understanding behavioral aspects of a software system is an essential enabler for many software engineering activities such as monitoring choke-points, debugging, and self-adaptation. Despite the usefulness of collecting system data, it may significantly impact the system execution by delaying response times and competing with system resources. Thus, runtime monitoring has limited practical use to online analysis or support real-time changes and adaptations on the program behavior. The typical approach to cope with this is to filter portions of the system to be monitored and to sample data. However, the majority of the existing solutions for filtering and sampling are limited to recording high-level events or based on predefined configurations, which unnecessarily limits the information available for analysis (i.e. relevance and representativeness of the collected set of traces). As systems often have varying workloads, some approaches dynamically change filtering and sampling configurations at runtime. Although these approaches are a step towards achieving a desired trade-off between the amount of collected information and the impact on the system performance, they focus on collecting data for a particular purpose or may capture a sample that may not correspond to the actual system behavior. In this thesis, we increase the practical feasibility of software runtime monitoring by reducing the monitoring overhead and pursuing the relevance and representativeness of the collected traces. Therefore, we propose a solution to address the challenges of filtering and sampling of execution traces. In order to filter relevant execution traces, we propose a domain-independent and low-impact framework, called Tigris, which abstracts the reasoning related to monitoring from the particularities of each problem addressed by filtering relevant execution traces according to the goal of monitoring. To tackle the challenges of collecting a representative sample, we propose an adaptive runtime monitoring process to dynamically adapt the sampling rate while monitoring software systems. It includes algorithms with statistical foundations to improve the representativeness of collected samples without compromising the system performance. We evaluated both approaches with empirical studies to assess different aspects of the proposed solutions. The results show that our techniques can reduce the overhead of monitoring by filtering and sampling traces, and pursue relevance and representativeness of collected traces.

Keywords: Execution traces. monitoring. sampling. performance. logging. adaptation. self-adaptation.

Filtragem e Amostragem Adaptativas para
Monitoração de Software em Tempo de Execução

RESUMO

Compreender o comportamento de um sistema de software é uma tarefa essencial para muitas atividades de engenharia de software, como identificação de bugs, melhorias de desempenho, depuração e auto-adaptação. Apesar da utilidade da coleta de traços de execução, isso pode afetar significativamente a execução do sistema, atrasando os tempos de resposta e competindo com os recursos. Assim, o monitoramento em tempo de execução tem uso prático limitado para análise ou suporte a mudanças em tempo real. A abordagem típica para lidar com tal problema é filtrar partes do sistema a serem monitoradas e amostrar traços. No entanto, a maioria das soluções existentes para filtragem e amostragem são baseadas em configurações predefinidas, o que limita as informações disponíveis para análise. Como sistemas costumam ter cargas de trabalho variáveis, algumas abordagens mudam dinamicamente as configurações de filtragem e amostragem em tempo de execução. Embora essas abordagens podem atingir um melhor balanço entre a quantidade de informações coletadas e o impacto no desempenho do sistema, elas se concentram na coleta de traços para uma finalidade específica ou podem capturar uma amostra não correspondente ao comportamento atual do sistema. Nesta tese, aumentamos a viabilidade prática de monitoramento de tempo de execução de software, reduzindo a sobrecarga de monitoramento e aumentando a relevância e representatividade dos traços coletados. Assim, propomos uma solução para enfrentar os desafios de filtragem e amostragem de traços de execução. Para filtrar os traços de execução relevantes, propomos um framework independente de domínio e de baixo impacto, denominado Tigris, que abstrai o raciocínio relacionado ao monitoramento das particularidades de cada problema abordado, filtrando os traços de execução relevantes de acordo com o objetivo do monitoramento. Em relação aos desafios de coletar uma amostra representativa, propomos um processo de monitoramento adaptativo, que altera dinamicamente a taxa de amostragem de acordo com a carga de trabalho do sistema. Nós avaliamos ambas as abordagens com estudos empíricos, e os resultados mostram que as técnicas propostas podem reduzir a sobrecarga de monitoramento por meio da filtragem e amostragem de traços, e aumentar a relevância e representatividade dos traços coletados.

Palavras-chave: traços de execução, monitoração, adaptação, amostragem, desempenho.

LIST OF ABBREVIATIONS AND ACRONYMS

ADP	Adaptive Monitoring Process Sampling
AOP	Aspect-oriented Programming
BNF	Backus–Naur Form
DSL	Domain-specific Language
F1	F-Measure
FUM	Full Monitoring
HR	Hit Ratio
INV	Inversely Proportional Sampling
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
LOC	Lines of Code
PADLA	Phase-Aware Dynamic Log Level Adapter
RMSE	Root-mean Square Error
RpS	requests per second
RV	Runtime Verification
SAS	Self-adaptive Systems
SR	Sampling Rate
TR	Throughput
TTL	Time-to-live
UNI	Uniform Sampling

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

As modern software systems become increasingly large and complex, effective analysis methods to understand the dynamic behavior of a software system are becoming essential to ensure software quality by supporting fundamental software engineering tasks such as runtime verification, debugging, program comprehension, and self-adaptation (KANG, 2018; FENG et al., 2018). The understanding of the system behavior can be achieved by monitoring runtime information (GAO et al., 2017) (i.e. dynamic system's executions), which are typically represented in the form of execution traces (PIRZADEH et al., 2011; YUAN; ESFAHANI; MALEK, 2014; REGER; HAVELUND, 2016). *Execution traces* are log events from rules, components, functions, or tasks executed during a program run. An execution trace can be composed of basic information about the execution like event name, program location, and a timestamp. In addition to basic information, more complex and detailed data about the event can also be recorded. Inputs, outputs, the complete call trace, software execution states, message communication, and resource usage are examples of additional data that can be recorded to represent an execution trace. Therefore, analyzing and comparing program execution traces is useful for various purposes, such as the validation of quality requirements (FINOCCHI, 2013), identifying security vulnerabilities (YUAN; ESFAHANI; MALEK, 2014) or model inconsistencies (BARTOCCI et al., 2018), performance engineering (DELLA TOFFOLA; PRADEL; GROSS, 2015; MERTZ; NUNES, 2018), and optimization (FENG et al., 2018). AWS X-Ray[1], for example, is a tool that uses tracing execution to provide an end-to-end view of request paths in software applications, including a map of the application's underlying components. When an exception occurs while the application is serving an instrumented request, AWS X-Ray records details about the exception, including the stack trace. This helps, e.g., identify and troubleshoot the root cause of performance issues and errors.

A widely used form of collecting execution traces is intercepting particular points in the execution to record information, e.g. intercepting methods to record their input and output data. The act of monitoring a system to collect trace information is called *instrumentation* and it works by hooking additional code instructions into the different layers and locations of the software system. These code instructions are responsible for collecting and storing runtime information from specific components of a system or its

[1] <https://aws.amazon.com/xray/>

execution environment while the software is running, depending on the monitoring goal. Despite the usefulness of execution traces, collecting them at runtime consumes resources and may cause performance decays (BARTOCCI et al., 2018), mainly when they include detailed information, such as method parameters.

To address this, execution traces can be filtered or sampled. *Filtering* techniques usually consist of focusing the monitoring on *relevant* executions (KOUAME; EZZATI-JIVAN; DAGENAIS, 2015; EICHELBERGER; SCHMID, 2014), i.e. the collected subset of traces is focused on target software locations or execution patterns that are more important according to a specific set of criteria to achieve a particular goal with monitoring. In trace *sampling*, the aim is to collect a representative subset of traces based on a sampling rate or strategy (HAMOU-LHADJ; LETHBRIDGE, 2004; PIRZADEH et al., 2011; PIRZADEH et al., 2013; LAS-CASAS et al., 2018), where *representativeness* means that the characteristics of the collected subset of traces should closely match the complete set of target execution traces, by following the same distribution. The key advantage of filtering and sampling techniques is the bounded overhead, which linearly decreases with the filtering configuration, sampling period or size.

Filtering and sampling execution traces have been commonly adopted with predefined and fixed configurations, which specify certain software locations to be monitored and/or a sampling rate. These configurations may be unsuitable to cope with software usage peaks and unable to handle unforeseen scenarios. AWS X-Ray, for instance, applies a conservative sampling strategy and records only the first request of each second and five percent of any additional requests. Any different strategy must be manually managed, considering the performance impact it may cause to the application. These limitations are addressed by *adaptive* approaches (ZAVALA; FRANCH; MARCO, 2019). However, existing work either focuses only on collecting traces for a particular purpose (FEI; MIDKIFF, 2006; LAS-CASAS et al., 2018) or uses a strategy that cannot guarantee that the collected traces are a representative sample of the population (HAUSWIRTH; CHILIMBI, 2004; BRÖNINK; ROSENBLUM, 2016). This can potentially cause wrong decision making based on the sample or missing information.

These limitations of existing work call for reusable and configurable approaches that make software monitoring practically feasible by maintaining the overhead to acceptable levels to preserve the normal software behavior and increasing the relevance and representativeness of the monitored traces. In the next section, we detail the problem we are looking at in this thesis and the limitations of existing work. In Section 1.1, we

introduce the proposed solution and give an overview of the contributions of this work. Finally, Section 1.3 presents the structure of the remainder of the thesis.

1.1 Problem Statement and Limitations of Existing Work

As previously introduced, there are two main strategies, namely filtering and sampling, that may help address the monitoring overhead imposed by the runtime collection of execution traces. Both strategies have inherent limitations. Filtering lacks reuse across different systems and application domains, being customized to particular cases, e.g., by limiting the tracing only to high-level events or a pre-defined set of executions. Sampling usually focuses on collecting traces for a particular purpose or relies on a technique that cannot guarantee that the collected traces are representative of the population. Based on these problems, the research question considered in this thesis is stated as follows.

? Research Question

How to monitor software systems to collect execution traces at runtime that are relevant and representative with an acceptable performance impact?

The limitations of existing work related to this research question are presented as follows.

Limited scalability. Runtime monitoring often leads to an overhead in the observed program, being mostly used within offline analysis tools (post-mortem manner) and having a limited practical use to change or adapt program behavior at runtime (KANG, 2018). Thus, the monitoring task in such approaches is often limited to high-level events or pre-defined executions because collecting detailed measurements (e.g. method-level tracing) and arbitrary executions may disrupt execution. This may reduce the accuracy of the analysis when finding and solving issues at the code level (HORKÝ et al., 2016). Although some techniques such as sampling and filtering have been demonstrated as a practical solution to reduce the trace size and enable faster trace analysis, it is necessary to pursue a representative and relevant set of collected traces, in order to generalize the results of the analysis.

Limited adaptability. The majority of the approaches require manual tuning of several parameters, such as the definition of custom filtering behavior patterns or fixed sampling

configurations (ZAVALA; FRANCH; MARCO, 2019). Finding adequate parameters can be difficult and effective only for specific sets of traces (even within a single system). In addition, these configurations may be unsuitable to cope with software usage peaks and unable to handle unforeseen scenarios. Thus, there is a need for finding ways to adapt the monitoring to deal with the trade-off between effectiveness and overhead of monitoring.

Lack of generality. Most existing solutions that rely on monitoring are application-specific, i.e. while they carefully collect selected data at runtime for a particular purpose, they are not designed and implemented in a more general, reusable way. The lack of reusable approaches to support the development of monitoring-based approaches requires them to be implemented from scratch. In addition, low-overhead runtime monitoring strategies serve specific purposes and, consequently, are not generalizable. The development of reusable monitoring approaches that abstract and encapsulate these functionalities would reduce both the effort to develop new systems implementing these strategies and the probability of bugs in newly implemented solutions. In addition, it would promote software reuse across different goals and application domains.

1.2 Proposed Solution and Contributions Overview

Because software monitoring consists of the execution of additional instructions, it necessarily implies an overhead (BARTOCCI et al., 2018). Although it can be reduced to an acceptable level by leveraging filtering and sampling techniques, existing work either focuses only on collecting traces for a particular purpose (FEI; MIDKIFF, 2006; LAS-CASAS et al., 2018), are adopted with pre-defined and fixed configurations, or uses a strategy that cannot guarantee that the collected traces are relevant and representative of the population (HAUSWIRTH; CHILIMBI, 2004; BRÖNINK; ROSENBLUM, 2016). By addressing these issues, it is possible to provide a solution able to comprise an adequate trade-off between monitoring overhead and relevance or representativeness of the collected information.

As filtering and sampling are essentially different problems in which solutions can be combined to address the existing monitoring issues and compose an effective monitoring solution, we individually tackle them. To address the issues around filtering and provide a solution that can be reused across different systems and application domains, we propose a two-phase monitoring approach for filtering execution traces at runtime. In its

first phase, lightweight and coarse-grained monitoring are performed to identify relevant parts of the software execution. As relevance is essentially a domain-specific concept, this process is informed by the *goal of monitoring* in the form of high-level *relevance criteria* expressed in a proposed domain-specific language, called *TigrisDSL*. In practice, the criteria are translated into software metrics, which are collected and analyzed at runtime to guide an in-depth and fine-grained monitoring in the second phase of the approach. Both relevance criteria and software metrics are derived from a systematic literature review. Our approach is implemented as a framework, named Tigris, which seamlessly integrates the proposed solution to existing software systems to support monitoring-based activities. Consequently, our approach and framework can be used as a monitoring component to effectively monitor a software system and provide information for different purposes, e.g. to identify security vulnerabilities (YUAN; ESFAHANI; MALEK, 2014), model inconsistencies (BARTOCCI et al., 2018) or performance bugs (DELLA TOFFOLA; PRADEL; GROSS, 2015; MERTZ; NUNES, 2018). To validate our proposal, we instantiate Tigris as the monitoring component of an approach that improves the performance of applications using caching. Our evaluation shows that our proposal can maintain the effectiveness of the caching approach while reducing the monitoring overhead.

To address the limitations associated with sampling, we propose an adaptive sampling process to collect execution traces with detailed information in environments where the performance impact is critical, such as production environments. Our goal is to guarantee the representativeness of the samples of execution traces while adjusting the sampling rate used to monitor a software system to cope with increases in its workload. The process decides whether the operations executed to respond to each incoming request should be recorded as execution traces. Our process is composed of three activities: (1) *sampling decision*, which decides whether a request (with associated execution traces) should be recorded and included in a sample; (2) *sampling rate adaptation*, which adjusts the sampling rate at runtime; and (3) *sample evaluation*, which assesses the representativeness of the sample to identify the end of a monitoring cycle. These activities include algorithms with statistical foundations to pursue that, by the end of each monitoring cycle, the collected sample is representative of the population. Our evaluation targets five applications of the well-known DaCapo benchmark and the results show that it can achieve higher representativeness and lower performance impact than existing solutions.

Therefore, both proposed solutions increase the practical feasibility of software monitoring by reducing the monitoring overhead and pursuing the relevance and repre-

sentativeness of collected traces. The proposed solutions can be used individually or combined to effectively monitor software applications to provide information for approaches with different purposes, such as identifying security vulnerabilities (YUAN; ESFAHANI; MALEK, 2014), model inconsistencies (BARTOCCI et al., 2018) or performance bugs (DELLA TOFFOLA; PRADEL; GROSS, 2015; MERTZ; NUNES, 2018) and, eventually, adapt the system.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents a theoretical background that serves as the foundations needed for this thesis. It also discusses related work that addresses challenges on filtering and sampling as a way to reduce the monitoring overhead. In Chapter 3, we present a foundational study to identify the relevance criteria associated with each monitoring goal and its results, which includes the derivation of a domain-specific language, named TigrisDSL. Chapter 4 presents the two-phase monitoring approach for filtering execution traces at runtime, as well as its implementation as a framework, called Tigris. The adaptive sampling process is presented in Chapter 5. Finally, Chapter 6 concludes this thesis and outlines directions for future work.

2 BACKGROUND AND RELATED WORK

Software runtime monitoring is the ability to record log events with metadata about executions of a program run. This ability is usually mentioned as a required feature to support several software engineering tasks focused on the analysis of the system behavior or automation of a software system (KANG, 2018; FENG et al., 2018). In Section 2.1, we explore applications of monitoring in a broader sense and highlight the main challenges faced by approaches that rely on monitoring. Then, in Sections 2.2 and 2.3, we present the limitations of the two main existing techniques, respectively, to minimize the monitoring impact. By monitoring, we refer solely to the activity of collecting data at application runtime, rather than this activity combined with data analysis, which together are sometimes considered a monitoring system (ZAVALA; FRANCH; MARCO, 2019).

2.1 Introduction to Runtime Software Monitoring

Runtime monitoring has been applied to different purposes to systematically collect, analyze, and use information to gain insights or solve problems in software engineering tasks. Typical applications of runtime monitoring are the validation of quality requirements with runtime verification, analysis and improvement of non-functional requirements such as performance and security engineering or adaptive software systems. Examples of these applications of monitoring and its research areas are discussed as follows.

Runtime Verification (RV) is a popular field that is entirely based on runtime monitoring. It is a lightweight, yet rigorous, formal method that complements classical exhaustive verification techniques (such as model checking and theorem proving) with a more practical approach towards ensuring correctness (BARTOCCI et al., 2018). RV thus employs techniques for observing the internal operations of a software system and its interactions with other external entities to determine whether the system satisfies or violates a correctness specification (CASSAR et al., 2017). RV is widely employed in academia and industry before system deployment, for testing, verification, and debugging purposes, and after deployment to ensure reliability, safety, robustness, and security (BARTOCCI et al., 2018). Runtime monitoring has also been used to acquire information that guides decisions on *non-functional requirements*, such as security (e.g. detection of anomalous or malicious behavior (TUN et al., 2018a; BAILEY et al., 2014)), and performance (e.g.

finding performance bottlenecks and optimization opportunities (SHEN et al., 2015; TOFFOLA; PRADEL; GROSS, 2018; MERTZ; NUNES, 2018)). For example, based on monitoring small code executions such as method calls or functions, some approaches extract metrics that can be used to identify caching opportunities and manage caching systems (HOLMQVIST; NILSFORS; LEITNER, 2019; MERTZ; NUNES, 2017b).

When used to support *Self-adaptive Systems* (SAS), monitoring and tracing have been used as a source of information to find adaptation opportunities. The idea is that the system can collect information about itself and act proactively to configure, heal, optimize, and protect itself without human intervention (YUAN; ESFAHANI; MALEK, 2014; HUEBSCHER; MCCANN, 2008; LALANDA; MCCANN; DIACONESCU, 2013).

Runtime monitoring has been performed based on different techniques, but function call interception (or instrumentation) has traditionally been a major approach used to monitor system behaviors, because a function is considered the most basic element comprising a program module (KANG, 2018).

Instrumentation can be applied at the program source or its binary level, but it is usually performed through dynamic techniques by binary instrumentation at load time or at runtime to a program loaded in memory (KANG, 2018). One of the popular techniques for runtime monitoring is aspect-oriented programming (AOP) (KICZALES et al., 1997; KANG, 2018), which was initially conceived to address cross-cutting concerns that cannot be easily moduralized in an application, such as logging, security, and caching. By using AOP, these cross-cutting concerns can be centralized in one location, thus encouraging modularity and maintainability in the software design process. Rabiser et al. (2017) provided a comparison framework for runtime monitoring approaches, which is based on an analysis of the literature and existing taxonomies for monitoring languages and patterns. They evaluated 32 existing monitoring approaches, concluding that the most popular and suitable to different scenarios are those based on AOP. Listing 2.1 gives an example of how AOP is adopted in practice. In the example, business logic is being intercepted for logging purposes.

Regardless of the goal of monitoring or the technique being used, two major problems must be addressed, as presented in Figure 2.1: (1) the overhead caused by instrumenting an execution of the running systems; and (2) analyzing a large amount of execution traces that can be generated.

The overhead becomes a significant issue mainly in real-time applications, which are time-sensitive and must often meet deadlines in resource-constrained environments (MI-

Listing 2.1 – A code snippet of the business logic from a system that is intercepted via AOP with an AspectJ annotation. The `logBefore()` method is executed before the execution of the `doSomething()` method of the `BusinessLogic` class.

```
1  public class BusinessLogic {
2    public void doSomething() { ... }
3  }
4
5  @Aspect
6  public class LoggingAspect {
7     // pointcut at the beginning of the method execution
8    @Before("execution(* BusinessLogic.doSomething(..))")
9    public void logBefore(JoinPoint joinPoint) {
10     System.out.println(joinPoint.getSignature().getName() + " is running!");
11   }
12 }
```

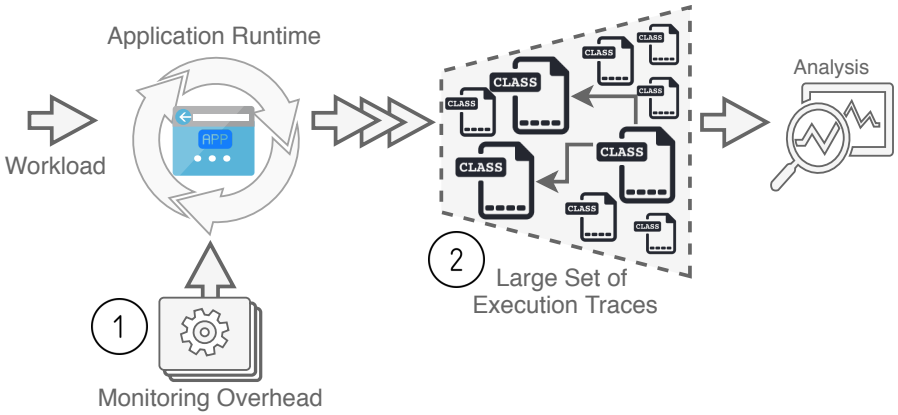


Figure 2.1 – Monitoring a Running Application and its Challenges.

RANSKYY et al., 2016). The runtime overhead is most critical because this is what the users of the system will instantly experience. The large sets of collected traces, in turn, frequently imply high storage space and analysis time demands, resulting in a limited analysis of a subset of traces (PIRZADEH et al., 2011). The main issue in this context is how to adequately monitor and acquire information from the application and its environment without compromising its performance. Such monitoring should be as non-intrusive as possible to avoid significantly affecting the performance of the application, which is not always possible (KANG, 2018; RABISER et al., 2017).

Filtering and sampling have been demonstrated as a practical solution to reduce the monitoring overhead and enable faster trace analysis (BARTOCCI et al., 2018). While *filtering* focuses the monitoring only on particular executions or software locations that are supposed to be relevant for the goal of monitoring, *sampling* involves establishing a sampling rate to monitor a subset of execution traces, assuming that it is a representative sample of the population of traces. The key idea behind these solutions is that only a small subset of traces is typically significant and enough to the analysis (HAMOU-LHADJ; LETHBRIDGE, 2004; PIRZADEH et al., 2013; MIRANSKYY et al., 2016).

In addition, these two approaches can be combined to keep the monitoring overhead at acceptable levels. There is existing work to support the specification of filtering and sampling configurations, which give the scope of monitoring (i.e. locations to monitor) and the sampling rate, respectively. In the following sections, we analyze solutions that support the development of monitoring-based approaches based on filtering and sampling.

2.2 Filtering

Filtering approaches have been proposed to cope with the monitoring impact by selecting a subset of traces based on event type, time, description, spatial location, or even the priority or importance (PIRZADEH et al., 2013; KOUAME; EZZATI-JIVAN; DAGENAIS, 2015). Consequently, only traces that match a particular pattern are selected (ZAIDMAN; DEMEYER, 2004; REISS, 2005; HAMOU-LHADJ; LETHBRIDGE, 2006; CORNELISSEN et al., 2008; MIRANSKYY et al., 2008; PIRZADEH et al., 2013), such as being of a certain program region (FEI; MIDKIFF, 2006; NARAYANAPPA; BANSAL; RAJAN, 2010). They can be classified into two groups: (i) domain-specific solutions that address the filtering challenges on particular applications or goals of monitoring (Section 2.2.1); and (ii) approaches that provide generic filtering capabilities, designed to be applied to different domains and purposes (Section 2.2.2). Approaches in these groups are discussed as follows.

2.2.1 Domain-specific Runtime Monitoring

Domain-specific runtime monitoring approaches have been proposed to address filtering on individual systems, particular architectural styles, or technologies (VIERHAUSER; RABISER; GRÜNBACHER, 2016). Although these approaches address *monitoring scalability* challenges in a domain-specific way, it is not always trivial to specify monitoring patterns and triggers to obtain traces of interest (RABISER et al., 2017). The selection of monitoring targets may be guided by a prior and possibly static analysis of the code (CHEN et al., 2014; KIM et al., 2016a). Focusing on memory, Daoud, Ezzati-Jivan and Dagenais (2017) proposed a dynamic tracing approach that monitors memory usage to decide whether to collect a trace based on pre-defined conditions and thresholds such as the time elapsed since the last trace or the amount of memory allocation calls. Further-

more, these specifications are usually fixed, which limits the flexibility and adaptability of the monitoring and couples the tracing strategy with the analysis, resulting in a possibly hidden relevant behavior that remains undetected (HORKÝ et al., 2016).

In addition, new applications of monitoring have created dependence between the goal of monitoring and the monitoring itself, by embedding the monitoring logic into the logic related to the analysis of the behavior or the action being automated (ALONSO et al., 2009). These solutions are limited in terms of reuse because they are developed in an ad-hoc manner and require re-engineering work in order to adapt applications to obtaining tracing features. To support specifying a filtering configuration, some approaches perform an automated offline analysis of the program to define relevant application regions or paths (APIWATTANAPONG; HARROLD, 2003; SANTELICES; SINHA; HARROLD, 2006; SRIDHARAN; FINK; BODIK, 2007; NARAYANAPPA; BANSAL; RAJAN, 2010). Although helpful, these solutions are not suitable when the areas of interest vary at runtime. In these cases, the approaches should re-executed to tune the monitoring configuration.

2.2.2 Domain-neutral Runtime Monitoring

There are proposals to increase the *generality* of filtering in different directions to achieve independence of the purposes for which the monitoring is performed, requiring a reduced amount of development effort. Alonso et al. (2009) proposed a monitoring framework based on Aspect-Oriented Programming (AOP) with a flexible architecture, which can collect a large set of internal and external metrics and change the level of monitoring when it is needed to obtain more fine-grained data from the system. Similarly, Eichelberger and Schmid (2014) described *SPASS-meter*, a resource monitoring approach for Java and Android Apps. It is able to monitor the resource consumption of individual semantic units of the observed program, such as components or services, based on a specification provided by the user. However, although the proposal aims to achieve generic monitoring, SPASS-meter focuses on countable resources such as CPU time, response time, memory allocation, network, and file usage. These are useful for traditional performance engineering but less applicable to other monitoring goals, such as enhanced bug finding or anomaly detection approaches, which demand details of the data being processed.

Adaptive filtering proposals can change the metrics being collected or target lo-

cations oriented by the domain-specific analysis and occurrence of an event of interest. Work in this direction has recently been investigated in a systematic mapping (ZAVALA; FRANCH; MARCO, 2019) that reveals that most of the proposed *adaptations* focus on improving the monitoring results for a *specific purpose*. Fei and Midkiff (2006), for example, observed that executing a (region of an) application with the same context tends to produce the same outcome. Therefore, repeated executions do not need to be monitored when the goal is to identify where bugs are likely to occur. Similarly, Brand and Giese (2019) proposed an approach in which runtime models that represent properties of the system that can be monitored are used to derive different monitoring configurations. Then, this information is provided, at runtime, to other mechanisms that can access the monitored data through search queries. The adaptation is put in place by constantly observing how the monitored data is queried and used by other mechanisms. Based on such observations, the approach can change the monitoring configuration to avoid unnecessary monitoring effort, i.e. monitored properties that are not being used are disabled. Therefore, the monitored data, which describes the relevant information to the specific current needs, influences the monitoring configuration.

Recently, Mizouchi et al. (2019) proposed PADLA (Phase-Aware Dynamic Log Level Adapter), which is an extension of a popular logging library that enables the dynamic change of the log level that should be captured. PADLA employs an online phase-detection that splits a program execution into phases, and based on instrumentation, it collects metadata about these phases, such as the execution time. Despite being only focused on pre-defined log instructions, PADLA allows the system to dynamically increase and decrease the log level of specific execution phases dynamically in the face of relevant events, such as performance anomalies. By doing so, it is possible to automatically record relevant events in detail while recording regular events concisely. Although there are approaches that support to achieve adaptation on monitoring, generic adaptation goals, such as increasing the relevance of the collected traces or dynamically managing the monitoring overhead, remain unaddressed and should be achieved by other means.

2.3 Sampling

Sampling-based approaches involve establishing a sampling rate to monitor a subset of execution traces, assuming that it is a representative sample of the population of traces. The key advantage is the bounded overhead, which linearly decreases with the

reduction of the sampling period and size. Proposed approaches can be classified into two groups: (i) *fixed configuration*, which keeps the same configuration throughout the software execution until it is manually updated; (ii) *adaptive configuration*, which dynamically adjusts the configuration based on constraints and lightweight monitored data. These groups are discussed as follows.

2.3.1 Fixed Configuration

A straightforward way to cope with the monitoring overhead is to use random or systematic sampling (CHAN et al., 2003; DUGERDIL, 2007; JUNG et al., 2014; ZHOU et al., 2016; SONG; LU, 2017), which is used in various commercial and open-source tools (VAN HOORN; WALLER; HASSELBRING, 2012; HORKÝ et al., 2016). However, as there are traces that are not recorded, important traces may be missed. Thus, choosing a sampling rate is a challenge (LAS-CASAS et al., 2018; MIRANSKYY et al., 2016) because of the trade-off between the representativeness of the sample and the performance overhead. A suitable solution to the monitoring *scalability* challenges in some scenarios is the use of a fixed (higher) sampling rate but targeting particular executions or regions of an application (i.e. combining filtering and sampling). Nevertheless, when the population of traces is not homogeneous, focusing on statically defined regions may lead to reduced coverage and thus an unrepresentative sample (PIRZADEH et al., 2013).

2.3.2 Adaptive Configuration

Adaptive sampling approaches change, at runtime, the sampling configuration, and even collected metrics. However, the proposed adaptations are mostly focused on improving the monitoring results for a specific purpose and limited to changing configurations based on pre-defined setups or assumptions like that by reducing the sampling rate it implies the performance overhead is also reduced (ZAVALA; FRANCH; MARCO, 2019). For example, Las-Casas et al. (2019) aim to maximize the diversity of execution traces in the sample with infrequent patterns in order to capture outliers and anomalous traces in the sample. Their proposal computes the distance among traces and pursues the diversity in the set of traces, given a fixed budget. Targeting performance, Ding et al. (2015) proposed a cost-aware logging mechanism that decides whether to keep log mes-

sages based on (1) a dynamic measurement of the performance of the code snippet that generated the log and (2) an allowed maximum volume of logs in a time interval. The goal is to keep logs related to code snippets that execute slower than in the past.

These aforementioned adaptive approaches focus on dynamic filtering and adopt a fixed sampling rate—some (FEI; MIDKIFF, 2006) allow it to be given as a parameter. Targeting the application workload, three approaches (GONG; PRADEL; SEN, 2015; HAUSWIRTH; CHILIMBI, 2004; BRÖNINK; ROSENBLUM, 2016) propose to use a sampling rate that is inversely proportional to the frequency of execution, which gives the application throughput. Although this strategy can reduce the monitoring overhead when the application is overloaded, the collected sample may not correspond to the population of execution traces. In usage peaks, the proportion of collected execution traces is smaller than in typical workloads. As result, collected samples are likely not representative considering the population.

2.4 Final Remarks

Runtime monitoring often leads to an overhead in the observed program, being mostly used within program analysis tools and having limited practical use to change or adapt program behavior at runtime (KANG, 2018). In the latter case, the monitoring is often limited to logging high-level events. Detailed measurements, e.g. method-level tracing, tend to be avoided because their overhead can disrupt execution. This limits the information available for analysis when finding and solving issues at the code level (HORKÝ et al., 2016).

Filtering and sampling have been demonstrated as practical solutions to reduce the monitoring impact. However, the existing solutions have limitations in terms of generality, adaptability, and scalability, and achieving an effective solution requires manually dealing with these limitations. Table 2.1 presents the identified related work, with the approaches that address the different challenges on monitoring and how it is done.

Thus, there is a need for monitoring techniques that (1) make an adequate trade-off between the monitoring overhead, by not compromising the system operation and still collecting relevant and representative execution traces; and (2) can be used for different purposes, such as detecting failure points or performance issues. The next chapters present alternatives to address the challenges of monitoring, reducing the limitations imposed by existing solutions.

Table 2.1 – Comparison of related work on monitoring. Empty columns mean that the approach does not offer explicit support to the characteristic.

		Monitoring Characteristic		
	Approach	**Generality**	**Adaptability**	**Scalability**
Filtering	Chen et al. (2014), Kim et al. (2016a), Apiwattanapong and Harrold (2003), Santelices, Sinha and Harrold (2006), Sridharan, Fink and Bodik (2007)	—	—	Selects a subset of executions to monitor
	Zaidman and Demeyer (2004), Daoud, Ezzati-Jivan and Dagenais (2017), Miranskyy et al. (2008), Pirzadeh et al. (2013)	—	—	Filters traces that match a particular pattern or constraint
	Eichelberger and Schmid (2014)	Filters different target locations based on custom definition	—	—
Both	Fei and Midkiff (2006), Narayanappa, Bansal and Rajan (2010)	—	—	Sampling applied to a predefined subset of executions
	Alonso et al. (2009)	Addresses different levels of granularity	—	—
	Brand and Giese (2019), Brand and Giese (2018)	Captures different properties according to a monitoring configuration	Changes results based on access history	—
	Mizouchi et al. (2019)	Extension of a popular logging library	Changes the log level dynamically in face of relevant events	—
	Approaches reported by Zavala, Franch and Marco (2019)	—	Changes the metrics collected, sampling configurations or target locations oriented by a suspected problem	—
	Ding et al. (2015)	—	—	Keeps traces according to a predefined budget
Sampling	Las-Casas et al. (2018), Las-Casas et al. (2019)	—	—	Maximizes the diversity of execution traces
	Dugerdil (2007), Chan et al. (2003), Jung et al. (2014), Zhou et al. (2016), Song and Lu (2017)	—	—	Sampling applied to all executions
	Fischmeister and Ba (2010), Thomas, Fischmeister and Kumar (2011)	—	—	Changes sampling rate based on predefined scenarios
	Gong, Pradel and Sen (2015), Hauswirth and Chilimbi (2004), and Brönink and Rosenblum (2016)	—	—	Sampling rate is inversely proportional to the frequency of execution

3 RELEVANCE CRITERIA AND TIGRISDSL

A complete set of execution traces captures the execution of every operation of a software system. However, not every trace is equally relevant given a particular goal when monitoring the system, because only a subset of the traces contains the information needed to diagnose a target system symptom. These traces may be concentrated in parts of the code with specific characteristics. For example, if the goal of the monitoring is runtime verification (HAMOU-LHADJ; LETHBRIDGE, 2004; REGER; HAVELUND, 2016), methods that handle many exceptions or include type castings may be the primary source of relevant traces, as such methods are error-prone. Alternatively, if the purpose of monitoring is performance optimization, methods that are more frequently invoked might be those to be tracked (CHEN et al., 2018). The relevance of an execution trace for analysis depends on the *purpose* of the analysis, i.e. the monitoring goal. A sample of collected execution traces is said *relevant* if the portions of the source code being monitored satisfy a set of relevance criteria. A relevance criterion is defined as follows.

Relevance Criterion. A relevance criterion is the specification of a property of system events (e.g. the execution of a method or a function) that characterizes the types of events that are more likely to be useful than others, according to a particular monitoring goal.

To identify the relevance criteria associated with each monitoring goal, we surveyed monitoring-based approaches from the literature, identifying their goals as well as adopted criteria and metrics to analyze execution traces to understand the system behavior. Next, we first describe the method used to select and analyze research work in this context and then present obtained results. Founded on them, we introduce the *TigrisDSL*, which is a domain-specific language (DSL) to specify relevance criteria.

3.1 Method

We surveyed full papers published in the Computer Science conferences presented in Table 3.1. These conferences have on their scope of interest research related to the analysis of execution traces for profiling, adaptation or code understanding. We selected the two most highly ranked conferences in software engineering (ICSE and ESEC/FSE), and five other conferences where system monitoring is a core concern due to its importance to the software adaptation area (ICAC, SASO, and SEAMS) and code analysis and un-

Table 3.1 – Conferences from where papers from the past six years (2014–2019) were obtained and analyzed.

Acronym	**Conference**
ESEC/FSE	Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
ICSE	International Conference on Software Engineering
OOPSLA	Conference on Object-Oriented Programming Systems, Languages, and Applications
ICAC	International Conference on Autonomic Computing
SASO	International Conference on Self-Adaptive and Self-Organizing Systems
SEAMS	International Symposium on Software Engineering for Adaptive and Self-Managing Systems
ICPC	International Conference on Program Comprehension

derstanding (OOPSLA and ICPC). We focused only on conferences to pursue face-paced publications considering that they have faster review and publication processes than journals. Moreover, papers published in journals are often extensions of conference papers. In addition, we also covered journal-first publications of relevant journals in the area, as these papers can be presented at selected conferences. We restricted our search to the past six years (2014–2019). To filter relevant papers, we searched the databases where the proceedings of these conferences were published using the following query string.

(*runtime* OR *dynamic*) AND (*monitor* OR *instrument* OR *record* OR *track* OR *profile*) AND (*trace* OR *execution*)

As result, we obtained 147 papers, from which 64—based on the title and abstract—fit the criterion of including a monitoring-based approach. They were analyzed according to four dimensions:

1. identification of the monitoring *goal*, the *criteria* adopted to analyze execution traces and the used *metrics*;
2. assessment of the *generality* of the monitoring approach in terms of its use for different application domains and purposes;
3. examination of *scalability* aspects, such as the overhead and size of traces; and
4. inspection of the *adaptability* of the monitoring approach, that is, the runtime changes in the monitoring strategy to address a particular issue, such as the reliability of collected traces or overhead reduction.

Table 3.2 – Analysis Approach. Questions used in the analysis of monitoring-based approaches.

#Q	Question
1	What is monitored?
2	What are the metrics used?
3	What are the criteria related to the metrics?
4	What is the goal of monitoring?
5	How are traces collected?
6	What is the granularity of the collected traces?
7	What is the amount of the collected traces?
8	How were the traces generated for evaluation?
9	Is the monitoring automatically integrated?
10	Is the monitoring adaptive? What is the goal of adaptation?
11	How is adaptation achieved? What is the adaptation trigger?
12	Is there monitoring overhead? How is the overhead dealt with?
13	Is representativeness considered when collecting traces?
14	How is representativeness ensured?
15	Is relevance considered when collecting traces?
16	How is relevance ensured?
17	How specific is the monitoring approach in terms of technology or domain/purpose?

To guide the analysis of the selected papers, we used pre-specified questions, presented in Table 3.2, which served as a checklist while analyzing the papers. The analysis was performed manually by reading the selected papers and answering the questions. Table 3.3 provides examples of such analytical processes and illustrates how we reached the results of our study[1].

The combined analysis of the four dimensions based on information gathered from the selected papers provided us with the foundation needed to elaborate a solution that can effectively support the development of monitoring-based approaches. The results are presented as follows.

3.2 Analysis and Results

3.2.1 Goals, Criteria, and Metrics

Based on the analysis of our selected papers, we identified the criteria presented in Table 3.4. This table also shows the number of papers in which each criterion appears. As

[1]The detailed analysis of each paper is available at <http://www.inf.ufrgs.br/prosoft/resources/2020/jss-effective-tracing>

Table 3.3 – Analysis Approach. Example of how information from the selected papers were collected by answering the questions specified in Table 3.2.

Sample Quote from Selected Papers	Gathered Information	#Q
(BARNA et al., 2017): The proposed monitoring module gathers CPU utilization for all **the back-end tiers of the applications (Web, Spark and Cassandra) on the container level**, CPU utilization of **the host VMs**, as well as response time and throughput from **the entire system**.	Target of monitoring	1
(DELLA TOFFOLA; PRADEL; GROSS, 2015): [...] uses state of the art CPU time profiling to identify the set of initial memoization candidates. We record, **for each executed method *m*, the time t_m spent in *m*** (including time spent in callees) and **the number c_m of calls**. Furthermore, we also measure **the total execution time** t_{prgm} of the program.	Metrics	2
(KANG et al., 2016): **Based on the profiling data, DiagDroid detects performance issues and analyzes their causes**. A report can finally be generated with an **aim to direct the debugging process**.	Goal of monitoring	4
(SU et al., 2016b): There are several factors that can contribute to the **runtime overhead of HitoshiIO** [...] The most dominant factor for execution time in our experiments was the clone identification time: application analysis was relative quick (order of seconds), and the input-output recorder **added only a roughly 10x overhead**	Scalability	12

can be seen, the *frequency* of an execution event is the most adopted criterion to classify relevant traces, followed by *maintainability*.

Although the order in this table may give evidence of the importance of the criteria, note that each criterion has a different purpose. Consequently, the number of occurrences is also associated with the most common goals of monitoring. We thus, in addition to the identification of criteria, investigated the goal of monitoring in the selected papers. The identified goals are clustered into five groups, as presented in Table 3.5. For each group, we show examples of specific goals that appeared in the papers. We also highlight the most used criteria for each goal. To assess whether execution events match the relevance criteria of a specific goal, various metrics are used to make an objective evaluation. Cells of Table 3.5 indicate the most relevant metrics associated with each pair of goal and criterion that appeared in our surveyed literature.

Table 3.4 – Relevance Criteria. List of relevance criteria identified in our searched literature, with their description. The number of occurrences (#) in the analyzed papers is also detailed.

Criterion	**Description**	**#**
Frequency	Amount of times that an event occurs during a period of time	30
Maintainability	Complexity of the operations associated with an event	20
Expensiveness	Consumption of computational resources associated with an event	16
Changeability	Analysis of divergences among the result of multiple occurrences of a same event	14
Error-proneness	Likelihood of an event to cause errors when they occur	13
Usage pattern	Characteristics of an event that tracks or deals with user requests	12
State variation	Amount of changes in the system state caused by an event	9
Concurrency	Amount of execution threads that are executed in parallel and share resources	8
Latency	The delay of the occurrence of an event	8

Table 3.5 – Goals, Relevance Criteria and Metrics. Association between groups of goals and relevance criteria, along with examples of goals and metrics used by the surveyed approaches. Cells in gray highlight the three most relevant criteria of each goal. The given examples are non-exhaustive.

Goal Group		Efficiency	Maintainability	Reliability	Security	Testability
Goal Examples		performance, energy saving, caching, improving resource consumption	bug finding, understanding, reuse, documentation, troubleshooting	health checking, fault tolerance, disaster recovery, adaptation, configuration fix	anomaly detection, data protection, malicious attack detection	testing (generation, validation, selection), reporting, verification
Criterion	Frequency	Number of occurrences in a period	Number of references and dependencies	Inter-arrival times	Changes in occurrence history	Number of occurrences in test case
	Maintainability	Number of operations involved	Static source code metrics	—	Contextual information of objects/classes	Fail test coverage
	Expensiveness	Execution time	Source code locations of expensive methods	CPU and heap utilization, processing times	Transaction duration	Depth of call stack
	Changeability	Number of repeated computations	Similarity between call graphs	Number of operations with cached results	Changes in contextual information	—
	Error-proneness	Number of failures of a component	Number of handled exceptions	Number of failures perceived by users	Increase of failures in a specific component	Number of failure assertions or exception-throwing statements
	Usage pattern	Changes in user navigational activity	—	Number of active users and idle/active intervals	Variations in the request payload for same operations	—
	State variation	I/O consumption per operation	Changes in the system state	Number of write operations performed	—	—
	Concurrency	Number of active users and threads	Number of references and dependencies	Number of race conditions	—	Number of locks per test case
	Latency	Processing and bandwidth consumption	—	Throughput	—	—

3.2.2 Generality

All the surveyed approaches have been proposed for a specific goal, such as identifying performance bottlenecks or bugs. Despite their sheer number and heterogeneity, 49 papers (76.5%) are limited to specific purposes, individual systems, particular architectural styles, or technologies, which couples the monitoring phase of these approaches to the analysis they perform (i.e. the goal of monitoring). These solutions are limited in terms of reuse because they are developed in an ad-hoc manner and require re-engineering work in order to adapt applications to obtaining tracing features.

From the surveyed approaches, 15 papers (23.5%) are flexible in terms of configuration. It means that they can be customized in terms of constraints, rules, and properties of the approach to better fit the user-specific needs. This flexibility is achieved by offering lower-level interfaces, functions, and probes that can be manually implemented and customized by users, such as in (JUNG et al., 2014; LEE; FORTES, 2016; DEVRIES; CHENG, 2018). However, only 5 papers (7.8%) increase the *generality* of monitoring by providing higher-level support so that users can achieve a domain-specific specification of the goal of monitoring, which is automatically implemented by the monitoring proposal. This is offered in the form of annotations, parameterization or domain-specific languages, e.g. in (GHEZZI et al., 2014; ANGELOPOULOS et al., 2016; CHRISTAKIS et al., 2017).

In addition to analyzing whether the monitoring can be customized, we also inspected if it relies on the particularities of specific technologies, programming languages, or execution environments. From the 64 papers, only 17 (26.5%) are technology-independent. The remaining 47 surveyed approaches (73.5%) rely on traces and events with properties and format tight to specific programming languages such as JavaScript (GONG; PRADEL; SEN, 2015) or Java (HUANG; LUO; ROSU, 2015). The same applies to approaches focused on software platforms such as Android (KANG et al., 2016) and Linux (SONG; LU, 2017). In these scenarios, traces are usually collected by instrumenting or profiling the execution platforms at a lower-level, and thus the monitoring approach becomes dependent on the platform specificities.

This means that most of the existing solutions that rely on monitoring are application-specific, i.e. while they carefully collect selected data at runtime for a particular purpose, the principles used to monitor data are not generalizable. However, the rationale behind the criteria used to filter traces is shared by the approaches.

3.2.3 Scalability

Monitoring and collecting information from a managed system impact on its performance. From all approaches, 31 papers (48.5%) explicitly mention that their solution implies an overhead to the observed system, such as in (SU et al., 2019; BRÖNINK; ROSENBLUM, 2016; MADSEN et al., 2016; ZHANG; ERNST, 2014; BIELIK; RAYCHEV; VECHEV, 2015; KANTERT et al., 2014). The overhead implied may vary according to the amount of information required to the analysis and the monitoring technique used to collect the execution traces. For example, Su et al. (SU et al., 2016b) report an overhead of 10× compared to running a Java-based application without the profiling technique used to detect functional clones, which relies on collecting detailed inputs and outputs of function calls. A different monitoring technique adopted by Hu et al. (HE; DAI; GU, 2018), which relies on collecting occurrences of operational system calls based on a Linux kernel (low-level) extension, reports an overhead of less than 1%. The remaining 33 surveyed approaches (51.5%), e.g. (CÁMARA; MORENO; GARLAN, 2014; TUN et al., 2018b; BASTANI; ANAND; AIKEN, 2015; LARSSON et al., 2019; CHEN et al., 2016; DENARO et al., 2015), do not even mention the impact of monitoring and gathering data from the managed system. This could raise questions of their practical feasibility because even in cases where a small amount of data is collected during system monitoring, like execution time and identification of the event (HE; DAI; GU, 2018; YANG et al., 2019), there is an impact on the memory consumption and processing time of the system.

Sampling and filtering have been demonstrated as the most used solution to reduce the trace size and enable faster trace analysis, adopted by 26 approaches (40.5%). However, the reduction of the amount of traces may lead to an unrepresentative sample and, consequently, inadequate to achieve the goal of monitoring. None of the surveyed approaches ensured the representativeness of the sample collected.

In terms of overhead evaluation, only 16 papers (25%) assessed the extent of the overhead. The reported overhead varies from negligible (0.8%–5%) (JUNG et al., 2014; KIM et al., 2016b; KANG et al., 2016; HAWKINS; DEMSKY, 2017; LEE; JUNG; PANDE, 2014; HE; DAI; GU, 2018; LIU; CURTSINGER; BERGER, 2016) to high impact (such as +16% or 41.7× the original execution) (SU et al., 2019; ALIABADI et al., 2017; BRÖNINK; ROSENBLUM, 2016; JENSEN et al., 2015; GONG; PRADEL; SEN, 2015; MADSEN et al., 2016; ZHANG; ERNST, 2014; BIELIK; RAYCHEV; VECHEV, 2015; SU et al., 2016b), which compromises the practical feasibility of the approach in

real-time scenarios. These high impact approaches are typically not supposed to be used at runtime because they demand detailed monitoring of the system and are usually focused on testing scenarios, possibly with production workloads. The results can be manually analyzed and applied by developers at design time to benefit the system in future executions.

Towards addressing the observed monitoring overhead, 16 of the approaches (25%) explored alternatives to reduce the impact of the monitoring activity. From these, six (JENSEN et al., 2015; MADSEN et al., 2016; HUANG; LUO; ROSU, 2015; SAMAK; TRIPP; RAMANATHAN, 2016; BIELIK; RAYCHEV; VECHEV, 2015; XIAO et al., 2014) apply specific tuning and optimization in the proposed algorithms to reduce the overhead, which is not possible to generalize to other approaches. For example, Madsen et al. (MADSEN et al., 2016), who focused on providing developers with detailed information about crashes at execution time, iteratively increase the instrumentation level on regions of code in which crashes are detected. In addition, Huang et al. (HUANG; LUO; ROSU, 2015), which monitors thread-related operations in Java-based applications, do not collect global traces but instead focus on the events of each thread separately. The remaining ten approaches (SU et al., 2019; KIM et al., 2016b; BRÖNINK; ROSENBLUM, 2016; GONG; PRADEL; SEN, 2015; SONG; LU, 2017; CHEN et al., 2014; JUNG et al., 2014; LEE; JUNG; PANDE, 2014; DELLA TOFFOLA; PRADEL; GROSS, 2015; ZHOU et al., 2016) make use of generic solutions such as sampling (simple systematic or random) and static analysis to filter out locations and avoid collecting the so-called useless traces.

The granularity level of the monitoring may play an important role in the overhead. In this regard, 40 of the surveyed approaches (62.5%) rely on monitoring function or method calls, which implies a considerable overhead given the high number of traces and the detailed information usually collected from method calls such as the input parameters and return. In 36 approaches (56%), the trace collection is performed based on low-level profiling and instrumentation, according to the programming language in which the approach is implemented such as based on Low Level Virtual Machine[2] (LLVM) (SONG; LU, 2017; KIM et al., 2016c) for C++ applications, Jalangi framework[3] (GONG; PRADEL; SEN, 2015; JENSEN et al., 2015) for Javascript-based applications, or ASM-based[4] instrumentation (DELLA TOFFOLA; PRADEL; GROSS, 2015; ZHANG; ERNST, 2014; CHEN et al., 2018; SU et al., 2016b) for Java applications. In addition, eight approaches (12.5%) either demand manual implementation of the tracing

[2]<https://llvm.org/>
[3]<https://jacksongl.github.io/files/demo/jalangiff/index.html>
[4]<https://asm.ow2.io/>

collection code (ANGELOPOULOS et al., 2016; GHEZZI et al., 2014; YANDRAPALLY; SRIDHARA; SINHA, 2015; XU et al., 2016; SU et al., 2016a) or provide ways of generating the implementation automatically (LEE; JUNG; PANDE, 2014; XIAO et al., 2014; DEVRIES; CHENG, 2018). However, code-level changes imply maintenance issues and increase the complexity of the software system base code.

For practical reasons, sometimes the monitoring is limited to high-level events (CÁMARA; MORENO; GARLAN, 2014; CLARK; BEAL; PAL, 2015; BARNA et al., 2017; YUAN; ESFAHANI; MALEK, 2014; BROCANELLI; WANG, 2017; DONG et al., 2018; ZHOU et al., 2019). Examples of this type of event are requests to a web server (CÁMARA; MORENO; GARLAN, 2014) or occurrences of failures in software components (ZHOU et al., 2019). This is the case of 15 of the surveyed approaches (23.5%) that rely on system or module-level monitoring. However, this may reduce the power of the analysis, given the lower amount of collected information.

3.2.4 Adaptability

The way that the monitoring is performed can change at runtime, either to focus on relevant traces or to reduce the overhead. Examples of changes are the metrics being collected, sampling configurations, or target locations oriented by a domain-specific analysis and occurrence of an event of interest. These are forms of adaptive monitoring.

Only seven of all surveyed approaches (11%) employ adaptive mechanisms to improve the monitoring efficiency (BRÖNINK; ROSENBLUM, 2016; GONG; PRADEL; SEN, 2015; MADSEN et al., 2016; SAMAK; TRIPP; RAMANATHAN, 2016; DELLA TOFFOLA; PRADEL; GROSS, 2015; BARNA et al., 2015; CASANOVA et al., 2014). In all cases, the adaptation is triggered based on a set of specific constraints or hard-coded rules and are thus not flexible to be modified. For example, Bronik et al. (BRÖNINK; ROSENBLUM, 2016) increase the reliability of collected data by dynamically placing probes in component connections according to fault detection. As its goal is fault localization, it changes the trace rate to have more information about which components are more susceptible to faults. It triggers the adaptation based on rules such as if a problem has been diagnosed in a component, probes are deployed to obtain a more accurate diagnosis.

Another example is the evaluation of the health of a managed resource (GONG; PRADEL; SEN, 2015). Because the instances in a distributed system can come and go

dynamically, when a new component appears or leaves the system, the monitoring component is capable of reflecting changes in the topology and keeping the reliability of the collected data. Adaptation is also used to dynamically reduce the level of the overhead. Barna et al. (BARNA et al., 2015), e.g., adapt the sampling technique to focus on specific code locations depending on the computational resources available for monitoring.

In addition, all the proposed adaptations are focused on controlling the monitoring overhead (BARNA et al., 2015; MADSEN et al., 2016; SAMAK; TRIPP; RAMANATHAN, 2016), or increasing the trace reliability for a specific purpose (BRÖNINK; ROSENBLUM, 2016; GONG; PRADEL; SEN, 2015; CASANOVA et al., 2014; DELLA TOFFOLA; PRADEL; GROSS, 2015). All of them are limited to changing configurations based on pre-defined setups or ad hoc assumptions, without considering the trade-off involved in the process. An example of such an assumption is that the performance overhead is reduced by merely reducing the target locations (BARNA et al., 2015), which may not be valid if the majority of the system executions are concentrated on the filtered locations. Thus, generic adaptation goals, such as maximizing the representativeness of the collected traces or dynamically managing the monitoring overhead, are not addressed by any of the surveyed approaches.

3.2.5 Discussion

Although runtime monitoring approaches have been employed to different goals and purposes, there are still limitations in terms of generality, scalability, and adaptability that should be addressed in order to achieve an effective monitoring approach. Thus, there is a need for a monitoring technique that:

1. can be applied generically and flexibly for different types of software systems and purposes, such as detecting and dealing with performance issues or energy bottlenecks, considering different levels of monitoring granularity;

2. can deal with the trade-off between the impact caused by the monitoring and its effectiveness in terms of data representativeness, relevance and location coverage; and

3. is able to respond to changing requirements and constraints in the monitoring component in order to maintain the monitoring effectiveness.

In addition, the development of reusable monitoring approaches that abstract and encapsulate monitoring functionality would reduce both the effort to develop new systems implementing these strategies as well as the probability of bugs in newly implemented solutions. Moreover, it would promote software reuse across different goals and domains. Based on the findings of our survey of monitoring-based approaches, we derived *Tigris-DSL*, a domain-specific language (DSL) designed to provide users with a standardized and comprehensive way to specify monitoring goals in terms of metrics and relevance criteria. TigrisDSL is founded on the relevance criteria presented in Tables 3.4 and 3.5 and is the basis of our monitoring solution (introduced in Section 4).

3.3 TigrisDSL: a Generic Way to Specify Relevance Criteria

TigrisDSL allows users to write monitoring filters by means of high-level relevance criteria. These relevance filters can be used to guide monitoring components to collect a set of relevant traces that are analyzed to achieve the goal of monitoring. It can be incorporated into any monitoring approach to specify monitoring requirements.

The Backus–Naur Form (BNF) grammar of TigrisDSL is presented in Listing 3.1. TigrisDSL is based on the recursive definition of a `filter`, which can be composed of multiple definitions (`filterdef`). Filter definitions are the main structure to allow users to inform which group of data from the relevance criteria should be considered (`modifier`) and the relevance criteria from the set of pre-defined criteria (`criterion`). It is important to highlight that these pre-defined criteria are based on our systematic literature review. Nevertheless, future versions of our DSL can include extended criteria or modifiers if those derived from our systematic analysis are considered not enough for the specifications of filters.

The semantics of the `modifier` should be specified by an approach employing our DSL. For example, `frequent` can be events that are executed in a frequency above the average, while `most frequent` can be the top 5% of the most frequent execution events (e.g. invoked methods or called functions).

In addition, `filters` can be combined to form a complex filter using `operators`, which are based on basic `set operations`, namely union, intersection, and subtraction. With these operations, it is possible to specify how the data from different groups of the relevance criteria can be combined to identify and filter a set of relevant events. To illustrate, we give examples of expressions written in TigrisDSL as follows.

Listing 3.1 – TigrisDSL Syntax Grammar. Presentation of the BNF grammar of the TigrisDSL language.

⟨*filter*⟩ ::= ⟨*filterdef*⟩ | (⟨*filter*⟩) | ⟨*filter*⟩ ⟨*operator*⟩ ⟨*filter*⟩

⟨*filterdef*⟩ ::= ⟨*modifier*⟩ ⟨*criterion*⟩ | ⟨*criterion*⟩

⟨*operator*⟩ ::= ∪ | ∩ | \

⟨*modifier*⟩ ::= more | less | most | least

⟨*criterion*⟩ ::= frequent | maintainable | expensive | changeable | error-prone | usage-pattern | state-variation | concurrent | latent

- `least frequent`, which indicates that in terms of frequency, only the least frequent events of the system execution should be traced.
- `more frequent ∪ most expensive`, which indicates that the monitoring should be focused on methods that are more frequent or most expensive, considering all the system events.
- `most changeable ∩ (most concurrent ∪ more error-prone)`, which indicates that only the most changeable events, which also have higher levels of concurrency or tend to cause errors, should be traced.

Depending on the semantics of `modifiers`, increasingly complex expressions can be specified as needed. Examples are presented as follows.

- `less changeable ∩ more frequent ∩ (more usage-pattern ∪ (more expensive ∩ less usage-pattern))`
- `(least changeable ∪ most changeable) ∩ more frequent ∩ (most usage-pattern \ less expensive)`

In order to demonstrate how relevance criteria can be used to abstract the desired behavior and filter execution traces for a specific purpose, we take a monitoring-based approach as an example (DELLA TOFFOLA; PRADEL; GROSS, 2015). This approach was identified in our literature survey. Della Toffola et al. (DELLA TOFFOLA; PRADEL; GROSS, 2015) proposed a method to identify memoization opportunities based on profiling method executions of applications. During this analysis, all the method calls are

Table 3.6 – Relevance Filter Examples. Example of a monitoring-based approaches specified in TigrisDSL.

Specification made by Della Toffola et al. (DELLA TOFFOLA; PRADEL; GROSS, 2015)	**Relevance Criterion**	**TigrisDSL-based Filter**
The program spends a non-negligible amount of time to process a method.	Expensiveness	`more expensive`
The program repeatedly provides equivalent inputs to a method, and this method repeatedly produces the same outputs for these inputs.	Changeability	`∩ least changeable`
The number of times that a result can be reused over the total number of cache lookups is at least 50%.	Frequency	`∩ more frequent`

filtered by processing three specifications, which are presented in the first column of Table 3.6. These specifications are informal and presented in a non-standardized way. The other two columns of this table show with which relevance criterion each specification is associated and a filter that represents it. The filters shown in the third column of Table 3.6 are less ambiguous and more concise than natural language. In addition, they are a generic and explicit way to express what sort of monitoring events of interest.

3.4 Final Remarks

The proposed language TigrisDSL is generic and abstract in the sense that it does not define the semantics of criteria such as frequency or expensiveness, as well as the meaning of more or less error-prone when comparing execution traces. Essentially, TigrisDSL captures the most representative concerns about the monitoring observed in the papers in our systematic literature review (criteria) and provides a syntactic construction to express a comparison among elements within a criterion in quality or degree (modifiers), along with a way to correlate and operate on top of different criteria to create relevance filters (operators). The filter expressions made using TigrisDSL can be used as input of any monitoring approach. To use TigrisDSL to create event filters, it is necessary to employ a mechanism that can interpret and translate these criteria, modifiers, and operators into quantitative and comparable metrics. We, in particular, propose a two-phase monitoring approach, which is guided by user-made specifications using TigrisDSL and provides

semantics to the language criteria, modifiers and operators. Our proposal, which is presented in the next chapter, is a step towards achieving an effective monitoring approach and provides a way to define and customize monitoring components.

4 ADAPTIVE FILTERING

As discussed, monitoring all execution traces in detail comes at the cost of extensive and detailed instrumentation, which causes a high overhead in software applications (MERTZ; NUNES, 2017b). Moreover, there are situations when it is infeasible to select software locations or executions that should be monitored a priori in detail (i.e. design-time), or such locations frequently change overtime. It thus becomes necessary to rely on an automated and adaptive process that can identify such executions of interest (e.g. the most frequent or more error-prone executions) with reduced performance overhead. Available monitoring approaches in this direction lack generality and fail in enabling software reuse across different domains with varying goals.

To address this, we propose a two-phase monitoring approach for collecting execution traces, which is a generic and customizable solution for monitoring. As presented in Figure 4.1, our approach is driven by user definitions supported by the proposed TigrisDSL, and thus provides semantics to all the terms of the TigrisDSL, such as criteria, modifiers, and operators. The first monitoring phase (described in Section 4.2) is *coarse grained*, focused on computing low-overhead metrics of event executions according to the specification of relevance criteria. The second phase (detailed in Section 4.3) takes as input the data collected in the coarse-grained phase. It processes the computed metrics from the previous phase and dynamically identifies the relevant events that are assumed to generate traces that are relevant for a given goal. These traces are collected in detail by a monitoring process that relies on a sampling strategy to control the overhead of monitoring. We implemented our proposed approach as a framework, namely Tigris (presented in Section 4.4), which serves as basis for conducting empirical studies (Section 4.5) with the aim of assessing different aspects of the proposed solution. Before detailing the phases of our monitoring approach, we next present a running example that is used throughout this section to illustrate details on how the proposed approach works.

4.1 Running Example: Application-level Caching

Previous work (MERTZ; NUNES, 2018) proposed an automated caching approach, which chooses and manages cacheable content according to the *Cacheability Pattern* (MERTZ; NUNES, 2017a). It targets the caching of *method-level content*. The automatically selected cache configuration is based on observations made by monitoring web applications

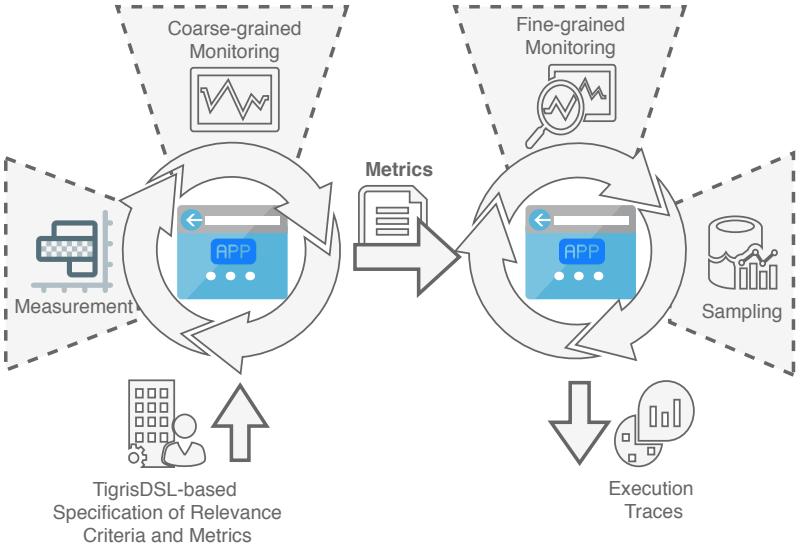


Figure 4.1 – Overview of the Two-phase Monitoring Approach. It shows the input provided in the first phase (Coarse-grained Monitoring) and the resulting output of the second phase (Fine-grained Monitoring).

at runtime, that is, a monitored application workload. This approach was conceived and implemented in the form of a caching framework, named *APLCache*, which seamlessly integrates the proposed solution to web applications.

One of the limitations of APLCache is that the overhead of the data tracking activity may significantly impact the application execution because it is necessary to monitor method inputs and outputs to make caching decisions. This was addressed by disabling the monitoring in situations when it is not possible to keep an acceptable overhead. Thus, the caching approach may lack information and provide outdated decisions.

We use APLCache as a running example to explain the details of our proposed approach in the following sections. Thus, the target type of event execution, in this case, are *methods*. APLCache is also used as a baseline for our evaluation, where we investigate the benefits of the two-phase monitoring to APLCache in terms of the overhead and relevance of the provided execution traces. Details about the evaluation are presented in Section 4.5.

4.2 Coarse-grained Monitoring

In its first phase, our proposed approach monitors the application in a way that it is possible to capture data that enables the identification of relevant traces with low overhead. First, it is necessary to instantiate the solution by providing domain-specific information in terms of relevance criteria and metrics. Based on such information, the

coarse-grained phase can collect data to identify the most relevant subset of events of the application. We next detail the manual input required by the coarse-grained phase and how it is used to monitor events.

The coarse-grained phase of the proposed monitoring approach relies on two inputs from the user: (a) the definition of high-level relevance filters using the TigrisDSL language; and (b) the selection of metrics to be used as a quantitative measurement of each relevance criterion referred in filters. To provide such information, users are provided with the guidance derived from our systematic literature review. Based on the user's goal, a set of suitable criteria from those presented in Table 3.4 must be selected to be used in relevance filters, and corresponding metrics should be indicated. The metrics presented in Table 3.5 are the most frequently used metrics in the literature, and can be used to represent the desired relevance criteria.

Considering that our running example is focused on identifying suitable method calls for caching, according to Table 3.5, its goal of monitoring is related to *efficiency*. Thus, the relevance criteria to achieve this goal should include the most popular criteria of this goal group, which are *frequency* and *expensiveness*. In addition, to find caching opportunities, method calls that always provide the same output given a specific input are well-suited for caching due to reuse opportunity (MERTZ; NUNES, 2018). Thus, *changeability* is also a criterion considered. As result, we specify the following relevance criteria.

```
(more frequent ∪ most expensive) ∩ least changeable
```

The TigrisDSL specification presented above contains different modifiers and operators. We are interested in filtering the *more* frequent method calls, i.e. the subset of method calls that ranks higher according to a specified metric for frequency, because caching methods that are frequently executed usually leads to performance improvements (MERTZ et al., 2020). In addition to the frequent method calls, we select the *most* expensive method calls, as caching a result that takes more time to be processed would lead to major performance benefits. However, managing cache consistency is a major challenge in the area (MERTZ et al., 2020). Thus, we only include method calls that are the *least* changeable because it would allow us to use a simple consistency strategy, such as an expiration time, and reduce the chances of caching stale content for longer periods. The detailed semantics of the modifiers (*more*, *most*, and *least*) and operators ($\cup$ and $\cap$), as defined by our proposed approach, are presented in Section 4.3.

Table 4.1 – Running Example Collected Metrics. Example of metrics collected and maintained for each method in the coarse-grained phase. Cells in gray highlight the `more frequent`, `most expensive` and `least changeable` according to the Grouping step.

Method	**Frequency**	**Expensiveness**	**Changeability**
ClinicService.findOwner(args)	12 (less)	180 (least)	6 (more)
ClinicService.updateOwner(args)	2 (least)	500 (most)	0 (most)
VisitController.newVisit(args)	50 (frequent)	250 (more)	12 (changeable)
ClinicService.findVets()	200 (most)	300 (expensive)	200 (least)
OwnerRepository.findAll()	100 (more)	200 (less)	90 (less)

For each criterion presented in the TigrisDSL specification, it is necessary to use of a quantitative metric so that the approach is able to track and give an objective interpretation to the criteria. As mentioned, this metric can be any of the metrics presented in Table 3.5, primarily those that match the selected criteria and the intended goal of monitoring. By taking into account the goal of our running example, possible metrics for frequency, expensiveness and changeability are, respectively, the absolute number of times a method occurs, the average time taken to execute a method and the absolute number of times that each pair of input and output of method occurs.

With a quantitative way of measuring the relevance criteria, the coarse-grained phase starts collecting these metrics from events at runtime. The coarse-grained monitoring results in a summary of the application in terms of statistics about all the event executions of the system. Collected metrics about the execution events are maintained in memory. Consequently, these estimations of the metrics are computationally cheaper than the metrics and do not demand recording individual traces.

These metrics are used as a reference to assess how expensive, frequent, and changeable methods are. For example, considering the computation pattern (changeability), methods with higher standard deviation are less changeable than others because it might indicate that there are equal outputs that are (much) more frequent than others, causing the standard deviation to be high. In our running example, by monitoring events in a coarse-grained manner, our approach gives as result the information presented in Table 4.1, where a single metric value represents each relevance criterion for each method.

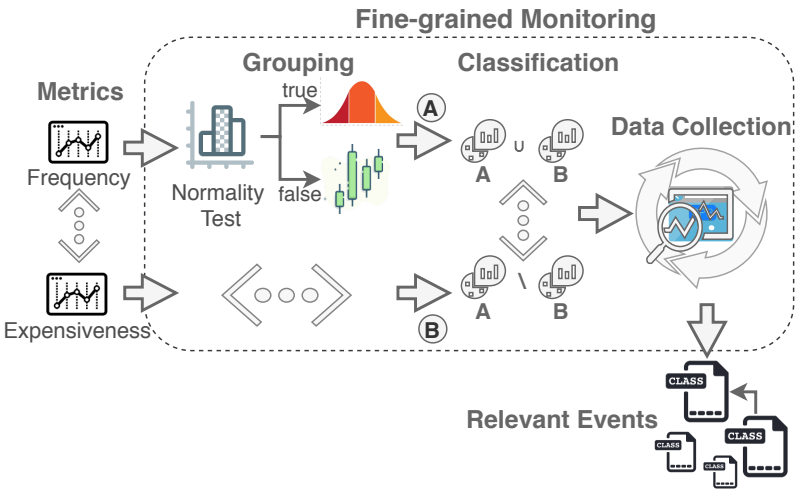


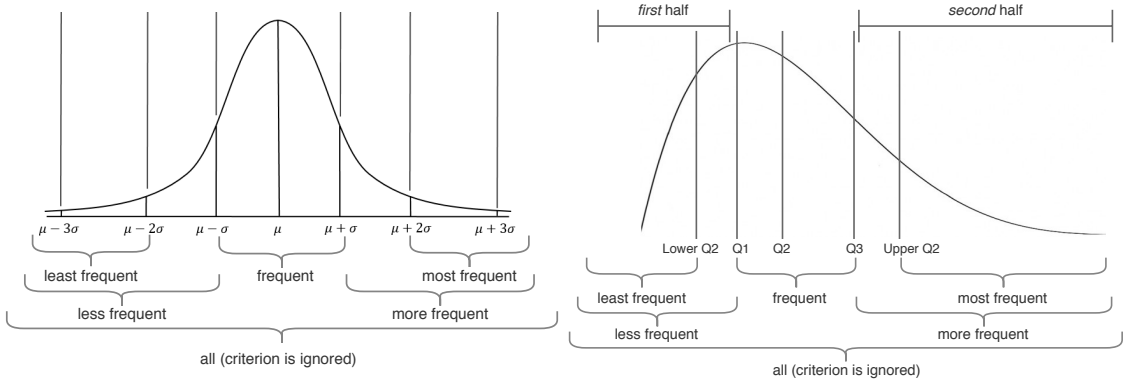
Figure 4.2 – Fine-grained Monitoring Steps. Illustration of the three steps that comprise the fine-grained monitoring: Grouping, Classification and Data Collection.

4.3 Fine-grained Monitoring

By having a summary of the application in terms of statistics about all the event executions of the system and the relevance criteria, it is possible to identify relevant events that should be monitored at runtime in the fine-grained phase. In this phase, our approach uses the data collected in the previous phase to determine which parts of the monitored software system are relevant to the goal of monitoring and thus should be inspected in detail. Figure 4.2 presents an overview of the steps performed during the fine-grained monitoring. The identification of relevant parts happens periodically and consists of processing the calculated metrics to group the values collected in partitions (*Grouping* step) and using such partitions and relevance filters to classify which types of event executions are relevant (*Classification* step). This process generates or updates a list of relevant types of event, which are monitored at a fine-grained level (*Data Collection* step).

Grouping The relevance filters specified using TigrisDSL refer to types of events that satisfy a set of criteria, with each criterion possibly having a modifier (*least*, *less*, *more* and *most*). A criterion with a modifier is used to indicate if events of interest are those that have an associated metric value that is very low, low, average, high, or very high. This way of referring to execution events and their metric values is subjective, and the *Grouping* step uses the distribution (average and spread) of the values of each metric to give an objective meaning.

First, to understand how metric values are distributed, we apply a normality test to the data set (the set of collected metrics for each event execution) to verify whether



(a) Grouping based on Normally Distributed Data

(b) Grouping based on Quantiles

Figure 4.3 – Frequency Groups. Using the frequency relevance criteria as example, the charts presents the semantics given for the different modifiers of TigrisDSL. The semantics considers the splitting of the data into groups and the normality of the data.

the observed values of a criterion follow a normal distribution. This process can be performed based on different statistical tests. We adopt the non-parametric statistical test Kolmogorov-Smirnov ($p > 0.05$), which is widely used and performs better with large sample sizes—generally the case when dealing with execution traces. If the Kolmogorov-Smirnov significance value is higher than the alpha value (0.05), then the data follows a normal distribution.

Based on the shape of the distribution, we have two different strategies to classify data into five partitions. In the case of a normal distribution, we group the data based on K standard deviations below or above the mean to create five groups of data. In case the data do not follow a normal distribution, we apply the quantiles strategy by calculating the Q1, Q2, and Q3 of the sample to obtain three groups of data: the lower quarter (below Q1), the interquartile range (between Q1 and Q3) and the upper quarter (above Q3). Then, we calculate the median of the upper and lower quarter and split each of them according to that value, leading to five groups of data. These strategies to group values are presented in Figure 4.3.

In our running example, we must group methods to identify those that are `more frequent`, `most expensive`, and `least changeable`. The normality test has shown that all distributions are not normal (this is expected as our running example considers a small set of methods). Then, using quantiles to classify the data, we obtain the groups presented in Table 4.1. In this table, we also highlight the group of interest.

Classification The Grouping step evaluates each criterion individually and creates groups of events. The Classification step relies on these groups to evaluate the filters specified in

TigrisDSL. This is done by evaluating the provided filter expressions that contain set operations (union, intersection, and subtraction). The filter specified in our running example is `(more frequent ∪ most expensive) ∩ least changeable`. Considering the groups presented in Table 4.1, the only method that is relevant considering our goal is `ClinicService.findVets()`, which satisfies the informed filter. This is the method that should be monitored in detail, in the last step of this phase.

Data Collection The list of relevant types of event is updated periodically by the previous steps. It is used in the Data Collection step to perform in-depth monitoring of the relevant event types, obtaining details of their execution, such as returned objects and parameters provided as input. Although filtering a subset of relevant methods reduces the overhead of monitoring, it may still impact the application performance if the traffic is concentrated in those methods supposed to be relevant. Thus, in addition to filtering, traces are collected according to a specified *sampling rate*, which bounds the cost of monitoring. Consequently, our proposal allows the user to achieve an efficient trade-off between sample relevance and monitoring overhead. Considering our running example and a sampling rate of 50%, the method calls that are traced are those highlighted in Table 4.2, that is, 50% of the calls to the `ClinicService.findVets()` method.

4.4 Tigris Framework

The conceptual approach described above provides a generic means for monitoring software systems and can be instantiated to particular technologies. However, to provide concrete support to this activity and evaluate our approach, it has been implemented as a framework, namely Tigris[1], using particular technologies. Tigris is implemented in Java and thus can be instantiated and integrated into monitoring-based approaches to leverage the monitoring results of Java-based approaches and applications. This choice is due to our previous programming experience and available tools that were adopted as part of our implementation. To collect data during the coarse-grained and fine-grained monitoring phases, we intercept method executions with AspectJ[2], which allows Tigris to acquire lightweight and dynamic software metrics without changing the base code. Metrics available in the framework are the most frequently used metrics in the literature, listed

[1]<http://prosoft.inf.ufrgs.br/git/Repository/Tree/d32a32bf-1a9e-45e1-8117-b4d2adf3c106>
[2]<https://eclipse.org/aspectj/>

Table 4.2 – Sample Execution of an Application. Cells in gray highlight the occurrences (#Occ.) of a specific method call in the application execution sequence (#Seq.) that would be traced at a sampling rate of 50%.

#Seq.	Method Call	#Occ.
1	VisitController.newVisit("X")	1
2	OwnerRepository.findAll()	1
3	ClinicService.findVets()	1
4	ClinicService.findVets()	2
5	ClinicService.findVets()	3
6	ClinicService.updateOwner("X")	1
7	OwnerRepository.findAll()	2
8	ClinicService.findVets()	4
9	VisitController.newVisit("X")	2
10	ClinicService.updateOwner("X")	2
11	ClinicService.findVets()	5
12	ClinicService.findVets()	6
13	ClinicService.findVets()	7
14	ClinicService.updateOwner("X")	3
15	ClinicService.findVets()	8
16	OwnerRepository.findAll()	3
...	...	...
$n-1$	ClinicService.findVets()	$n-1$
n	ClinicService.findVets()	n

in Table 4.3.

The metrics used to assess relevance criteria might be costly to be collected, because they may require "heavy" information to be calculated, such as parameter and return values of event executions. However, as the first phase monitors all system events, this monitoring is coarse grained and collects *lightweight* versions of the metrics to avoid disrupting the application execution. Therefore, although these metrics have some impact on the application execution, it is much lower than collecting fine-grained (and heavy) information and recording execution traces.

The coarse-grained monitoring phase is thus limited to the computation of *estimations* of the metrics—listed in the third column of Table 4.3—which are kept as a single in-memory number, continuously updated whenever a new event is intercepted at runtime. For example, for execution time and invocation frequency, the estimations are the mean execution time and the absolute number of all the calls of a method, respectively. This is opposed to the complete distribution of these metrics with traces giving information such as which inputs lead to high execution times.

Computation pattern, in turn, consists of the analysis of common computations of

Table 4.3 – Tigris Framework Metrics. List of names and descriptions provided by the Tigris framework, together with how these metrics are estimated.

Metric	Description	Estimation
Concurrency Level	the number of times a method is executed concurrently	mean number of active threads during all calls of the method
Computation Pattern	the number of times that each pair of input and output of method occurs	standard deviation of the return size of all calls of a method
Energy Consumption	the amount of energy demanded by a method	mean estimate of energy consumption of all calls of the method
Error level	the number of times the execution of a method thrown exceptions	absolute number of exceptions raised by all calls of the method
Execution Time	the time taken to execute a method	mean execution time of all calls of the method
Inter-Arrival Time	the time taken between executions of a method	mean time between each call of a method and the next
Invocation Frequency	the number of times a method occurs	absolute number of calls of a method
Memory Consumption	the amount of memory demanded by a method	mean estimate memory consumption of all calls of the method
User Behavior	the number of times a method is shared among different users	absolute number of user sessions that lead to calls of a method

a method, that is, the identification of frequent pairs of inputs and output of a method. This is expensive to be computed as it requires tracing and comparing each method call accompanied by the parameter values and the method return value. Because the goal of this metric is to identify repeated computations and the output of a method is usually highly dependent on its input, our estimation relies on observing the return values of different calls of a method in terms of allocation size in memory and then computing the standard deviation of these values. Thus, for example, if the standard deviation is low, it means that most of the returns of the method calls are the same, thus less changing.

For estimations that would demand to store the list of observed values such as those based on standard deviation and average, to keep a single in-memory number updated on-the-fly, we compute mean and standard deviation based on online and incremental algorithms (KNUTH, 1997). The TigrisDSL specification, as well as metrics to be used while assessing execution events can be configured through property files and annotations. An example of such configuration is presented in Listing 4.1.

Listing 4.1 – Tigris annotation-based configuration example.

```
@TigrisConfiguration(
   logRepository = RepositoryType.MEMORY,
   staticMetricFile = "petclinic.csv",
   samplingPercentage = 0.5,
   analysisFixedDelay = 120)
@TigrisCriteria(
   criteria = "more frequent U more expensive",
   granularity = GranularityType.METHOD,
   frequencyMetric = Metrics.INVOCATION_FREQUENCY,
   expensivenessMetric = Metrics.EXECUTION_TIME,
   changeabilityMetric = Metrics.COMPUTATION_PATTERN)
@ComponentScan(allowed = "org.springframework.samples.petclinic.*",
   denied = "org.springframework.samples.petclinic.model.*")
public class Configuration {...}
```

The sampling rate can be controlled and adjusted at runtime (using a function provided by our implementation), varying from 0% (no monitoring) to 100% (complete monitoring of selected methods). This function can be used to adapt the sampling according to the overhead tolerance and monitoring coverage requirement. In addition, the amount of time in which the framework should keep collecting lightweight metrics (first monitoring phase) of the event executions until triggering the process to select methods to monitor in detail (second monitoring phase) can be controlled and adjusted at runtime through an input parameter.

New criteria and metrics can be included by extending and implementing interfaces provided by the framework. The same interfaces are used to customize and define how metrics should be calculated. In addition to the usage of TigrisDSL filters to identify relevant event executions based on the goal of monitoring, Tigris also offers customizations. For example, it is possible to set up the framework to focus on specific monitoring locations, which can improve the set of events to be evaluated as relevant as well as exclude events that must not be monitored, thus reducing the time overhead for tracking them. Tigris also supports loading output metrics from Understand[3], to evaluate criteria that are based on static metrics.

Next, we describe our evaluation procedure and then discuss the obtained results.

4.5 Evaluation: Adaptive Monitoring for APLCache

In order to evaluate our proposed solution for monitoring, we perform an empirical evaluation by instantiating Tigris as monitoring support for APLCache, the application-

[3]<https://scitools.com/>

level caching approach that was used to illustrate the phases of our framework.

4.5.1 Study Settings

APLCache (MERTZ; NUNES, 2018) is used to monitor web applications to identify cacheable methods with the goal of improving application performance. The monitoring of APLCache captures execution traces of each method call with its input parameters and return. Monitoring the application has a performance cost, so we aim with Tigris to reduce the monitoring cost without compromising its effectiveness of identifying cacheable methods. Therefore, the original version of APLCache is used as a baseline.

Our evaluation aims to answer three research questions, presented in Table 4.4. In this table, we also detail the metrics used to answer each research question. In RQ1, we assess how Tigris reduces the cost of monitoring. However, as the application performance is influenced not only by the monitoring but also by the identified cacheable spots that are discovered based on the collected traces, we also compare the performance of our target web applications using APLCache with full monitoring and with Tigris. In RQ2, we compare the effectiveness of the cacheable spots identified using the traces collected using the two alternatives under evaluation. Improved effectiveness to identify cacheable methods is desirable. Nevertheless, our ultimate goal is to monitor the application with lower costs, without compromising the results of the analysis, that is, we aim to collect a subset of execution traces that would lead to the same results as if we had collected the complete set of traces. Thus, we also assess the effectiveness (precision and recall) of APLCache with Tigris using as ground-truth the cacheable spots identified by APLCache with full monitoring. Finally, in RQ3 we assess how Tigris copes with workload variations over time in terms of changes in the relevance evaluation performed by its first phase (coarse-grained monitoring) and, consequently, in the selection of execution traces to be collected in detail by its second phase (fine-grained monitoring).

To assess both versions of APLCache, we must select target web applications, simulation parameters, and workloads. To avoid bias, we follow the design choices of the study previously performed to evaluate the original version of APLCache (MERTZ; NUNES, 2018). In the study, we use three target open-source web applications[4], presented in Table 4.5, which summarizes the general characteristics of each target system.

[4] Available at <http://www.cloudscale-project.eu/>, <https://github.com/SpringSource/spring-petclinic/> and <http://www.shopizer.com/>.

Table 4.4 – Research Questions and Metrics. List of the research questions of our evaluation and the metrics used to answer each of them.

Research Question	Metric
RQ1. What performance gain does Tigris provide?	**M1-1.** Throughput (average number of requests handled per second) of the target applications with monitoring (and no caching) **M1-2.** Throughput of the target applications using APLCache
RQ2. What is the effectiveness achieved with execution traces collected by Tigris?	**M2-1.** Number of identified caching opportunities **M2-2.** Hit ratio **M2-3.** Number of hits **M2-4.** Precision **M2-5.** Recall **M2-6.** F-measure
RQ3. How does Tigris cope with workload variations over time?	**M3-1.** Difference in the number methods selected by the first phase of the approach (coarse-grained monitoring) through multiple monitoring cycles in sequence

We also keep the same APLCache parameters, such as cache provider and eviction policy.

For all the RQs we consider performance test suites in the form of workload simulations. The simulation starts with five simultaneous users continuously navigating through the application based on a navigation pattern that falls into a specific distribution (transition table). Then, at every second, we randomly add or remove a number of users (from 1 to 10) from the simulation until all the users make the total of 60k requests to the application. We adopt a minimum number of 5 simultaneous users to keep a minimum of concurrency in the workload, and a maximum of 20 to avoid disruptions in the response times due to struggles from the web server. To stimulate changes in the workload, we created three variations of navigation patterns for each application, and whenever a user is added to the simulation, we randomly decide which of the three workload variations the new user should follow. To keep a fresh selection of execution traces collected in detail by the second phase (fine-grained monitoring), the processing of lightweight metrics is triggered every two minutes during the simulation. These parameters were empirically chosen based on a sample application, which was not used in our evaluation.

To increase the reliability of the results, we create the workload with the above settings once and execute the exactly same sequence of requests and user variations per second ten times. To evaluate how changes in the workload may impact in our proposal, the simulation is segmented in three *monitoring cycles* of 20k requests, we collect and

Table 4.5 – Target Systems. List of the target web applications used in our evaluation, together with their application domain and size. Size is detailed by the number of lines of code (LOC) and the number of files.

Project	**Domain**	**LOC**	**# Files**
Cloud Store	File Synchronization	7.6 K	98
Pet Clinic	Sample application	6.3 K	72
Shopizer	e-Commerce	111.3 K	946

inspect from the simulation the subset of methods selected by the first phase of Tigris as well as the cacheable opportunities found by APLCache. Thus, for all the metrics in the results we report mean and standard deviations of these multiple executions. For all the simulations, we use two machines located within the same network, one machine (16G RAM, Intel i7 2GHz) for the Tomcat[5] web server and MySQL[6] database, and one machine (16G RAM, Intel i5 2.4GHz) to handle the performance test suite with JMeter[7].

To configure Tigris, we must specify relevance filters in TigrisDSL. We used as a basis the Cacheability Pattern (MERTZ; NUNES, 2017a), which indicates a set of criteria for deciding whether a method should be cached. We assess two alternative filters in our study, presented in Table 4.6. The *Restricted Filter* leads to a subset of methods selected by the *Expanded Filter* to be monitored in the second phase of our framework. The specified relevance filters indicate that four relevance criteria are considered and, for each of them, we must indicate the metric estimations to be used. We selected the metrics Invocation Frequency, Computation Pattern, User Behavior and Execution Time, for the criteria Frequency, Changeability, Usage pattern and Expensiveness, respectively. The Tigris second phase also receives as parameter a sampling rate. We selected six sampling rates (ranging from 1% to 100%) to understand how the number of collected traces can impact in the analysis of the traces.

4.5.2 Results

We next present and analyze the results obtained by running the simulations with each of our three target applications and collecting the specified metrics.

[5]<http://tomcat.apache.org/>
[6]<https://www.mysql.com/>
[7]<http://jmeter.apache.org/>

Table 4.6 – Relevance Filter Specification. Specification of the two filters used in our evaluation, namely Restricted Filter and Expanded Filter.

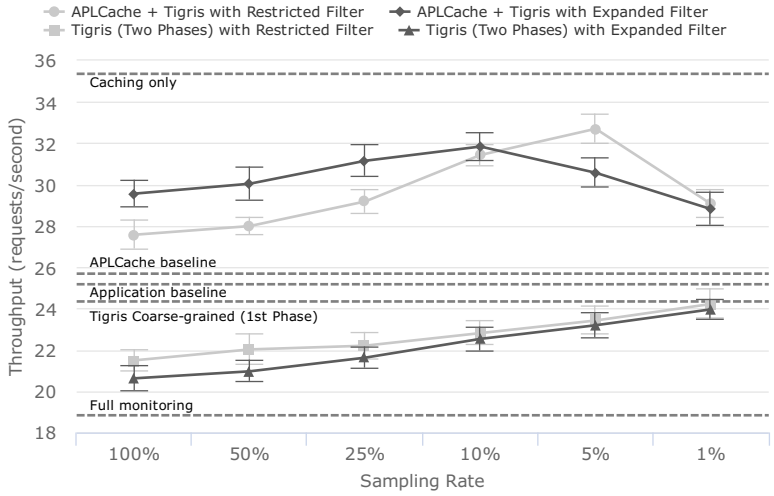
Name	TigrisDSL-based Specification
Restricted Filter	`less changeable ∩ more frequent ∩ (more usage-pattern ∪ (more expensive ∩ less usage-pattern))`
Expanded Filter	`(less changeable ∪ changeable) ∩ (more frequent ∪ frequent) ∩ ((more usage-pattern ∪ usage-pattern) ∪ (more expensive ∪ expensive))`

4.5.2.1 RQ1. What performance gain does Tigris provide?

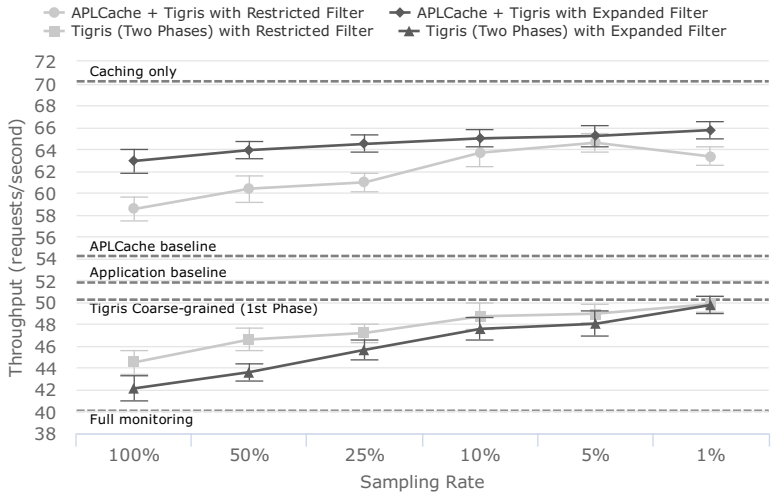
Our first results consist of assessing how much Tigris reduces the cost of monitoring, considering its ability to filter and reduce the scope of monitoring. For that, we compare the performance of each application using Tigris (restricted and expanded filters) to the baselines with full monitoring and no monitoring. The information collected with full monitoring leads to the ground-truth decisions made based on execution traces, while no monitoring provides a baseline of the application performance without any overhead. This analysis allows us to assess how far our decisions are from the ground truth and the performance costs associated with them. To compare the application performance under these different monitoring configurations, we use *throughput*, which is measured by calculating the average number of requests handled per second throughout the simulation. Thus, high throughput (i.e. close to the throughput achieved with no monitoring) indicates an effective monitoring configuration, as more requests can be processed within the same period of time.

The results are presented in Figure 4.4. In each chart, we present the throughput of executing the application with no and full monitoring, which serve as references. We also detail the cost of running the application only with the Tigris first phase activated. All these are presented as horizontal lines because they do not vary in terms of the sampling rate. The throughput of running Tigris with the two considered filters is presented with its varying results according to the sampling rate.

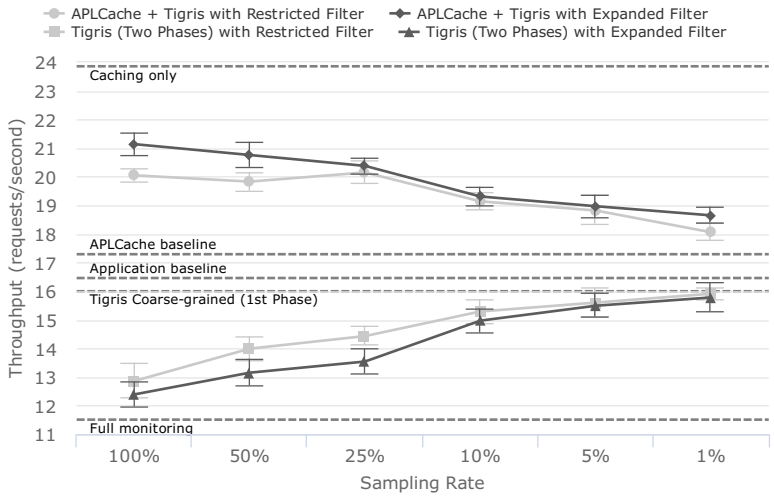
As expected, monitoring an application causes an overhead, even with Tigris. Considering the different applications and varying evaluated configurations, we observed that the minimum overhead was obtained with the restricted filter and 1% sampling rate for Shopizer (Figure 4.4c). This configuration achieves a throughput of 15.91 req/s vs. 16.47 req/s obtained with no monitoring, resulting in 3.42% of performance impact. The



(a) Cloud Store



(b) Pet Clinic



(c) Shopizer

Figure 4.4 – RQ1: Throughput by Sampling Rate. Performance of each application executed with and without APLCache using varying monitoring configuration.

maximum observed overhead came from the combination of expanded filter and 100% sampling rate for Shopizer (Figure 4.4c), which led to a throughput of 12.39 req/s—24.7% lower than the baseline with no monitoring (16.47 req/s). In brief, Tigris resulted in a performance penalty ranging from 3.4% to 24.7%, compared to the baseline without monitoring.

However, the framework largely reduces the cost of full monitoring. The overhead of this configuration ranges from 22.6% (its throughput is 40.09 req/s for Petclinic as seen in Figure 4.4b, vs. 51.81 req/s of the baseline) to 29.9% (its throughput is 11.53 req/s for Shopizer as seen in Figure 4.4c, vs. 16.47 req/s of the baseline).

Assessing solely the cost of the Tigris, we observe that the overhead is caused mostly by its second phase, in which methods are fine-grained monitored. The maximum overhead of the coarse-grained monitoring (first-phase) was 3.3%, because Cloud Store achieved 24.32 req/s when compared to the baseline without monitoring of 25.16 req/s.

The performance overhead of the second phase varies according to the results of the first phase and the sampling rate in which it collects traces. The overhead decreases when the sampling rate decreases because fewer executions are traced. For example, when collecting 1% of the traces with the restricted filter, the overhead is marginal for all three applications, ranging from 3.4% to 3.8% when comparing to the baseline with no monitoring in Figure 4.4, being similar to the coarse-grained phase only. Collecting traces with the expanded filter at a 100% sampling rate, i.e. no sampling is performed, largely increases the overhead to the application performance, ranging from 18.0% to 24.7%, when comparing to the baseline with no monitoring in Figure 4.4.

To compare the overall performance of APLCache with full monitoring and with Tigris, we also assess the throughput. However, the overall performance is not only affected by the monitoring but also by the opportunities cached based on the analysis of collected traces. Obtained results are presented in Table 4.7 (column Throughput) and Figure 4.4. We observe that the throughput achieved by APLCache when supported by Tigris is higher than using full monitoring, with both filters and any sampling rate. Tigris improves the throughput of all the three target applications with gains ranging from 4.4% to 27.4%, in comparison to APLCache with full monitoring. Because Tigris can filter a subset of relevant methods and consequently monitor fewer methods in detail, the overhead of monitoring tends to decrease.

In Pet Clinic (Figure 4.4b), the overall performance typically increases as the sampling rate decreases. Nevertheless, this does not always hold. With fewer execution traces

Table 4.7 – Simulation Results. Results of executing each application with full monitoring, restricted filter and expanded filter. Reported metrics (average of all the ten executions) for the different sampling rates (Sample) are throughput, hit ratio (HR), number of hits (Hits), number of cacheable opportunities (Cacheability), precision (Pr.), recall, and F-Measure (F1). Throughput and Cacheability are shown in absolute and relative (percentage change in comparison with full monitoring) terms. Cacheability, precision, recall and F-Measure are presented as the average of all three monitoring cycles.

	Monitoring	Sample	Throughput		HR	Hits	Cacheability		Pr.	Recall	F1
Cloud Store	Full Monitoring		25.6		0.96	45,877	7.67				
	Restricted Filter	100%	27.5	+7.3%	0.96	30,252	3.67	-52.2%	1.0	0.47±0.04	0.64±0.03
		50%	27.9	+9.0%	0.95	27,036	2.67	-65.2%	1.0	0.34±0.05	0.51±0.05
		25%	29.1	+13.7%	0.97	25,924	2.33	-69.6%	1.0	0.30±0.06	0.46±0.07
		10%	31.4	+22.4%	0.98	26,100	2.33	-69.6%	1.0	0.30±0.06	0.46±0.07
		5%	32.7	+27.4%	0.97	25,532	2.33	-69.6%	1.0	0.30±0.06	0.46±0.07
		1%	29.0	+13.2%	0.98	17,208	1	-87.0%	1.0	0.13±0.01	0.23±0.01
	Expanded Filter	100%	29.5	+15.1%	0.96	42,144	6.33	-17.4%	1.0	0.83±0.14	0.90±0.08
		50%	30.0	+17.0%	0.94	41,668	6.33	-30.4%	1.0	0.70±0.18	0.81±0.13
		25%	31.1	+21.3%	0.98	36,060	4	-47.8%	1.0	0.52±0.04	0.68±0.03
		10%	31.8	+24.0%	0.94	33,384	3.67	-52.2%	1.0	0.48±0.09	0.64±0.09
		5%	30.5	+19.1%	0.96	25,524	2.33	-69.6%	1.0	0.30±0.10	0.46±0.11
		1%	28.81	+12.2%	0.95	20,328	1.33	-82.6%	1.0	0.17±0.09	0.29±0.12
Petclinic	Full Monitoring		54.17		0.94	52,860	4				
	Restricted Filter	100%	58.5	+8.0%	0.94	41,040	2.33	-41.7%	1.0	0.58±0.14	0.73±0.11
		50%	60.3	+11.4%	0.94	41,404	2.33	-41.7%	1.0	0.58±0.14	0.73±0.11
		25%	60.9	+12.5%	0.94	38,076	2	-50.0%	1.0	0.50±0.00	0.66±0.00
		10%	63.6	+17.5%	0.94	38,488	2	-50.0%	1.0	0.50±0.00	0.66±0.00
		5%	64.5	+19.2%	0.94	38,980	2	-50.0%	1.0	0.50±0.00	0.66±0.00
		1%	63.3	+16.8%	0.94	26,224	1	-75.0%	1.0	0.25±0.00	0.40±0.00
	Expanded Filter	100%	62.9	+16.1%	0.94	52,616	4	0.0%	1.0	1.00±0.00	1.00±0.00
		50%	63.9	+17.9%	0.94	52,772	4	0.0%	1.0	1.00±0.00	1.00±0.00
		25%	64.4	+18.9%	0.94	49,032	3.67	-8.3%	1.0	0.91±0.14	0.95±0.08
		10%	64.9	+19.9%	0.94	46,900	3	-25.0%	1.0	0.75±0.00	0.85±0.00
		5%	65.1	+20.3%	0.94	46,652	3	-25.0%	1.0	0.75±0.00	0.85±0.00
		1%	65.7	+21.3%	0.94	45,824	3	-25.0%	1.0	0.75±0.00	0.85±0.00
Shopizer	Full Monitoring		17.30		0.92	691,352	24.33				
	Restricted Filter	100%	20.0	+15.9%	0.92	419,656	16.33	-32.9%	1.0	0.66±0.03	0.80±0.02
		50%	19.8	+14.6%	0.93	257,012	13.33	-45.2%	1.0	0.54±0.01	0.70±0.01
		25%	20.1	+16.4%	0.92	182,520	10	-59.0%	1.0	0.40±0.03	0.58±0.03
		10%	19.1	+10.6%	0.94	71,460	6	-75.3%	1.0	0.24±0.01	0.39±0.02
		5%	18.8	+8.7%	0.93	35,892	4	-83.6%	1.0	0.16±0.02	0.27±0.03
		1%	18.0	+4.4%	0.91	17,998	1	-95.9%	1.0	0.04±0.00	0.07±0.01
	Expanded Filter	100%	21.1	+22.2%	0.91	513,960	17.67	-27.4%	1.0	0.74±0.17	0.84±0.11
		50%	20.7	+20.0%	0.91	350,072	14.33	-41.1%	1.0	0.60±0.14	0.74±0.11
		25%	20.3	+17.8%	0.92	184,232	10.33	-57.5%	1.0	0.43±0.10	0.60±0.10
		10%	19.3	+11.7%	0.91	68,660	6	-75.3%	1.0	0.25±0.06	0.40±0.08
		5%	18.9	+9.7%	0.91	32,818	3.33	-86.3%	1.0	0.14±0.03	0.24±0.05
		1%	18.6	+7.8%	0.93	18,412	1	-95.9%	1.0	0.04±0.00	0.08±0.01

(lower sampling rate), the identification of cacheable opportunities is less consistent with the actual behavior of the application, thus reducing the gains of caching the application. This can be seen in the other two applications. The overall performance of CloudStore (Figure 4.4a) increases up to 5% sampling rate for the restricted filter and 10% sampling rate for the expanded filter, and then it decays. The overall performance of the Shopizer

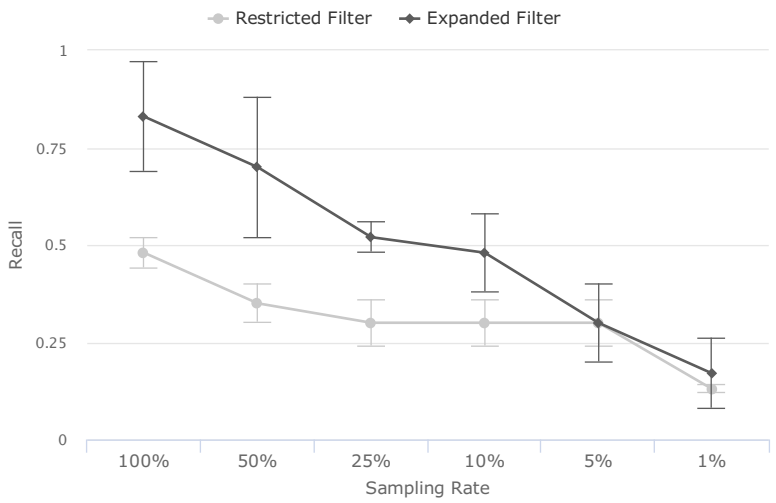
application (Figure 4.4c), in turn, tends only to decrease as the sampling rate decreases. In most cases, the restricted filter achieves worse results than the expanded filter. This means that, although the restricted filter leads to fewer methods to be monitored in the second monitoring phase, the set of fine-grained monitored methods causes the identification of caching opportunities that provide lower increases in the performance. That is, the balance between the cost of monitoring and the gain of caching is higher with the expanded filter than with the restricted filter.

RQ1: Findings. The impact of full monitoring applications is high, causing performance impacts ranging from 22.6% to 29.9% when compared to the baseline of no monitoring. Tigris, with varying configurations, implies a lower impact to the application performance, with values ranging between 2.9% and 24.7%, when compared to the baseline of no monitoring. The overhead of Tigris is mostly caused by fine-grained monitoring, which can be reduced by decreasing the sampling rate. With respect to overall performance with enabled caching, it provides improvements ranging from 4.4% to 27.4% in relation to monitoring all method calls with APLCache. The relevance filter and sampling rate provide a configuration space that allows the approach to achieve the best trade-off between the cost of monitoring and the quality of the analysis of execution traces.

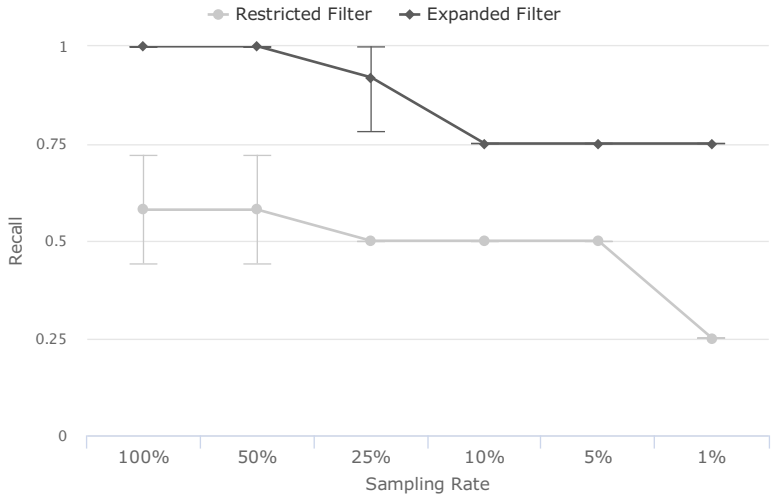
4.5.2.2 RQ2. What is the effectiveness achieved with execution traces collected by Tigris?

The previous research question has shown that the performance gains depend on which methods are cached. This decision is made by APLCache, which analyzes the collected execution traces. Therefore, we now evaluate the cached opportunities identified with each monitoring configuration. We assess the number of cached opportunities, the hit ratio, and the number of hits. These are presented in Table 4.7 (columns Cacheability, HR and Hits, respectively).

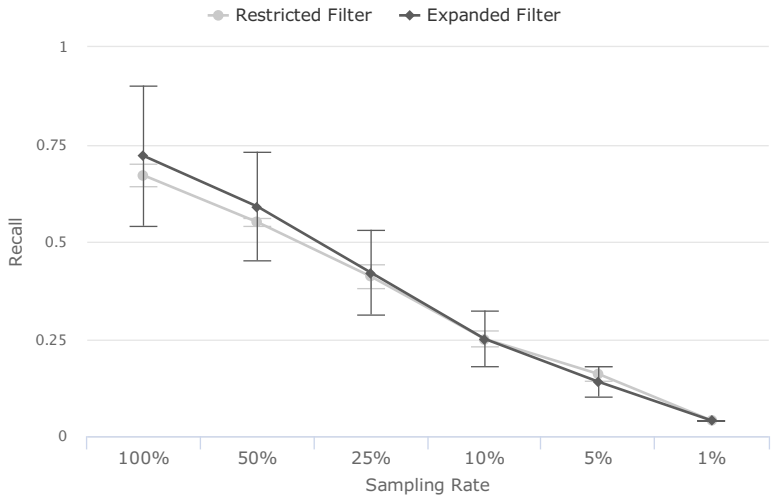
Although the number of selected methods to be cached using Tigris is smaller in comparison to APLCache monitoring, we observe that this number changes according to the filtering and sampling configuration. For all applications, restricting the filter and decreasing the sampling rate may significantly reduce the number of caching opportunities. The amount of information collected is reduced, and thus the accuracy of APLCache may be compromised.



(a) Cloud Store



(b) Pet Clinic



(c) Shopizer

Figure 4.5 – RQ2: Recall by Sampling Rate. Recall obtained for each combination of filter (Restricted Filter and Expanded Filter) and sampling rate.

None of the combinations of filters and sampling rates can identify all the opportunities initially identified by the original APLCache. However, the throughput is still improved due to the filtered monitoring guided by the coarse-grained phase of our approach. The absolute number of hits varies according to the number of cacheable opportunities. The lower number of hits follows reduced cacheable opportunities. The hit ratio remains almost the same along with all the simulations, and small variations are due to the variations in cacheable opportunities found at different monitoring cycles.

We also analyze the effectiveness achieved with each set of execution traces obtained using Tigris with the different filters and sampling rates. Although improving the overall performance is important, our key goal is to record the traces that are actually those needed for the analysis made by APLCache. Consequently, we consider as ground-truth the set of caching opportunities identified with full monitoring and compare it to the sets obtained with the various Tigris configurations. We measure this using typical classification performance metrics, namely precision, recall and f-measure. Results are presented in Table 4.7 (columns Pr., Recall and F1, respectively).

As can be seen, the precision—for both filters and sampling rates—is 1.00, that is, there are no false positives. This means that the subset of methods to be monitored in the second phase allows APLCache to identify caching opportunities correctly. These results, however, do not hold for recall. The recall decreases when the filter is more restricted and the sampling rate is lower. Filters can cause relevant methods (those that should be identified as a cacheable opportunity) to be not monitored in the second phase. Consequently, they are not traced and considered for caching. The low sampling rate, in turn, can lead to samples that are not representative of the population of the method calls. The variation of recall across the different Tigris configurations can be seen in Figure 4.5. It shows that the filter has a larger effect on recall than the sampling rate.

The restricted filter, for CloudStore and PetClinic, is the main cause for a low recall. It can be seen that even with sampling rates higher than 5%, the recall remains quite similar or even the same. For Shopizer, although the recall with 100% is not high (0.66), it decreases as the sampling rate decreases. Both the filter and the sampling rate, therefore, affect the recall. Although the expanded filter leads to false negatives, it achieves high recall, up to 1.00 (i.e. all the cacheable opportunities were found). Increasing the sampling rate improves the recall for all applications, as more information is provided for APLCache to analyze. For CloudStore and PetClinic, the highest recall can be achieved even with sampling rates lower than 100%. Shopizer potentially has results different

from the other applications because it is larger and thus has a broader range of methods. In addition, the margin of error of the recall, observed in the different monitoring cycles, demonstrated to be high for the expanded filter. It demonstrates how the workload variations can impact the number of methods selected for fine-grained monitoring and the number of cacheable opportunities found at each monitoring cycle by APLCache.

RQ2: Findings. The relevance filter and sampling rate of Tigris can reduce the number of identified caching opportunities and, consequently, the number of hits. The hit ratio, however, remains almost the same (0.91–0.98). Therefore, lower performance gains are obtained due to less cached opportunities. Tigris leads to no false positives considering APLCache and the target applications, thus achieving a precision of 1.00. However, recall can decrease due to the used relevance filter and low sampling rates, being larger the effect of the filter.

4.5.2.3 RQ3. How does Tigris cope with workload variations over time?

To evaluate how changes in the workload impact the methods selected by the first phase of the proposed approach, we inspected the subset of methods selected by the first phase of Tigris at the end of each monitoring cycle. Obtained results are presented in Table 4.8. We first observe that the number of selected methods for fine-grained monitoring changed on every cycle for all the applications. These changes were expected since the relevance criteria are domain-neutral, and the metrics used to analyze them do not rely on pre-defined thresholds or assumptions regarding the workload. As a consequence, the selected methods are based on the application's emerging behavior. In addition, the results show that bigger applications (i.e. with more methods and navigation paths) such as CloudStore and Shopizer, tend to have significant differences among monitoring cycles. As Petclinic has fewer navigation paths to be executed, the workload variations do not affect much the relevance-based selection. Still, all Petclinic monitoring cycles resulted in changes in the selection. This demonstrates the ability of our approach to adapt to workload variations. For all the applications, the restricted filter results in less relevant methods being selected for detailed monitoring than the expanded filter.

Table 4.8 – Simulation Results. Results in terms of changes in the relevance evaluation and selected methods to monitor for each application with restricted and expanded filters. After each monitoring cycle a snapshot of the coarse-grained selection is taken, reporting the amount of selected methods in that cycle (Selected), the overall difference from the last cycle (Difference), including the specific amount of additions and exclusions.

	Monitoring	Cycle	Selected	Difference
Cloud Store	Restricted Filter	1	8	
		2	14	+6 (+8/-2)
		3	13	-1 (0/-1)
	Expanded Filter	1	36	
		2	24	-12 (+0/-12)
		3	20	-4 (+1/-5)
Petclinic	Restricted Filter	1	4	
		2	5	+1 (+2/-1)
		3	4	-1 (+1/-2)
	Expanded Filter	1	14	
		2	13	-1 (+0/-1)
		3	12	-1 (+0/-1)
Shopizer	Restricted Filter	1	33	
		2	45	+12 (+13/-1)
		3	38	-7 (+2/-9)
	Expanded Filter	1	76	
		2	55	-21 (+6/-27)
		3	62	+7 (+11/-4)

RQ3: Findings. Because the proposed relevance criteria and metrics are domain-neutral and do not rely on pre-defined thresholds or assumptions, Tigris can adapt to different workloads, identifying relevant methods according to the application's emerging behavior.

4.5.2.4 Threats to Validity

We now analyze the threats to the validity of our empirical evaluation. First, the performance impact of monitoring highly depends on workloads. Although we do not make any assumptions regarding the workload and rely on the randomness added to the tests, the workload used in the experiments may not be representative enough to be generalized. Nevertheless, our approach does not depend on a particular workload and can find relevant traces with any pre-specified workload, which may evolve in real-world scenarios. Therefore, even if the workload changes substantially and initial relevant methods

are no longer useful, our approach can adapt itself, automatically discarding outdated monitoring configurations and discovering a new set of relevant execution traces.

Second, our evaluation is limited to one goal of monitoring (i.e. application efficiency in terms of performance) and only one monitoring-based approach (i.e. a caching technique). Therefore, the results may not be generalizable. To address this threat, we selected three open-source target applications, with different sizes and domains, implemented by different developers. In addition, we provide a wide variety of the tunable parameters for adaptive monitoring (i.e. the relevance filter and sampling rate) and compare the results against a baseline. We acknowledge that all of the threats mentioned above may require several evaluations concerning multiple systems of different sizes, users, traces, workloads, and other environmental conditions that should be addressed as part of future work.

4.6 Limitations

Providing a solution for effective execution tracing requires dealing with many challenges other than those addressed in this thesis, such as overhead management, sampling gaps, and defining appropriate criteria, metrics and sampling rate to achieve the goal of monitoring. Consequently, although our approach makes substantial advances towards software monitoring, there are challenges that remain open. These correspond to limitations of our work, which are discussed as follows.

The coarse-grained monitoring demands to instrument all method calls, and thus a minimal but additional overhead is incurred due to this per-instruction instrumentation before the filtering and sampling decision making. To reduce even more the overhead of the approach, we can also apply a dynamic sampling strategy into the coarse-grained monitoring.

Although the framework supports the use of already implemented estimations of metrics, which were identified and conceived based on the investigation of existing monitoring-based approaches, our approach does not cover the challenge of deciding which estimate or criteria are appropriate to specific goals of monitoring. In fact, tunable parameters for adaptive monitoring (in terms of relevance criteria and sampling rate) create a configuration space and as a result expose a secondary problem of finding appropriate values for such configuration options. Thus it is not in the scope of this thesis. However, the results of our foundational study includes a list of relevance criteria, an

occurrence-based association between groups of goals and relevance criteria, and examples of goals and metrics used by the surveyed approaches. This information provides support to specify or at least reduce the configuration space created by our approach. In addition, although the available criteria, goals, and metric estimations achieved good results in our experiment, they may not fit well in all the domains and workloads. To solve this problem, Tigris was designed to be extensible and flexible, providing interfaces that can be used to customize and change how metrics are calculated.

Regarding the classification of data into groups, we do not deal with possible outliers that may appear due to the transient behavior of the application, such as an increase in the execution time of a method given the high level of concurrency. It can be addressed in the future with enhanced statistical analysis and filtering of outliers. In addition, some tests for normality may be not sensitive enough given the sample size and the property of the data. Ideally, testing for normality should be executed and interpreted alongside histograms, QQ-plots, and skewness and kurtosis values. To solve this problem, the framework can be evolved in such a way that small parts of the monitoring phases could be customized by users, such as how to classify data.

The coarse-grained monitoring data is stored in memory, and despite its low memory usage, it may reach an imposed limit if kept forever. In our evaluation, this was not an issue. However, this can be configured in the form of a time frame and added as an additional parameter of the framework. We also need to understand how we could combine sliding windows of monitoring to avoid losing historical information.

Although the framework is adaptive as it can vary the selected list of relevant methods according to the application's emerging behavior, it is necessary to specify when this adaptation should be triggered. In our evaluation, it uses a two-minute interval. However, choosing the most appropriate interval to update the list of relevant methods involves a trade-off between collecting enough lightweight information about the application behavior to reach good decisions and changing the list fast enough to keep it in sync with the application behavior and collect more relevant traces. To solve this problem, future work can provide an adaptive triggering strategy, which detects significant variations in the workload characteristics or degradation in the quality of traces being collected, according to the goal of monitoring.

4.7 Final Remarks

This chapter presented a framework, called Tigris, to perform monitoring based on user-specified relevance criteria. The framework is guided by relevance criteria and metrics that are specified by users using our proposed TigrisDSL. The first monitoring phase is *coarse grained*, computing low-overhead metrics of executed methods. The second phase is *fine grained*, after dynamically selecting which methods should be traced in detail, given a relevance criteria specification and a sampling strategy to reduce the overhead. By splitting the process into different phases (or cycles), our proposed framework achieves a flexible and extensible architecture, where each phase can be supplemented or replaced by alternative approaches.

Although the proposed filtering strategy is based on the goal of monitoring, given as input of the approach, the sampling strategy used in the second phase of the approach follows a fixed sampling rate, leading to uncertainty in the monitoring result in terms of representativeness. Choosing a sampling rate is not an easy task and depends on the representativeness needed and also the supported overhead. A low sampling rate may give a lousy precision, and a high sampling rate may generate a considerable amount of useless data and overhead. To deal with this situation, we can adopt an adaptive sampling, which dynamically adjusts the sampling rate by observing the impact of sampling rates on the overall computational resource usage. Our proposal for adaptive sampling is presented in the next chapter.

5 ADAPTIVE SAMPLING

Given the limitations of existing work and our goal of collecting representative samples of detailed execution traces at runtime with controlled performance impact, we propose a three-activity process for monitoring software applications with an adaptive sampling rate. Our decisions are at the granularity of application request, which have method calls (executed to respond it) recorded as detailed execution traces. This occurs if the request is selected to be part of a sample collected in a monitoring cycle. We use the *PetClinic*[1] project as a running example to explain the activities of our process. It is a web application that demonstrates the use of the Spring Framework and its features. It provides features (possible requests) in which employees of a pet clinic can view and manage information regarding veterinarians (`/vets`), clients (`/owners`), and their pets (`/pets`). It also includes a home page (`/home`), which is the entry point for users.

We next first overview our process and its activities, describing each activity in detail (Section 5.1). Then, we present an evaluation of the proposed solution (Section 5.2) and its limitations (Section 5.3).

5.1 Process Overview

The key idea underlying our process is to decrease the monitoring overhead to an acceptable level when the target software application needs its resources for regular processing and increasing it after the situation has been normalized. At the same time, we keep track of general statistics about the sample and population of requests to identify when a sample is representative. Our monitoring process is performed in *cycles* and the result of each cycle is a representative sample. This behavior is shown in Figure 5.1, where the black line represents the application workload, the red line represents the monitoring overhead, and the green line indicates the amount of requests and their execution traces being collected over time.

In order to make this behavior possible, three activities are performed in parallel at runtime as part of our process, as shown in Figure 5.2. The first activity, *Sampling Decision*, is responsible for deciding whether an application request should have its associated execution traces collected and stored in detail (which is costly and requires I/O), taking into account both the sampling rate and the representativeness of the sample compared

[1] <https://projects.spring.io/spring-petclinic/>

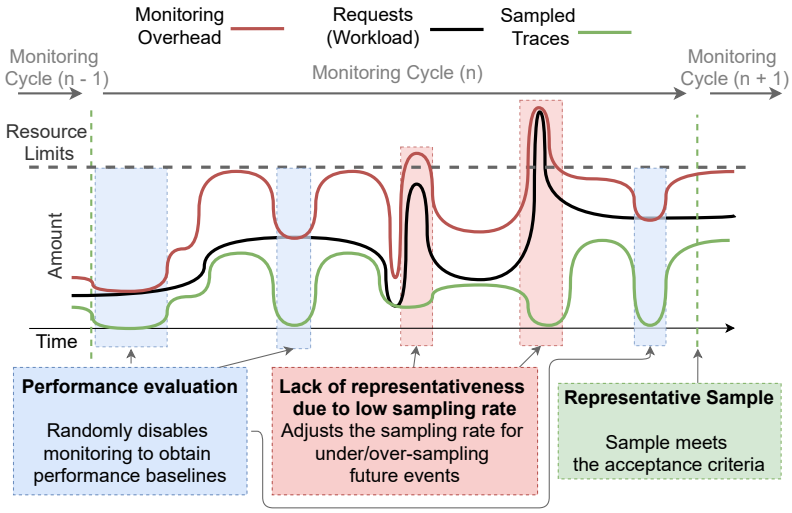


Figure 5.1 – Illustration of the Adaptive Sampling Process in Action: The figure shows how the sampling rate varies (in terms of the number of sampled traces) according to the current application workload. In peaks, the sampling rate is reduced to reduce the monitoring overhead.

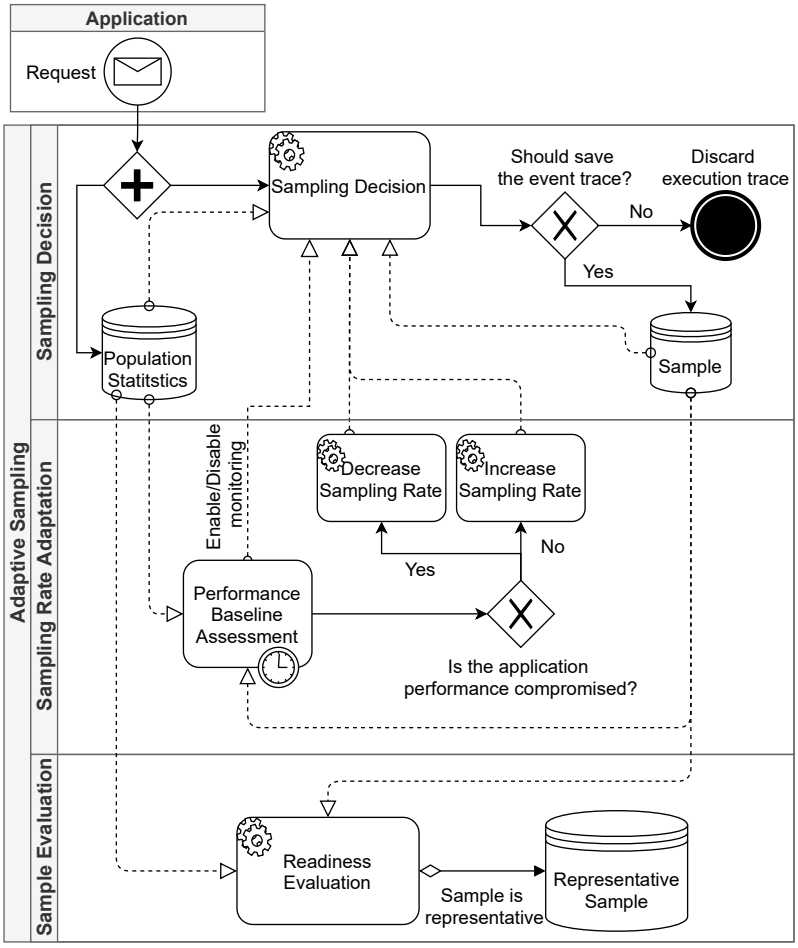


Figure 5.2 – Overview of our Adaptive Sampling Process.

to the population. The sample representativeness is based on the distribution of application requests. All requests go through screening and by doing so, we also keep general statistics of the population of requests.

The sampling rate used in the Sampling Decision activity is updated by the *Sampling Rate Adaptation* activity. It observes the current application workload as well as the monitoring overhead to decrease or increase the sampling rate. These adjustments of the sampling rate can be seen in Figure 5.1—the larger the space between the black and red lines, the higher the sampling rate and, consequently, the monitoring overhead. When a performance degradation is perceived, i.e. the application response time increases, the sampling rate is decreased proportionally to the perceived degradation. To exemplify, suppose that an unexpected increase in the application workload results in the application bumping in the limits of its runtime platform (shadowed red boxes). In this situation, the monitoring is gradually reduced to allow the application to maintain its throughput. The sampling rate can be restored when the application workload decreases. To be able to assess the impact caused by monitoring, this activity also involves the collection of a performance baseline, which is a measurement of the application performance without monitoring. In Figure 5.1, this occurs in the shadowed blue boxes.

Finally, the *Sample Evaluation* activity is responsible for continuously evaluating the sample being collected to identify when it is considered representative. When the sample satisfies acceptance criteria with respect to the representativeness of the population, it is released for analysis, and a new monitoring cycle starts. If our monitoring process is used, e.g., in a self-adaptive system, the analysis of the sample can take place right after each cycle is concluded. After broadly understanding how our process works, we next describe each of its activities.

5.1.1 Activity 1: Sampling Decision

The *Sampling Decision* activity involves the execution of Algorithm 1 whenever a new observable request happens in the application. It performs three main tasks: (i) store statistics of the population of application requests (line 1); (ii) decide whether a request should be recorded with execution traces (lines 2–13) and (iii) store statistics of the request being added to the current sample (line 8), when applicable.

Storing statistics associated with the population and sample means keeping track of the *frequency distribution* of each request in these two sets. This information is used to decide whether a particular request should be recorded with execution traces or whether a sample is representative and the monitoring cycle is complete. By adding requests to the population and sample (lines 1 and 8), we keep registered the number of each

Algorithm 1: Sampling Decision

Input: $request$ to be processed by the application;
Input: current sampling rate $rate \in (0, 1)$;
Input: monitoring control $isMonitoringEnabled$;
Data: $population$, $sample$;
Result: $True$ if the execution traces of $request$ should be recorded, $false$ otherwise.

```
1  add(request, population);
2  if isMonitoringEnabled then
3      shouldSample ← Bernoulli(rate);
4      if shouldSample then
5          E_p ← { x ∈ population | x = request.id };
6          E_s ← { y ∈ sample | y = request.id };
7          if ( |E_p| / |population| ≥ |E_s| / |sample|, ε ) then
8              add(request, sample);
9              return true;
10         end
11     end
12 end
13 return false;
```

Table 5.1 – Running example: Frequency distribution of the population and sample in the sampling decision activity.

Request	***population***		***sample***	
`/home`	105	(47.7%)	53	(47.7%)
`/vets`	43	(19.5%)	22	(19.8%)
`/pets`	62	(28.3%)	31	(27.9%)
`/owners`	10	(4.5%)	5	(4.5%)
Total	**220**	**(100%)**	**111**	**(55.5%)**

possible request, as shown in Table 5.1 considering the PetClinic example. These values are obtained with an initial sampling rate of 50%.

A request is added to the sample and recorded with execution traces when three conditions are satisfied. The first refers to whether the monitoring is enabled (line 2). As introduced in the previous section, there are moments when the monitoring is disabled to obtain a performance baseline—this is controlled by the Sampling Rate Adaptation activity. The second condition involves randomly deciding whether the request should be selected based on the current sampling rate (*rate*), which gives the probability of selecting a request as part of the sample. This decision is made using the Bernoulli distribution with parameter $p \in (0, 1)$ to assign the value 1 (*true*) with probability *p* and the value 0 (*false*)

with probability $1 - p$ to the *shouldSample* variable (line 3). The request satisfies the second criteria when *shouldSample* is *true*.

The third condition is related to the representativeness of the sample— we aim to keep its frequency distribution similar to that of the population (lines 5–7). The rationale is to not miss less frequent requests with, e.g., anomalies and exceptions. The idea is related to stratified sampling, where a population can be partitioned into subpopulations, i.e. clusters (PIRZADEH et al., 2011), and a representative sample has the same class distribution as the population. The verification that is performed consists of a runtime *resampling strategy* to balance the sample's class distribution according to the population's class distribution. This strategy is inspired by data analysis resampling to deal with unbalanced datasets (ESTABROOKS; JO; JAPKOWICZ, 2004). By resampling, we balance the sample classes according to the population distribution to keep representativeness in terms of proportion. It consists of ignoring new requests from the majority classes to allow minority classes to include more requests and increase their cardinality. As previously said, we keep the statistics of the population (*population*) and sample (*sample*) (exemplified in Table 5.1). These are kept as key-value mappings, where the type of a request is the key (*request.id*), and its execution frequency within a monitoring cycle is the value. Using these statistics, it is possible to compute E_p and E_s, which are the amount of requests of a particular type in the *population* (line 5) and in the current *sample* (line 6), respectively. Based on these values, we test if the sample is lacking requests of the type in consideration, considering an error margin ϵ. For example, if a request of type `/vets` occurred (Table 5.1), it will be not added to the sample because it already has enough traces from `/vets` (19.8%) when compared to the population (19.5%). When a request satisfies this condition (line 7), the request is added to the sample (line 8) and the algorithm returns *true* (line 9), that is, the request should be recorded with execution traces. If any of the three conditions is not satisfied, the algorithm returns *false* and the request is not recorded, implying no additional monitoring overhead.

5.1.2 Activity 2: Sampling Rate Adaptation

The sampling rate used in the previously described activity is updated by Algorithm 2 executed in the Sampling Rate Adaptation activity. This is done considering the following premisses.

Algorithm 2: Sampling Rate Adaptation

Input: current $sample$ being collected in the monitoring cycle;
Input: the maximum time window in which a performance baseline must be kept $duration$;
Input: the current performance $currentPerf$;
Input: monitoring control $isMonitoringEnabled$;
Data: the current $samplingRate$; $performanceReference$;
Result: The updated sampling rate.

addPerformanceSample($performanceReference$, $currentPerf$, $isMonitoringEnabled$);
$medianRps \leftarrow \text{median}(\{x \in performanceReference[RpS] \mid performanceReference[ME = isMonitoringEnabled]\})$;
$normalBehavior \leftarrow performanceReference[RpS = medianRpS \wedge ME = isMonitoringEnabled]$;
$equal \leftarrow \text{ttest}(normalBehavior, currentPerf, 0.05)$;
$diff \leftarrow \frac{\sum_{i \in currentPerf} i}{\sum_{i \in normalBehavior} i} - 1$;
if $isMonitoringEnabled$ **then**
    if *equal or diff > 0* **then**
        $rate \leftarrow \min(rate + (rate * |diff|), maxRate)$;
    else
        enablePerformanceBaseline($duration$);
    end
else
    if *not equal and diff < 0* **then**
        $rate \leftarrow \max(rate - (rate * |diff|), minRate)$;
    end
end
return $rate$;

1. A software engineer is able to provide a desired sampling rate that leads to an *acceptable performance impact* caused by monitoring when the application *workload is typical*.

2. The acceptable performance impact is not in terms of percentage but the absolute increase in the response time. For example, if the response time of a request is typically 100ms and with monitoring 105ms, the acceptable performance impact is 5ms and not 5% of overhead.

3. The sampling rate should be reduced to prevent an increase in the response time when the application is under stress, limited by a minimum required sampling rate.

4. The sampling rate should be increased if it is below the desired level and the software application is returning to its typical behavior after a peak.

Table 5.2 – Running example: *performanceReference* table containing the response time of each request according to a given workload in requests per second (RpS). The ME column indicates whether the record was collected when monitoring was enabled. Rows highlighted in gray refer to the median of each group (ME = true and ME = false).

#	RpS	/home	/vets	/pets	/owners	ME
1	500	325ms	450ms	800ms	1200ms	true
2	1500	400ms	550ms	900ms	1500ms	true
3	2500	600ms	780ms	1050ms	1100ms	false
4	550	350ms	400ms	650ms	900ms	true
...	...	...	...	...	...	...
$n-1$	325	430ms	420ms	480ms	700ms	false
n	200	270ms	200ms	235ms	500ms	true

Based on these premisses, the adaptation of the sampling rate requires three inputs: (i) *maxRate*, which is the desired sampling rate and a higher sampling rate is not needed; (ii) *minRate*, which is the minimum required sampling rate; (iii) the *duration* of the period in which the approach should collect data to understand the application performance with the monitoring disabled (this is required for assessing the monitoring performance impact); and (iv) the *frequency* in which the sampling rate is revised (in seconds).

Algorithm 2 performs the following tasks. First, it stores statistics of the application performance (line 1). The parameter *currentPerf* gives the current application performance as a record with the number of requests executed since the last algorithm execution (RpS) and the average response time of each executed request. These data are stored in the *performanceReference* table, which consists of a performance sample. For each record, we also store a flag indicating if these data correspond to a period in which monitoring is enabled. An example of *performanceReference* is shown in Table 5.2 for PetClinic. To avoid bias towards past observations and consuming unnecessary resources, *performanceReference* is size-limited and stores the most recent executions, i.e. the oldest record is discarded to store a new one when the size limit is reached.

Based on these statistics, it is possible to derive both the typical application workload *medianRpS* (line 2) and the corresponding normal behavior *normalBehavior* in terms of response time (line 3), with and without monitoring. The typical application workload is given by the median of requests per second *medianRpS* of the records in *performanceReference*. Considering our example, these are the records #2 and #3, with and without monitoring, respectively. Because we need to select a single record to be used in the next algorithm tasks, if *performanceReference* contains an even number of records, we select the highest requests per second (as opposed to the arithmetic mean of the two

middle values) and the associated record, which is the *normalBehavior*.

Then, we test if the current application performance *currentPerf* is significantly different from the normal behavior *normalBehavior* of the application to detect performance variations (line 4). This is done by comparing the averages of the execution times of each request type with a paired t-test, with the null hypothesis that the mean of the paired differences between *currentPerf* and *normalBehavior* is 0, with a significance level of 95% ($p = 0.05$). Assume that Algorithm 2 is executing in our example with monitoring disabled and thus the *normalBehavior* is record #3. Let *currentPerf* be

$$\{\langle /home, 500\rangle, \langle /vets, 720\rangle, \langle /pets, 950\rangle, \langle /owners, 1020\rangle\}. \tag{5.1}$$

The result of the comparison of *normalBehavior* and *currentPerf* is assigned to the variable *equal*, which is the result of

$$ttest(\langle 600, 780, 1050, 1100\rangle, \langle 500, 720, 950, 1020\rangle, 0.05). \tag{5.2}$$

This results in $equal = true$, indicating that there is no significant difference between these two groups. This is the first indicator of whether the sampling rate should be updated. The second indicator is the difference *diff* (in percentage) between *currentPerf* and *normalBehavior* (line 5). In the example, it is

$$diff = (3190/3530) - 1 = 0.9037 - 1 = -0.0963 \tag{5.3}$$

This means that the current performance is 9.63% lower than the normal behavior.

These two indicators (*equal* and *diff*) are used together with the monitoring state of the application to decide whether the sampling rate should be updated. The sampling rate is increased—proportionally to the observed difference, limited by *maxRate*—if the current performance (which includes monitoring) is similar or better than the normal behavior (lines 7–8). This means that the current execution times of the requests are similar or faster than the past observations, and thus the monitoring overhead is acceptable. The sampling rate is decreased (also based on *diff*), if the normal behavior is significantly different from the current performance (which does not include monitoring, limited by *minRate*) and the current performance is worse than the normal behavior (lines 13–15). This case occurs when the current execution times of the requests are slower when compared to past observations. In this situation, the application is (a) being impacted by users

with an increased workload, or (b) the monitoring overhead is impacting the performance above acceptable levels. Note that $minRate > 0$ because if the sampling rate reaches 0, it remains 0 indefinitely.

In order to identify whether the application performance still faces degradation regardless of the case, there is a need for collecting a performance baseline (line 10). This occurs when the application is being monitored and the current performance is worse (significantly different and lower) than the normal behavior. In this case, the monitoring is then globally disabled and the performance baseline is collected according to the specified *duration*. Finally, if none of these conditions are met, the sampling rate remains the same.

5.1.3 Activity 3: Sample Evaluation

Our process aims at collecting a representative sample in each monitoring cycle. Therefore, when a new request is added to the sample, the sample representativeness is evaluated to determine if it is ready for being used. This is done in the Sample Evaluation activity that involves the execution of Algorithm 3. A sample is considered representative if it satisfies three criteria: (i) it is larger than the minimum sample size; (ii) the performance between the sample and population is equivalent; and (iii) the sample distribution is similar to the population distribution. When the sample satisfies these criteria, there is statistical evidence that it is representative.

To prevent a scenario in which the sample being collected is never representative, Algorithm 3 uses an exponential *decaying confidence* (line 1). It is used to adjust the required level of representativeness of the sample based on the length of the monitoring cycle. The confidence (z) starts at 100% and decreases by a constant λ every second (t) in the monitoring cycle, where $\lambda = 1/maxLength$. *maxLength* is the maximum length of the monitoring cycle, which is the required parameter of this activity.

The first criterion—sample size—is checked in line 4. The required sample size (line 3) is given by Cochran's minimum sample size (COCHRAN, 1977) (line 2), with finite population correction, where the decaying confidence level z is used to estimate the associated standard normal distribution (e.g., 1.96 for $z = 0.95$), e is the margin of error ($e = 0.05$) and p is the degree of variability of the population, indicating how heterogeneous the population is. We use $p = 0.5$ because it does not assume that the population is homogeneous and leads to higher samples, being thus a conservative choice.

Given that we track the performance of application requests, we test the equiva-

Algorithm 3: Sample Evaluation

Input: the length of the current monitoring cycle t in seconds;
Data: the current $population$ and $sample$;
Result: $true$ is the $sample$ is representative of the $population$.

1 $z \leftarrow 100 * e^{-\lambda t}$;
2 $n_\infty \leftarrow \dfrac{z^2 p(1-p)}{e^2}$;
3 $n \leftarrow \dfrac{n_\infty}{1 + \dfrac{n_\infty - 1}{|population|}}$;
4 **if** $|sample| > n$ **then**
5 **if** *ttest(population, sample, z)* **then**
6 balanced $\leftarrow$ false;
7 **foreach** *request in population* **do**
8 $E_p \leftarrow \{\, x \in population \mid x.id = request.id \,\}$;
9 $E_s \leftarrow \{\, y \in sample \mid y.id = request.id \,\}$;
10 **if** $\left(\dfrac{|E_p|}{|population|} = \dfrac{|E_s|}{|sample|}, z \right)$ **then**
11 $balanced \leftarrow$ true;
12 **end**
13 **end**
14 **if** *balanced* **then**
15 release $sample$ for analysis;
16 $sample \leftarrow \emptyset$;
17 $population \leftarrow \emptyset$;
18 return $true$;
19 **end**
20 **end**
21 **end**
22 return $false$;

lence between the sample and population—stored in Table 5.2—using this measurement, which is the second criterion. This is done by verifying if the frequency distribution of the sample significantly differs from the postulated population mean using a one-sample parametric t-test (two-sided) (line 5) to test the null hypothesis that the sample mean is equal to the population mean, with a decaying confidence level (computed in line 1). The test compares the average values of the two data sets and determines if they came from the same population.

Finally, the third criterion is evaluated in lines 6–14, which checks the balance in the request distribution, measuring the over- or under-representation of requests in the sample compared to the population. This is similar to the comparison shown in Table 5.1. However, in line 10, the comparison between each request type in the sample and in the

population considers a margin of error of z (decaying confidence level). This evaluation aims to maximize the sample distribution in such a way that the sample is balanced according to the population to give confidence that all requests, even those that rarely happen, are present.

When the sample meets all these three criteria, it is said representative and is released for analysis. After releasing the sample for analysis, we reset the sample and population (lines 16 and 17), and a new monitoring cycle starts.

5.2 Evaluation

Having described our monitoring process in detail, we now evaluate it using applications of a widely known and used benchmark.

5.2.1 Evaluation Settings

5.2.1.1 Research Questions and Metrics

Our monitoring process aims at collecting representative samples of execution traces and, at the same time, keeping the performance overhead at an acceptable level. Therefore, the goal of evaluation is to assess these two aspects, which are the focus of our two research questions, listed as follows. Metrics used to answer each question are also detailed.

1. **RQ1** What is the *performance impact* of our monitoring process?

   TR Throughput (average number of requests / second)

   SR Average sampling rate / second

2. **RQ2** What is the *representativeness* of the samples of executions traces collected with our monitoring process?

   RMSE Root-mean-square error of memory consumption

Whenever we monitor a software application, there is performance overhead. Therefore, in RQ1, we quantitatively assess this overhead by measuring the application throughput (TR). In addition, given that the monitoring overhead is proportional to the

sampling rate, to better understand the throughput, we also measure the sampling rate (SR). A dynamic sampling rate may have a negative effect on the representativeness of collected traces, making them less useful for understanding the application behavior for a particular goal, such as debugging. For not biasing our evaluation towards our approach, we do not use the criteria on which our process relies to assess sample representativeness. Instead, we use a software characteristic—memory consumed by methods—that can be understood through monitoring, is straightforward to be collected, and can have its correctness evaluated. Based on a ground truth (explained as follows), we can assess the error (RMSE) of the sample with respect to the average memory usage of each method. If RMSE is low, it indicates that the collected traces are reliable for debugging memory consumption, for example.

5.2.1.2 Compared Approaches

Our adaptive monitoring process (`ADP`) is compared to the main alternative approach (HAUSWIRTH; CHILIMBI, 2004; BRÖNINK; ROSENBLUM, 2016), which is a sampling rate inversely proportional to the workload given by the throughput (`INV`), and a practically used approach, which is uniform sampling (`UNI`). The selected uniform sampling rate is 50%, which is the same used as initial (and desired) sampling rate for `ADP` and `INV`. As a reference, we also execute our evaluation with two additional configurations. The first is no monitoring (`NOM`), in which there is no monitoring overhead and it thus provides the maximum (best) possible value for throughput. The second is full monitoring (`FUM`), in which every application request is recorded with execution traces. `FUM` gives thus the minimum (worst) possible throughput value and also serves as ground truth for calculating RMSE because it contains data from the population and not a sample.

5.2.1.3 Target Applications

We use the DaCapo (BLACKBURN et al., 2006a; BLACKBURN et al., 2006b) benchmark suite for our evaluation. DaCapo provides various Java applications to evaluate approaches that focus on the execution environment of applications. Consequently, it does not explicitly provide extension points for customizing its execution. Because we need to simulate a workload with variation (not simply firing requests one after another) as well as instrument the applications, we selected a subset of five applications to be instrumented, described as follows.

cassandra Executes queries to recover documents from the NoSQL database management system Cassandra.

h2 Executes SQL transactions against a model of a banking application on top of the H2 database.

lusearch Executes search queries against the document search engine Lucene.

tradebeans Runs HTTP requests via Java Beans against a web application that simulates a stock trading system.

xalan Calls multiple times an XSLT processor for transforming XML documents into HTML pages.

The rationale for selecting these particular applications is that they are all distributed applications designed to process multiple requests in parallel, e.g. web requests or database queries, and are based on domains in which monitoring is valuable and difficult to control in terms of overhead.

5.2.1.4 Procedure

We instrumented all the target applications (which are open source) to enable monitoring using aspect orientation. Aspects intercept method calls, and then there are four Java implementations (ADP, INV, UNI, and FUM) to decide whether an application request should be recorded and, if so, collect execution traces. NOM corresponds to the original version of the applications.

As introduced, our monitoring process requires a set of parameters. They consist of a single configuration for all target applications and not values that must be tuned for specific applications. We used a sample application, not used in our evaluation, to empirically choose these parameters. The Sampling Rate Adaptation activity is triggered every 1s and the collection of performance baselines lasts 3s. The maximum and minimum sampling rates are 50% (as in UNI) and 1%, respectively. Finally, the maximum length of a monitoring cycle is 180s (3 min).

For generating application workloads, we use the workload simulation provided by DaCapo, which relies on a navigation pattern that either falls into a specific distribution (transition table) or follows a specific sequence of executions. We use this navigation pattern to simulate a varying number of simultaneous users (i.e. threads). This

allows us to observe the impact of monitoring when the application is under various stress conditions and how the sampling rate is adjusted. Inspired by load intensity modeling approaches (Von Kistowski et al., 2017), the designed workload includes (a) situations in which it keeps a stationary number of users, (b) seasonal patterns, and (c) bursts in the number of simultaneous users. The same workload settings are used to execute each application with each compared approach.

The simulations were executed on an Intel i7 2GHz with 16G RAM. The maximum heap size of the Java Virtual Machine (JVM) was limited to 4GB to cause the applications to execute under stress (with limited resources considering the workload). Each simulation was executed 10 times. The maximum number of simultaneous users was selected based on the identification of which number of users causes the application to deteriorate its performance due to the lack of resources. With 4GB of RAM available, this number varies from 6 to 200. The memory consumed by methods is not explicitly made available by the JVM. We use a standard way of performing this, i.e. checking the available memory before and after the method execution. Some measurements, however, must be discarded because they are invalid—negative values due to the execution of the JVM garbage collector to free up memory.

5.2.2 Results

The results obtained following the procedure described above are presented in Table 5.3. It shows the values obtained for each metric (TR, SR, and RMSE) with each compared approach (ADP, INV, UNI) and reference values (NOM and FUM) for each target application. Because we run the simulation 10 times for each configuration, we present the mean and standard deviation. As can be seen, the results are consistent across all applications, even though they vary in nature. A Friedman's test showed that there is significant difference among the compared approaches, both for TR ($\chi^2(2) = 84.28$, $p < 0.001$) and RMSE ($\chi^2(2) = 74.68, p < 0.001$). Post-hoc analysis with pairwise comparisons using Nemenyi-Wilcoxon-Wilcox all-pairs test for a two-way balanced complete block design revealed that this is due to the differences among all approaches in both cases.

With respect to performance overhead (RQ1), as expected, UNI achieves the worst results, causing the throughput to be 13.5–23.1% lower than NOM. The impact is approximately 50% of FUM, as it collects execution traces of roughly half of the requests. INV

Table 5.3 – Simulation Results: Comparison of the values obtained for the metrics Throughput (TR), Sampling Rate (SR), and Root-mean Square Error (RMSE).

	Mon.	TR	SR	RMSE
cassandra	NOM	23180.4±488.6	0%	—
	FUM	16769.5±468.7 (-27.6%)	100.0%	—
	ADP	20763.9±666.2 (-10.4%)	48.7%±1.3	496.6±42.9
	INV	21112.7±520.6 (-8.9%)	29.2%±2.5	699.9±57.0
	UNI	18900.2±661.7 (-18.4%)	50.0%	651.6±75.2
h2	NOM	1829.0±20.1	0%	—
	FUM	1197.1±30.2 (-34.5%)	100.0%	—
	ADP	1587.1±19.8 (-13.2%)	44.0%±3.4	628.0±108.6
	INV	1633.2±24.4 (-10.7%)	32.7%±1.5	1291.9±118.7
	UNI	1517.0±19.1 (-17.0%)	50.0%	1196.1±102.8
lusearch	NOM	74376.7±213.8	0%	—
	FUM	49397.7±363.9 (-33.5%)	100.0%	—
	ADP	66267.2±324.2 (-10.9%)	41.6%±2.2	1394.2±349.0
	INV	70951.0±216.0 (-4.6%)	28.0%±1.6	3010.8±450.0
	UNI	57142.0±433.3 (-23.1%)	50.0%	2086.0±251.1
tradebeans	NOM	1832.2±20.2	0%	—
	FUM	1204.6±17.9 (-34.2%)	100.0%	—
	ADP	1571.8±14.0 (-14.2%)	48.4%±0.9	721.2±40.3
	INV	1619.6±15.1 (-11.6%)	32.7%±1.1	797.7±55.5
	UNI	1512.8±22.6 (-17.4%)	50.0%	831.0±36.7
xalan	NOM	266.8±1.0	0%	—
	FUM	182.6±1.7 (-31.5%)	100.0%	—
	ADP	239.1±1.5 (-10.3%)	47.5%±0.9	111.7±9.1
	INV	241.1±1.7 (-9.6%)	24.0%±0.6	187.4±13.2
	UNI	230.5±1.3 (-13.5%)	50.0%	135.0±10.3

has the lowest performance overhead, with an overhead ranging from 4.6% to 11.6%. This occurs because it always reduces the sampling rate with more intense workloads, regardless of its impact on the collected execution traces. ADP, in turn, is the "middle option", which has an overhead from 10.3% to 14.2%, as it also takes sample representativeness into account while monitoring the application.

The performance overhead is in accordance with the sampling rate. The lower the performance overhead, the lower the sampling rate. Despite achieving the intermediate results, in all cases, ADP has a performance overhead closer to INV than to UNI. Nevertheless, its average sampling rate is, also in all cases, closer to UNI than to INV, sometimes as high as 48.7% (note that the sampling rate is always limited to 50%). This

indicates that ADP is able to choose the moments in which the sampling rate should be reduced (this is further discussed in the next section) as well as reduce the sampling rate in a sustainable manner.

With respect to the error present in collected samples (RQ2), ADP is not the middle option. In all cases, it has the best (lowest) results for RMSE, which is the average error of the sample (in memory kilobytes) with respect to the population. This provides evidence that ADP is able to collect execution traces that better represent the population. Although, as expected, INV has the highest error for most applications, this is not the case for `tradebeans`. A possible explanation is that the memory consumption of the different application requests largely varies for this application and, in this particular case, relying on randomization to collect traces, even with higher sampling rates, cannot guarantee good results.

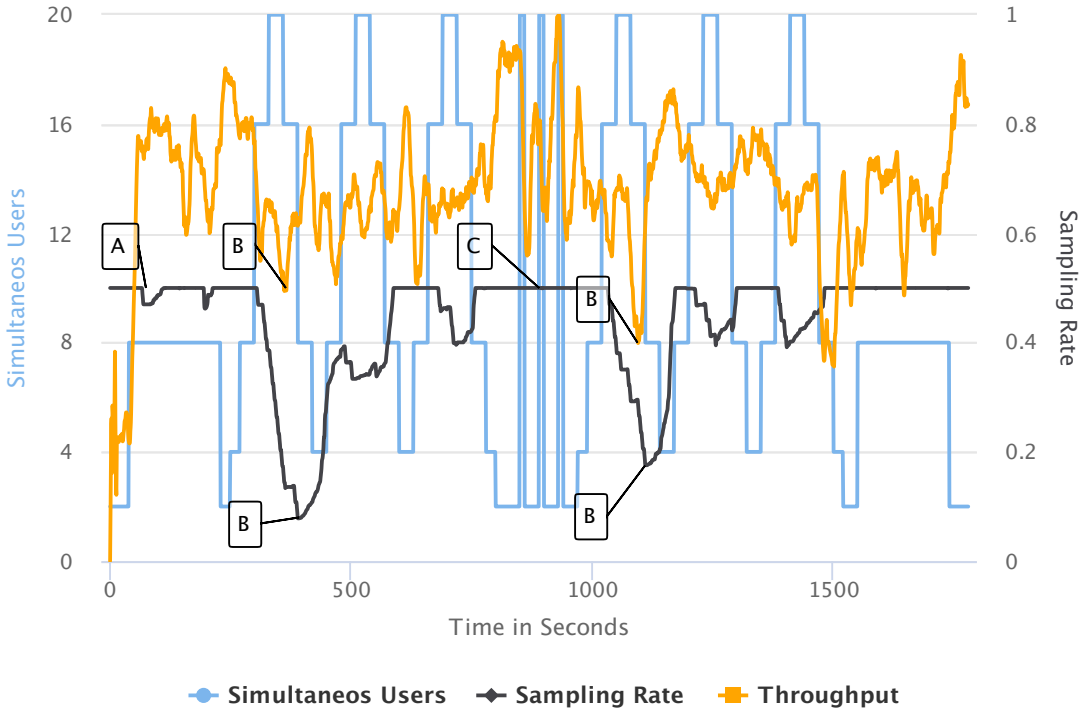
Conclusion. ADP is able to collect the most representative samples of execution traces, using memory consumption as representativeness measure. The error of the collected samples is 9–54% and 12–44% lower than INV and UNI, respectively. It also significantly reduces the performance overhead of UNI (3–12% lower). Although it has a performance overhead higher than INV, it is much lower (1–6%) than the reduction of the error in the collected samples.

5.2.2.1 Detailed Analysis

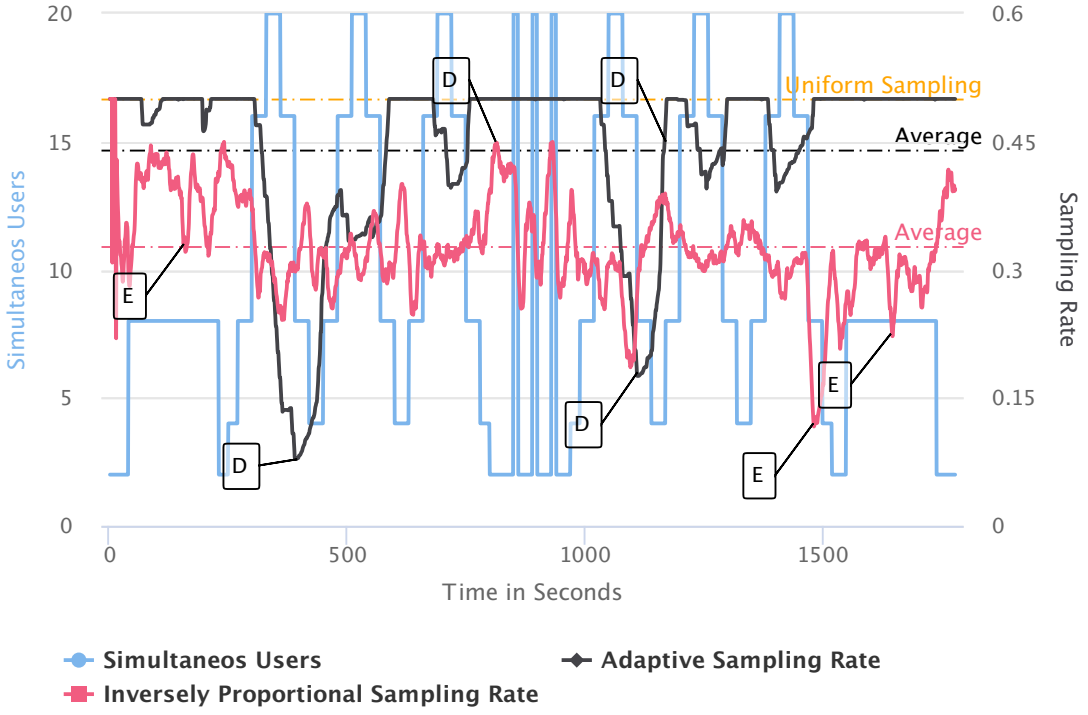
In the previous section, the results show that ADP significantly improves the representativeness of the collected samples of execution traces, with little impact on the performance overhead. To explain these results, we analyze in detail the results obtained with `h2`, shown in Figures 5.3 and 5.4[2]. From the 10 executions, we selected that with the median throughput value.

Interaction among workload, throughput, and sampling rate. We first analyze what happens over the course of the simulation in Figure 5.3a. The blue line shows the application workload. The typical workload is 8 simultaneous users, which can be 20 in peeks (recall that the memory limit is 4GB). As explained, the workload has stationary segments, seasonal patterns and bursts. In stationary segments, ADP is able to keep the

[2] All applications have similar results. Due to space restrictions, their charts are available in our complementary material at <https://tinyurl.com/adaptive-sampling-thesis>.



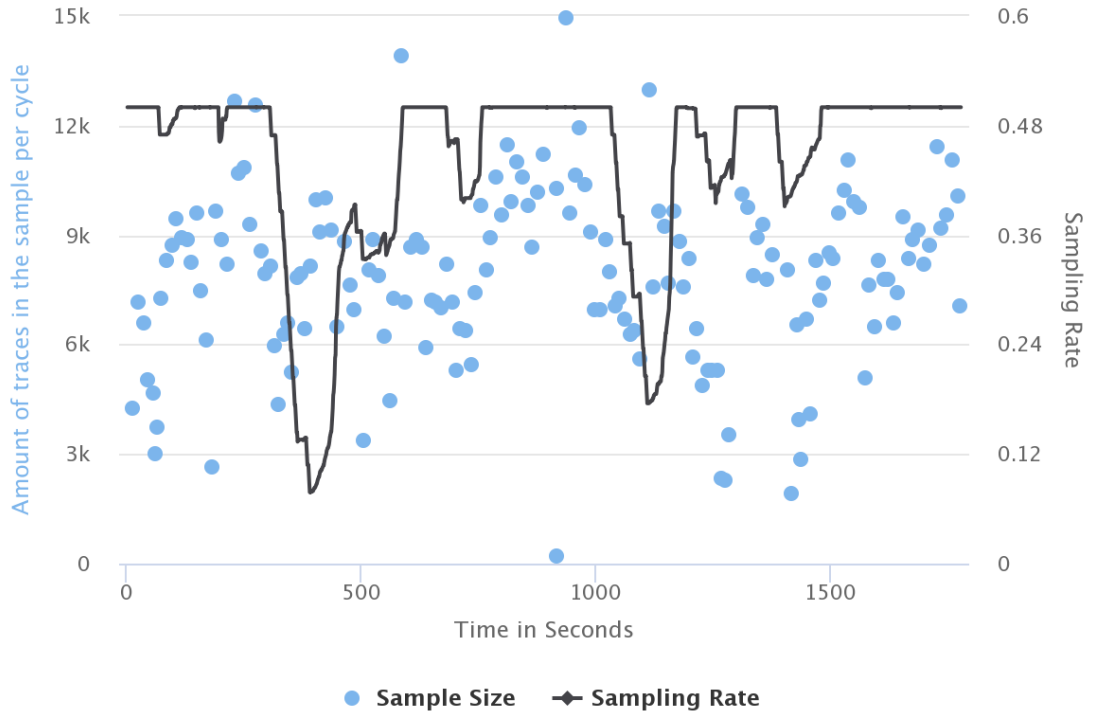
(a) Interaction among workload, throughput, and sampling rate.



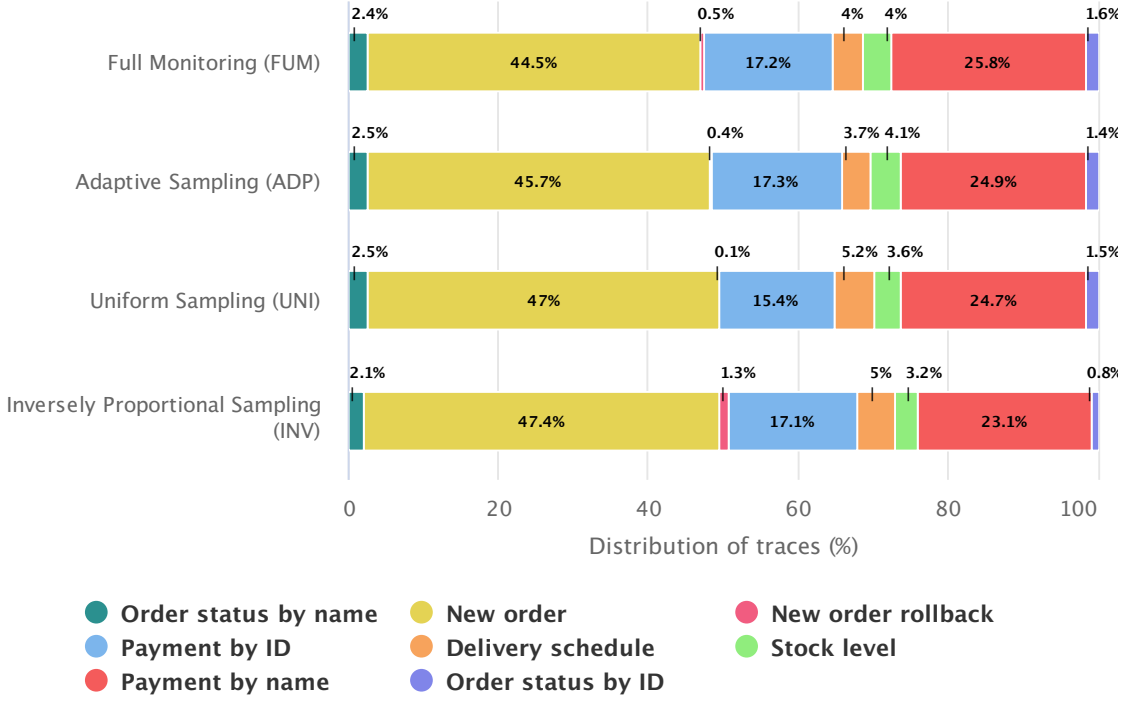
(b) ADP Sampling Rate vs. INV Sampling Rate.

Figure 5.3 – Analysis of the ADP Results with the `h2` Application.

sampling rate at a value close to 50% (label A), with small decreases due to variances in the response time, as this is the metric used to adapt the sampling rate. In seasonal patterns, ADP detects performance degradation and reduces the sampling rate (label B).



(a) Collected sample sizes over time.



(b) Comparison of the distribution of application requests.

Figure 5.4 – Analysis of the ADP Results with the `h2` Application.

Note that even with a decreased sampling rate, the throughput (orange line) decreases, showing that the user requests are causing the application to be under stress. Lastly, in isolated bursts, the sampling rate remains at 50% (label C) because the increased number

of users for brief moments does not have a major impact on the application performance. As can be seen, despite the monitoring and the peeks, the throughput is not lower than in the rest of the simulation.

ADP Sampling Rate vs. INV Sampling Rate. Now we look in detail at the sampling rate controlled by INV and how it differs from ADP. Both approaches apply mechanisms to reduce the sampling rate when the application is struggling with an increased workload (label D). However, while ADP uses the response time to make decisions, INV relies on the workload (throughput). In many cases, this correctly reduces the sampling rate to not cause a major performance impact on the application. But in certain situations (label E), low throughput is due to a low number of requests, thus there is no need to reduce the sampling rate. ADP is able to better understand the application as a whole as it keeps track of a performance baseline with and without monitoring, allowing it to identify when the monitoring is competing for resources with the application.

Collected Sample Sizes. ADP does not focus on collecting execution traces to be analyzed all together, but works in cycles providing a set of samples of execution traces, each being representative of the population in each cycle. Although there is a timeout for cycles, ideally the cycle ends when the representativeness criteria are met, leading to samples of various sizes. We present the collected samples sizes for `h2` in Figure 5.4a. The horizontal proximity between the dots indicates that no cycle reached the timeout of 180s—the maximum cycle time is 25 seconds. Figure 5.4a also shows that the sample size is not correlated to the sampling rate. This may occur in `h2` due to the low number of types of requests (8 distinct types) because it is easier to have similar distributions when the number of classes to be compared between the sample and population is low. Note that there is an outlier cycle composed only of 211 execution traces and that lasted less than 1s. This indicates that the sample satisfied the representativeness criteria with high confidence because the longer the cycle, the lower the confidence level as it decays over time. As result, on average, `h2` had 163 cycles. Because the request types of `xalan` and `tradebeans` are also low, 16 and 12, respectively, they manage to collect representative samples in shorter times, resulting in 422 and 151 cycles, respectively. `cassandra` and `lusearch`, in turn, have more than 100 request types, causing the lowest number of cycles (122 and 48, respectively). We observed longer monitoring cycles in these two applications, including timeouts.

Distribution of Application Requests. Lastly, we analyze the distribution of application requests in Figure 5.4b considering the data in all samples collected during one execution of the simulation of `h2`. FUM shows the distribution of the population (ground truth). Although ADP checks for distribution similarity by sample, the resulting set of samples has the distribution most similar to the population (considering the whole simulation), having the `new order` request the highest difference (1.2%). This request type, which is the most frequent, also led to the highest difference for UNI and INV. UNI has a difference of 2.5%; while INV, which focuses on performance rather than representativeness, has the highest difference (2.9%) among the three approaches.

5.2.2.2 Threats to Validity

Our evaluation involves runtime execution with a particular workload and, thus, there are many settings that may influence the results. All our settings were selected to avoid bias. The fired application requests have a key role in the obtained results. To minimize the chance of using a workload that favors a particular approach, we rely on the randomness and reliability provided by DaCapo. Another workload configuration that may influence the results is the number of simultaneous users and how it varies over time. Our designed workload includes different types of variations, which are those used in existing work. Moreover, the maximum number of users, based on preliminary executions, was selected to guarantee that the application executes under stress in certain moments. Another construction threat to validity is how we assess representativeness. The key goal is to evaluate whether the desired execution traces are included in the sample. Given that this depends on the monitoring goal, we use memory usage due to the reasons explained in the study settings. This measurement is not used by any of the compared approaches for adapting the sampling rate or making decisions. The only challenge is to collect this information in Java, because its virtual machine offers limited support to fine-grained memory measurements and, in addition, it has multiple features that can affect this kind of measurement during the application execution, such as garbage collector and just-in-time compilation. We used a standard way to measure memory usage as well as discarded invalid measurements—negative values due to the execution of the garbage collector—for all approaches, including FUM. An external threat to validity is the selected applications. We selected applications of different domains and that use various technologies. Although the number of applications is not large, we emphasize that the obtained results are consistent across all applications and thus provide evidence of the

generalization of the results. However, as any empirical study, further evaluations with different settings would improve the generality and reliability of the results.

5.3 Limitations

We now point out limitations of our monitoring process. A monitoring cycle finishes when the representativeness criteria are met. In situations that an application must timely react to particular requests, this may cause the application to give a delayed response. We addressed this issue using a decaying confidence level based on the monitoring cycle time frame, which can be customized. However, the more elapsed time, the lower the confidence level. Therefore, if an application requires samples with some confidence level guarantees, the sample evaluation activity must be adapted.

In our work, we monitor applications by continuously making decisions and adaptations to collect execution traces, which implies an overhead higher than making simple adjustments (as in INV). This is, however, done in a lightweight way and our evaluation showed that despite the execution of our process activities, we obtain the most representative samples with a performance not far from INV. Yet, it is possible to reduce the cost of the Sampling Rate Adaptation activity by using bootstrapping and other statistical techniques to generate data from samples and estimate the population based on monitoring time frames instead of instrumenting all the requests. Then, the monitoring can be disabled for extended periods when the sample is in good shape to be used in bootstrapping.

5.4 Final Remarks

We presented in this chapter an adaptive sampling process to find the sweet spot between two conflicting goals, namely overhead vs. representativeness. Our process is performed in monitoring cycles and is composed of three activities, which use algorithms with statistical foundations to decide whether a particular application request must be recorded, when and to what degree adapt the sampling rate, and determine when a sample has been collected to, then, begin a new monitoring cycle. We evaluated our process by comparing it with monitoring performed with uniform sampling and a sampling rate that is inversely proportional to the workload (INV) as well as used executions with no monitoring and monitoring every application request as a reference. Our results show

that our approach collects samples with the lowest errors with respect to the population, having a performance impact that is only 1–6% higher than INV, which achieves the highest errors. We next conclude this thesis, summarising its main contributions and pointing out directions for future work.

6 CONCLUSION

Software runtime monitoring has been largely used for a wide range of purposes, from debugging to self-adaptation. When it collects costly information like detailed execution traces in production environments, it is crucial to prevent the monitoring from causing unacceptable overhead. Typical approaches to address this issue rely on filtering or sampling strategies to reduce the monitoring overhead and enable faster trace analysis. However, such strategies have been commonly adopted with pre-defined and fixed configurations, which specify specific software locations to be monitored and a sampling rate. These configurations may be unsuitable to cope with software usage peaks and unable to handle unforeseen scenarios.

In this thesis, we increase the practical feasibility of software runtime monitoring by leveraging filtering and sampling strategies to reduce the monitoring overhead and increase the relevance and representativeness of the collected traces. We performed a systematic literature review to understand and identify relevance criteria and metrics used in monitoring approaches. Based on the literature review results, we derived a domain-specific language (DSL), named TigrisDSL, which allows the specification of monitoring filters through high-level relevance criteria. These relevance filters can guide monitoring components to collect a set of relevant traces that are analyzed to achieve the goal of monitoring. This DSL was incorporated into a proposed two-phase approach for adaptive filtering of execution traces. The filtering approach is domain neutral and can be instantiated to collect relevant traces for different domains and purposes. The first phase of our approach is based on obtaining lightweight metrics from the system's execution. Then, periodically, it evaluates the collected metrics according to a set of relevance criteria specified in TigrisDSL. The second phase filters and samples executions relevant to the goal of monitoring, thus achieving a reduced monitoring overhead. In addition, to address the limitations associated with sampling, we proposed an adaptive sampling process to collect execution traces with detailed information in environments where the performance impact is critical, such as production environments. The adaptive sampling is performed in monitoring cycles and is composed of three activities, which use algorithms with statistical foundations to decide whether a particular application request must be recorded, when and to what degree adapt the sampling rate, and determine when a sample has been collected to, then, begin a new monitoring cycle.

Both adaptive techniques were implemented in Java for our empirical evaluation,

but they are generic and language independent. These implementations served as basis for conducting empirical studies to assess different aspects of the proposed techniques. Regarding our research question, our results showed evidence that our proposals can collect execution traces at runtime that are relevant and representative with an acceptable performance impact. Thus, we conclude that our proposed adaptive solutions can be used as monitoring components of existing monitoring-based approaches in different domains and applications.

6.1 Contributions

Various contributions can be enumerated as a result of the work presented in this thesis. Together, they frame our solutions to increase the practical feasibility of software monitoring.

Research Topic Refinement. In the context of this thesis, while further exploring the applications of software runtime monitoring, we performed research work that helped in the understanding of the existing problems and idealization of possible solutions to be applied. During a three-month research visit at the University of Grenoble (FR), we investigated the monitoring needs and opportunities in IoT scenarios, such as smart homes. This investigation resulted in two research papers (LALANDA; MERTZ; NUNES, 2018; MERTZ et al., 2017), which are focused on monitoring communications between devices and pervasive platforms to identify caching opportunities and, consequently, improving the resilience and performance of the environment. In collaboration with researchers from TU Darmstadt and ETH Zurich, we extended previous work to define application-level caching as a research area and pinpoint research opportunities (MERTZ et al., 2020), highlighting the monitoring and understanding of applications as one of the main challenges in the area.

Relevance Criteria and TigrisDSL. Chapter 3 presented the design and results of a systematic literature review to identify relevance criteria and metrics that can be used for monitoring applications. This review also analyzed existing monitoring-based approaches in terms of generality, scalability, and adaptability (MERTZ; NUNES, 2019; MERTZ; NUNES, 2021). Based on the results of the literature review, we proposed TigrisDSL, a DSL that allows the specification of monitoring filters by means of high-level relevance

criteria. These monitoring filters can be used to capture and express different goals of monitoring in such a way that can be given as input for monitoring approaches to guide the collection of relevant traces for the given goal.

Adaptive Filtering Approach. In Chapter 4, we presented a two-phase monitoring approach for filtering execution traces at runtime (MERTZ; NUNES, 2019; MERTZ; NUNES, 2021). In its first phase, coarse-grained monitoring is performed to identify relevant parts of the software execution. It receives the goal of monitoring as input, expressed in TigrisDSL in the form of high-level relevance criteria. The criteria are translated into software metrics, which are collected and analyzed at runtime to guide in-depth and fine-grained monitoring in the second phase of the approach. Such filtering approach tailors the monitoring process according to the goal of monitoring, keeping its overhead at an acceptable level while collecting relevant traces. Consequently, the approach can be used as a monitoring component to effectively monitor a software system and provide information for different purposes, e.g. to identify security vulnerabilities, model inconsistencies or performance bugs.

Tigris Framework. Chapter 4 also presented a framework, named Tigris, which is one instance of the adaptive filtering approach that can seamlessly integrate the proposed solution to existing software systems to support monitoring-based activities. The goal of the framework is to decouple the monitoring phase of monitoring-based approaches, which is domain independent, from the analysis they perform (i.e. the goal of monitoring), which is domain specific, thus being possible to reuse the same monitoring process in different domains by defining the goal of monitoring in form of a set of relevance criteria and metrics. Thus, the framework provides reusable behavior based on minimum domain-specific input (MERTZ; NUNES, 2019; MERTZ; NUNES, 2021).

Adaptive Sampling Process. The process described in Chapter 5, provides a sampling strategy that pursues an adequate trade-off between monitoring overhead and representativeness of the collected sample of traces (MERTZ; NUNES, 2022. Submitted.). This process is performed in monitoring cycles and is composed of algorithms with statistical foundations to pursue that, by the end of each monitoring cycle, the collected sample is representative of the population. During the monitoring cycles, the proposed adaptive sampling dynamically decides which traces to keep and adapts the sampling rate towards

reducing the monitoring overhead to an acceptable level when the target software application needs its resources for regular processing. To empirically evaluate our proposal, we implemented our process and algorithms into a selected platform. However, our adaptive sampling proposal is independent of the programming language paradigm and technology.

6.2 Future Work

The contributions presented in this thesis advance research on the development of monitoring approaches. However, there still remains several open challenges in this context that should be addressed in future work. These challenges are discussed as follows.

Combination of Adaptive Filtering and Adaptive Sampling. We evaluated each strategy in different and specific empirical studies that were designed to solely assess the benefits and impacts of the proposed solutions without the influence of other components. However, the proposed solutions for adaptive filtering and adaptive sampling could be combined, as they essentially address different problems regarding the monitoring of software applications. Therefore, it is future work an evaluation of both approaches combined to reduce the monitoring overhead and increase the relevance and representativeness of collected traces at the same time.

Self adaptation with an Enhanced Feedback Loop. Our proposed approaches for adaptive filtering and sampling (Chapters 4 and 5, respectively) would benefit of enhanced self-adaptive capabilities, such as a complete feedback loop that takes the effectiveness of the domain-specific analysis of the collected traces as feedback to adjust the approach parameters (e.g. increasing the sampling rate or expanding the set of filtered methods). This would provide means of improving the results in terms of relevance and representativeness by leveraging domain-specific information, provided either manually, at design time, or dynamically, at runtime.

User Study and Case Studies. The evaluations conducted in Chapters 4 and 5 demonstrated that the proposals presented in this thesis have the potential to be reused in different application domains. Nevertheless, a user study that assesses the benefits of its reuse from the perspective of developers would provide valuable information regarding its strengths and weaknesses while monitoring software systems in production environment with real

workloads. In addition, our future work involves implementing and evaluating our approaches in projects written in other programming languages and for different purposes, such as post-mortem fault analysis.

In summary, this thesis advances research on reducing the impact of software runtime monitoring, paving the way to cover more unaddressed monitoring challenges in the future. Further work is required to develop a general solution that provides effective monitoring with practically acceptable costs, but our work is one of the steps in this direction.

REFERENCES

ALIABADI, M. R. et al. ARTINALI: Dynamic invariant detection for Cyber-Physical System security. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. New York, New York, USA: Association for Computing Machinery, 2017. Part F1301, p. 349–361. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3106237.3106282>.

ALONSO, J. et al. Towards Self-adaptable monitoring framework for self-healing. **Grid and Services Evolution**, p. 1–9, 2009. Available from Internet: <http://www.springerlink.com/index/K85L5065Q453W2P7.pdf>.

ANGELOPOULOS, K. et al. Model predictive control for software systems with CobRA. In: **Proceedings - 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2016**. [S.l.]: Association for Computing Machinery, Inc, 2016. p. 35–46.

APIWATTANAPONG, T.; HARROLD, M. J. Selective path profiling. **ACM SIGSOFT Software Engineering Notes**, 2003.

BAILEY, C. et al. Run-time generation, transformation, and verification of access control models for self-protection. In: **Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014**. New York, New York, USA: ACM Press, 2014. p. 135–144. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2593929.2593945>.

BARNA, C. et al. Hogna: A Platform for Self-Adaptive Applications in Cloud Environments. In: **Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2015. p. 83–87.

BARNA, C. et al. Delivering Elastic Containerized Cloud Applications to Enable DevOps. In: **Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2017. p. 65–75.

BARTOCCI, E. et al. Introduction to runtime verification. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Springer, Cham, 2018. v. 10457 LNCS, p. 1–33. Available from Internet: <http://link.springer.com/10.1007/978-3-319-75632-5_1>.

BASTANI, O.; ANAND, S.; AIKEN, A. Interactively verifying absence of explicit information flows in android apps. In: **Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**. New York, New York, USA: Association for Computing Machinery, 2015. v. 25-30-Oct-, p. 299–315. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2814270.2814274>.

BIELIK, P.; RAYCHEV, V.; VECHEV, M. Scalable race detection for android applications. In: **Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**. New York, New York, USA: Association for Computing Machinery, 2015. p. 332–348. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2814270.2814303>.

BLACKBURN, S. M. et al. The DaCapo benchmarks: Java benchmarking development and analysis. In: **OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications**. New York, NY, USA: ACM Press, 2006. p. 169–190.

BLACKBURN, S. M. et al. **The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version)**. [S.l.], 2006. Http://www.dacapobench.org.

BRAND, T.; GIESE, H. Towards generic adaptive monitoring. In: **International Conference on Self-Adaptive and Self-Organizing Systems, SASO**. IEEE, 2018. v. 2018-September, p. 156–161. Available from Internet: <https://ieeexplore.ieee.org/document/8614290/>.

BRAND, T.; GIESE, H. Generic Adaptive Monitoring Based on Executed Architecture Runtime Model Queries and Events. In: **IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)**. [s.n.], 2019. p. 17–22. Available from Internet: <https://ieeexplore.ieee.org/document/8780535/>.

BROCANELLI, M.; WANG, X. Smartphone Radio Interface Management for Longer Battery Lifetime. In: **Proceedings - 2017 IEEE International Conference on Autonomic Computing, ICAC 2017**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2017. p. 93–102.

BRÖNINK, M.; ROSENBLUM, D. S. Mining performance specifications. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2016. p. 39–49.

CÁMARA, J.; MORENO, G. A.; GARLAN, D. Stochastic game analysis and latency awareness for proactive self-adaptation. In: **9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014 - Proceedings**. New York, New York, USA: Association for Computing Machinery, 2014. p. 155–164. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2593929.2593933>.

CASANOVA, P. et al. Diagnosing unobserved components in self-adaptive systems. In: **9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014 - Proceedings**. New York, New York, USA: Association for Computing Machinery, 2014. p. 75–84. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2593929.2593946>.

CASSAR, I. et al. A Survey of Runtime Monitoring Instrumentation Techniques. aug 2017. Available from Internet: <http://arxiv.org/abs/1708.07229http://dx.doi.org/10.4204/EPTCS.254.2>.

CHAN, A. et al. Scaling an object-oriented system execution visualizer through sampling. In: **Proceedings - IEEE Workshop on Program Comprehension**. IEEE Comput. Soc, 2003. v. 2003-May, p. 237–244. Available from Internet: <http://ieeexplore.ieee.org/document/1199207/>.

CHEN, T.-H. et al. CacheOptimizer: Helping Developers Configure Caching Frameworks for Hibernate-based Database-centric Web Applications. In: **Proceedings of the 2016**

24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016. New York, New York, USA: ACM Press, 2016. p. 666–677. Available from Internet: <http://dx.doi.org/10.1145/2950290.2950303>.

CHEN, T.-H. et al. Detecting performance anti-patterns for applications developed using object-relational mapping. In: **Proceedings of the 36th International Conference on Software Engineering - ICSE 2014**. New York, New York, USA: ACM Press, 2014. p. 1001–1012. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2568225.2568259>.

CHEN, Z. et al. Speedoo. In: **Proceedings of the 40th International Conference on Software Engineering - ICSE '18**. New York, New York, USA: ACM Press, 2018. p. 811–821. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3180155.3180229>.

CHRISTAKIS, M. et al. A General Framework for Dynamic Stub Injection. In: **Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2017. p. 586–596.

CLARK, S. S.; BEAL, J.; PAL, P. Distributed Recovery for Enterprise Services. In: **International Conference on Self-Adaptive and Self-Organizing Systems, SASO**. [S.l.]: IEEE Computer Society, 2015. v. 2015-Octob, p. 111–120.

COCHRAN, W. G. **Sampling techniques**. New York, New York, USA: John Wiley & Sons, Ltd, 1977. 89 – 149 p. Available from Internet: <https://archive.org/details/Cochran1977SamplingTechniques{_}201703/page>.

CORNELISSEN, B. et al. Execution trace analysis through massive sequence and circular bundle views. **Journal of Systems and Software**, v. 81, n. 12, p. 2252–2268, 2008.

DAOUD, H.; EZZATI-JIVAN, N.; DAGENAIS, M. R. Dynamic trace-based sampling algorithm for memory usage tracking of enterprise applications. In: **2017 IEEE High Performance Extreme Computing Conference, HPEC 2017**. IEEE, 2017. p. 1–7. Available from Internet: <http://ieeexplore.ieee.org/document/8091061/>.

DELLA TOFFOLA, L.; PRADEL, M.; GROSS, T. R. Performance problems you can fix: a dynamic analysis of memoization opportunities. In: **Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015**. New York, New York, USA: ACM Press, 2015. p. 607–622. Available from Internet: <http://dx.doi.org/10.1145/2814270.2814290>.

DENARO, G. et al. Dynamic data flow testing of object oriented systems. In: **Proceedings - International Conference on Software Engineering**. [S.l.]: IEEE Computer Society, 2015. v. 1, p. 947–958.

DEVRIES, B.; CHENG, B. H. Run-time monitoring of self-adaptive systems to detect N-way feature interactions and their causes. In: **Proceedings - International Conference on Software Engineering**. New York, New York, USA: IEEE Computer Society, 2018. p. 94–100. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3194133.3194141>.

DING, R. et al. Log2: A cost-aware logging mechanism for performance diagnosis. In: **Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015**. Santa Clara, CA, USA: USENIX Association, 2015. p. 139–150.

DONG, F. et al. FraudDroid: Automated ad fraud detection for android apps. In: **ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, New York, USA: Association for Computing Machinery, Inc, 2018. p. 257–268. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3236024.3236045>.

DUGERDIL, P. Using trace sampling techniques to identify dynamic clusters of classes. In: **Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '07**. New York, New York, USA: ACM Press, 2007. p. 306–314.

EICHELBERGER, H.; SCHMID, K. Flexible resource monitoring of Java programs. **Journal of Systems and Software**, 2014.

ESTABROOKS, A.; JO, T.; JAPKOWICZ, N. A multiple resampling method for learning from imbalanced data sets. **Computational Intelligence**, v. 20, n. 1, p. 18–36, 2004.

FEI, L.; MIDKIFF, S. P. Artemis. In: **Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation - PLDI '06**. New York, New York, USA: ACM Press, 2006. v. 41, n. 6, p. 84. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1133981.1133992>.

FENG, Y. et al. Hierarchical abstraction of execution traces for program comprehension. In: **Proceedings of the 26th Conference on Program Comprehension - ICPC '18**. Association for Computing MachineryACM ACM, 2018. p. 86–96. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3196321.3196343>.

FINOCCHI, I. Software streams: Big data challenges in dynamic program analysis. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [S.l.: s.n.], 2013. v. 7921 LNCS, p. 124–134.

FISCHMEISTER, S.; BA, Y. Sampling-based program execution monitoring. In: **ACM SIGPLAN Notices**. New York, New York, USA: ACM Press, 2010. v. 45, n. 4, p. 133. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1755951.1755908>.

GAO, L. et al. A survey of software runtime monitoring. In: **2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)**. Beijing, China: IEEE, 2017. p. 308–313.

GHEZZI, C. et al. Mining behavior models from user-intensive web applications. In: **Proceedings - International Conference on Software Engineering**. New York, New York, USA: IEEE Computer Society, 2014. p. 277–287. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2568225.2568234>.

GONG, L.; PRADEL, M.; SEN, K. JITProf: pinpointing JIT-unfriendly JavaScript code. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015**. New York, New York, USA: ACM Press, 2015. p. 357–368. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2786805.2786831>.

HAMOU-LHADJ, A.; LETHBRIDGE, T. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: **IEEE International Conference on Program Comprehension**. [S.l.: s.n.], 2006. v. 2006, p. 181–190.

HAMOU-LHADJ, A.; LETHBRIDGE, T. C. **A Survey of Trace Exploration Tools and Techniques**. IBM Press, 2004. 42–55 p. Available from Internet: <http://dl.acm.org/citation.cfm?id=1034914.1034918>.

HAUSWIRTH, M.; CHILIMBI, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In: **Proceedings of the 11th international conference on Architectural support for programming languages and operating systems - ASPLOS-XI**. New York, New York, USA: ACM Press, 2004. v. 39, n. 11, p. 156–164. Available from Internet: <http://portal.acm.org.www.library.gatech.edu:2048/citation.cfm?id=1024393.1024412>.

HAWKINS, B.; DEMSKY, B. ZenIDS: Introspective Intrusion Detection for PHP Applications. In: **Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2017. p. 232–243.

HE, J.; DAI, T.; GU, X. TScope: Automatic timeout bug identification for server systems. In: **Proceedings - 15th IEEE International Conference on Autonomic Computing, ICAC 2018**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2018. p. 1–10.

HOLMQVIST, V.; NILSFORS, J.; LEITNER, P. Cachematic - Automatic Invalidation in Application-Level Caching Systems. In: **Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering - ICPE '19**. New York, New York, USA: ACM Press, 2019. p. 167–178. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3297663.3309666>.

HORKÝ, V. et al. Analysis of Overhead in Dynamic Java Performance Monitoring. In: **Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16**. New York, New York, USA: ACM Press, 2016. p. 275–286. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2851553.2851569>.

HUANG, J.; LUO, Q.; ROSU, G. GPredict: Generic predictive concurrency analysis. In: **Proceedings - International Conference on Software Engineering**. [S.l.]: IEEE Computer Society, 2015. v. 1, p. 847–857.

HUEBSCHER, M. C.; MCCANN, J. a. A survey of autonomic computing—degrees, models, and applications. **ACM Computing Surveys**, ACM, v. 40, n. 3, p. 1–28, aug 2008. Available from Internet: <http://dl.acm.org/citation.cfm?id=1380584.1380585>.

JENSEN, S. H. et al. MemInsight: platform-independent memory debugging for JavaScript. In: **Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015**. New York, New York, USA: ACM Press, 2015.

p. 345–356. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2786805.2786860>.

JUNG, C. et al. Automated memory leak detection for production use. In: **Proceedings - International Conference on Software Engineering**. [S.l.: s.n.], 2014. p. 825–836.

KANG, P. Function call interception techniques. **Software - Practice and Experience**, v. 48, n. 3, p. 385–401, may 2018. Available from Internet: <http://doi.wiley.com/10.1002/spe.2501>.

KANG, Y. et al. DiagDroid: Android performance diagnosis via anatomizing asynchronous executions. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. New York, New York, USA: Association for Computing Machinery, 2016. p. 410–421. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950316>.

KANTERT, J. et al. A Graph Analysis Approach to Detect Attacks in Multi-agent Systems at Runtime. In: **International Conference on Self-Adaptive and Self-Organizing Systems, SASO**. [S.l.]: IEEE Computer Society, 2014. v. 2014-Decem, n. December, p. 80–89.

KICZALES, G. et al. Aspect-oriented programming. **European conference on object-oriented programming**, v. 1241/1997, n. June, p. 220–242, 1997.

KIM, C. H. et al. PerfGuard: binary-centric application performance monitoring in production environments. In: **Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016**. New York, New York, USA: ACM Press, 2016. p. 595–606. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950347>.

KIM, C. H. et al. PerfGuard: binary-centric application performance monitoring in production environments. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. New York, New York, USA: ACM Press, 2016. v. 13-18-Nove, p. 595–606. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950347>.

KIM, D. et al. Apex: Automatic programming assignment error explanation. In: **Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**. New York, New York, USA: Association for Computing Machinery, 2016. v. 02-04-Nove, p. 311–327. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2983990.2984031>.

KNUTH, D. E. **The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms**. USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

KOUAME, K.; EZZATI-JIVAN, N.; DAGENAIS, M. R. A Flexible Data-Driven Approach for Execution Trace Filtering. In: **Proceedings - 2015 IEEE International Congress on Big Data, BigData Congress 2015**. IEEE, 2015. p. 698–703. Available from Internet: <http://ieeexplore.ieee.org/document/7207296/>.

LALANDA, P.; MCCANN, J. a.; DIACONESCU, A. **Autonomic Computing. Principles, Design and Implementation**. 1. ed. Springer London, 2013. XV, 288 p. Available from Internet: <http://dx.doi.org/10.1007/978-1-4471-5007-7>.

LALANDA, P.; MERTZ, J.; NUNES, I. Autonomic caching management in industrial smart gateways. In: **IEEE Industrial Cyber-Physical Systems, ICPS 2018**. IEEE, 2018. p. 26–31. Available from Internet: <https://ieeexplore.ieee.org/document/8387632/>.

LARSSON, L. et al. Quality-Elasticity: Improved Resource Utilization, Throughput, and Response Times Via Adjusting Output Quality to Current Operating Conditions. In: **Proceedings - 2019 IEEE International Conference on Autonomic Computing, ICAC 2019**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2019. p. 52–62.

LAS-CASAS, P. et al. Weighted Sampling of Execution Traces. In: **Proceedings of the ACM Symposium on Cloud Computing - SoCC '18**. New York, New York, USA: ACM Press, 2018. p. 326–332. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3267809.3267841>.

LAS-CASAS, P. et al. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In: **Proceedings of the ACM Symposium on Cloud Computing - SoCC '19**. New York, New York, USA: Association for Computing Machinery (ACM), 2019. p. 312–324. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3357223.3362736>.

LEE, G. J.; FORTES, J. A. Hadoop performance self-tuning using a fuzzy-prediction approach. In: **Proceedings - 2016 IEEE International Conference on Autonomic Computing, ICAC 2016**. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2016. p. 55–64.

LEE, S.; JUNG, C.; PANDE, S. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In: **Proceedings - International Conference on Software Engineering**. New York, New York, USA: IEEE Computer Society, 2014. p. 814–824. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2568225.2568307>.

LIU, T.; CURTSINGER, C.; BERGER, E. D. DOUBLETAKE: Fast and precise error detection via evidence-based dynamic analysis. In: **Proceedings - International Conference on Software Engineering**. New York, New York, USA: IEEE Computer Society, 2016. v. 14-22-May-, p. 911–922. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2884781.2884784>.

MADSEN, M. et al. Feedback-directed instrumentation for deployed JavaScript applications. In: **Proceedings of the 38th International Conference on Software Engineering - ICSE '16**. New York, New York, USA: ACM Press, 2016. p. 899–910. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2884781.2884846>.

MERTZ, J.; NUNES, I. A Qualitative Study of Application-Level Caching. **IEEE Transactions on Software Engineering**, v. 43, n. 9, p. 798–816, sep 2017. Available from Internet: <https://doi.org/10.1109/TSE.2016.2633992>.

MERTZ, J.; NUNES, I. Understanding Application-Level Caching in Web Applications. **ACM Computing Surveys**, v. 50, n. 6, p. 1–34, 2017. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3161158.3145813>.

MERTZ, J.; NUNES, I. Automation of application-level caching in a seamless way. **Software - Practice and Experience**, John Wiley & Sons, Ltd., v. 48, n. 6, p. 1218–1237, 2018. Available from Internet: <http://onlinelibrary.wiley.com/doi/10.1002/spe.2571/abstract>.

MERTZ, J.; NUNES, I. On the practical feasibility of software monitoring: A framework for low-impact execution tracing. In: **Proceedings of the 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems**. IEEE Press, 2019. p. 169–180. Available from Internet: <https://doi.org/10.1109/SEAMS.2019.00030>.

MERTZ, J.; NUNES, I. Tigris: A DSL and framework for monitoring software systems at runtime. **Journal of Systems and Software**, v. 177, p. 110963, 2021.

MERTZ, J.; NUNES, I. Software Runtime Monitoring with an Adaptive Sampling Rate to Collect Representative Samples of Execution Traces. **IEEE Transactions on Software Engineering**, 2022. Submitted.

MERTZ, J. et al. Satisfying increasing performance requirements with caching at the application level. **IEEE Software**, Institute of Electrical and Electronics Engineers (IEEE), 2020. Available from Internet: <http://dx.doi.org/10.1109/MS.2020.3033508>.

MERTZ, J. et al. Autonomic management of context data based on application requirements. In: **43rd Annual Conference of the IEEE Industrial Electronics Society**. [s.n.], 2017. v. 2017-January, p. 8622–8627. Available from Internet: <http://ieeexplore.ieee.org/document/8217515/>.

MIRANSKYY, A. et al. Operational-Log Analysis for Big Data Systems: Challenges and Solutions. **IEEE Software**, v. 33, n. 2, p. 52–59, 2016.

MIRANSKYY, A. V. et al. Sift: A Scalable Iterative-Unfolding Technique for Filtering Execution Traces. In: **Proceedings of the 2008 conference of the center for advanced studies on collaborative research meeting of minds - CASCON '08**. New York, New York, USA: ACM Press, 2008. p. 274. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1463788.1463817>.

MIZOUCHI, T. et al. PADLA: A Dynamic Log Level Adapter Using Online Phase Detection. In: **International Conference on Program Comprehension (ICPC)**. [S.l.: s.n.], 2019. p. 135–138.

NARAYANAPPA, H.; BANSAL, M. S.; RAJAN, H. Property-aware program sampling. In: **Proceedings of the 9th ACM SIGPLANSIGSOFT workshop on Program analysis for software tools and engineering PASTE 10**. New York, New York, USA: ACM Press, 2010. p. 45. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1806672.1806682>.

PIRZADEH, H. et al. The concept of stratified sampling of execution traces. In: **IEEE International Conference on Program Comprehension**. IEEE, 2011. p. 225–226. Available from Internet: <http://ieeexplore.ieee.org/document/5970192/>.

PIRZADEH, H. et al. Stratified sampling of execution traces: Execution phases serving as strata. **Science of Computer Programming**, v. 78, n. 8, p. 1099–1118, aug 2013. Available from Internet: <http://linkinghub.elsevier.com/retrieve/pii/S0167642312002080>.

RABISER, R. et al. A comparison framework for runtime monitoring approaches. **Journal of Systems and Software**, Elsevier, v. 125, p. 309–321, mar 2017. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0164121216302618>.

REGER, G.; HAVELUND, K. What is a trace? A runtime verification perspective. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Springer, Cham, 2016. v. 9953 LNCS, p. 339–355. Available from Internet: <http://link.springer.com/10.1007/978-3-319-47169-3_25>.

REISS, S. P. Dynamic Detection and Visualization of Software Phases. **Proceedings of the third international workshop on Dynamic analysis (WODA 2005)**, v. 30, n. 4, p. 1, 2005.

SAMAK, M.; TRIPP, O.; RAMANATHAN, M. K. Directed synthesis of failing concurrent executions. In: **Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA**. New York, New York, USA: Association for Computing Machinery, 2016. v. 02-04-Nove, p. 430–446. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2983990.2984040>.

SANTELICES, R.; SINHA, S.; HARROLD, M. J. Subsumption of program entities for efficient coverage and monitoring. **Proceedings of the 3rd international workshop on Software quality assurance - SOQUA '06**, 2006.

SHEN, D. et al. Automating performance bottleneck detection using search-based application profiling. In: **Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015**. New York, New York, USA: ACM Press, 2015. p. 270–281. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2771783.2771816>.

SONG, L.; LU, S. Performance diagnosis for inefficient loops. In: **Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017**. [S.l.: s.n.], 2017. p. 370–380.

SRIDHARAN, M.; FINK, S. J.; BODIK, R. Thin Slicing. **PLDI '07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation**, 2007.

SU, F. H. et al. Code relatives: Detecting similarly behaving software. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. New York, New York, USA: Association for Computing Machinery, 2016. v. 13-18-Nove, p. 702–714. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950321>.

SU, F. H. et al. Identifying functionally similar code in complex codebases. In: **IEEE International Conference on Program Comprehension**. [S.l.]: IEEE Computer Society, 2016. v. 2016-July.

SU, P. et al. Redundant Loads: A Software Inefficiency Indicator. In: **Proceedings - International Conference on Software Engineering**. [S.l.]: IEEE Computer Society, 2019. v. 2019-May, p. 982–993.

THOMAS, J. J.; FISCHMEISTER, S.; KUMAR, D. Lowering overhead in sampling-based execution monitoring and tracing. **ACM SIGPLAN Notices**, ACM, v. 46, n. 5, p. 101, apr 2011. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2016603.1967692>.

TOFFOLA, L. D.; PRADEL, M.; GROSS, T. R. Synthesizing Programs That Expose Performance Bottlenecks. **CGO: International Symposium on Code Generation and Optimization**, ACM, p. 314–326, 2018. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3168830>.

TUN, T. T. et al. Requirements and specifications for adaptive security. In: **Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '18**. New York, New York, USA: ACM Press, 2018. p. 161–171. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3194133.3194155>.

TUN, T. T. et al. Requirements and specifications for adaptive security: Concepts and analysis. In: **Proceedings - International Conference on Software Engineering**. New York, New York, USA: IEEE Computer Society, 2018. p. 161–171. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3194133.3194155>.

VAN HOORN, A.; WALLER, J.; HASSELBRING, W. Kieker: A framework for application performance monitoring and dynamic software analysis. In: **ICPE'12 - Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Performance Engineering**. [S.l.: s.n.], 2012. p. 247–248.

VIERHAUSER, M.; RABISER, R.; GRÜNBACHER, P. Requirements monitoring frameworks: A systematic review. **Information and Software Technology**, Elsevier, v. 80, p. 89–109, dec 2016. Available from Internet: <https://www.sciencedirect.com/science/article/abs/pii/S0950584916301288?dgcid=raven{_}sd{_}recommender>.

Von Kistowski, J. et al. Modeling and extracting load intensity profiles. **ACM Transactions on Autonomous and Adaptive Systems**, ACM New York, NY, USA, v. 11, n. 4, p. 1–28, jan 2017. Available from Internet: <https://dl.acm.org/doi/abs/10.1145/3019596>.

XIAO, T. et al. Cybertron: Pushing the limit on I/O reduction in data-parallel programs. **ACM SIGPLAN Notices**, v. 49, n. 10, p. 895–908, 2014.

XU, Z. et al. Python predictive analysis for bug detection. In: **Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering**. New York, New York, USA: Association for Computing Machinery, 2016. v. 13-18-Nove, p. 121–132. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2950290.2950357>.

YANDRAPALLY, R.; SRIDHARA, G.; SINHA, S. Automated modularization of GUI test cases. In: **Proceedings - International Conference on Software Engineering**. [S.l.]: IEEE Computer Society, 2015. v. 1, p. 44–54.

YANG, J. et al. View-Centric Performance Optimization for Database-Backed Web Applications. In: **Proceedings - International Conference on Software Engineering**. IEEE Computer Society, 2019. v. 2019-May, p. 994–1004. Available from Internet: <https://ieeexplore.ieee.org/document/8811938/>.

YUAN, E.; ESFAHANI, N.; MALEK, S. Automated mining of software component interactions for self-adaptation. In: **Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014**. New York, New York, USA: ACM Press, 2014. p. 27–36. Available from Internet: <http://dl.acm.org/citation.cfm?id=2593929.2593934>.

ZAIDMAN, A.; DEMEYER, S. Managing trace data volume through a heuristical clustering process based on event execution frequency. **Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.**, 2004.

ZAVALA, E.; FRANCH, X.; MARCO, J. Adaptive monitoring: A systematic mapping. **Information and Software Technology**, Elsevier, v. 105, p. 161–189, jan 2019. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0950584918301861>.

ZHANG, S.; ERNST, M. D. Which configuration option should i change? In: **Proceedings - International Conference on Software Engineering**. New York, New York, USA: IEEE Computer Society, 2014. p. 152–163. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2568225.2568251>.

ZHOU, N. et al. Autonomic parallelism and thread mapping control on software transactional memory. In: **Proceedings - 2016 IEEE International Conference on Autonomic Computing, ICAC 2016**. [S.l.: s.n.], 2016. p. 189–198.

ZHOU, X. et al. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In: **ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, New York, USA: Association for Computing Machinery, Inc, 2019. p. 683–694. Available from Internet: <http://dl.acm.org/citation.cfm?doid=3338906.3338961>.

7 RESUMO ESTENDIDO

À medida em que sistemas de software modernos se tornam cada vez maiores e complexos, métodos de análise eficazes para entender o comportamento de um sistema de software estão se tornando essenciais como ferramentas de suporte para diversas tarefas fundamentais de engenharia de software, como verificação de tempo de execução, depuração, compreensão do programa e auto-adaptação. Tal compreensão do comportamento de sistemas pode ser alcançada através de monitoração de traços de execução, que representam regras, componentes, funções ou tarefas executadas durante a execução de um programa. Um traço de execução pode ser composto de informações básicas sobre a execução, como nome do evento, local do programa e data, ou ainda dados mais complexos e detalhados sobre o evento como entradas, saídas, pilha de chamadas, mensagens comunicadas e uso de recursos. Portanto, analisar e comparar os traços de execução do programa pode ser benéfico para vários fins, como a validação dos requisitos de qualidade ou a identificação de vulnerabilidades de segurança. A solução corporativa, AWS X-Ray [1], por exemplo, é uma ferramenta que usa traços de execução para fornecer uma visão ponta a ponta dos caminhos de requisição, incluindo um mapa dos componentes subjacentes ao sistema. Isso ajuda, por exemplo, a identificar e solucionar a causa raiz de problemas e erros de desempenho.

Uma forma amplamente utilizada de coleta de traços de execução é a instrumentação, a qual é concebida a partir da interceptação de pontos específicos na execução do programa para coleta de informações. Essas instruções de código adicionais são responsáveis por coletar e armazenar informações em tempo de execução de componentes específicos de um sistema ou seu ambiente de execução enquanto o software está em execução, dependendo do objetivo de monitoração. No entanto, apesar da utilidade dos traços de execução, coletá-los em tempo de execução consome recursos do sistema e pode levar à queda de desempenho do sistema, principalmente quando incluem informações detalhadas, como parâmetros de métodos. Em ambientes de produção, onde é necessario garantir um tempo de resposta consistente, torna-se crucial evitar que a monitoração cause uma sobrecarga inaceitável no sistema.

Para amenizar os impactos da monitoração, traços de execução podem ser filtrados ou amostrados. As técnicas de *filtragem* geralmente consistem em focar na monitoração de execuções *relevantes*, ou seja, o subconjunto de traços coletado é focado em locais es-

[1] <https://aws.amazon.com/xray/>

pecíficos do sistema sendo monitorado. Em *amostragem*, o objetivo é coletar um subconjunto representativo de traços com base em uma taxa de amostragem ou estratégia, onde *representatividade* significa que as características do subconjunto de traços coletados devem corresponder ao conjunto completo de traços de execução (população), seguindo a mesma distribuição. A principal vantagem das técnicas de filtragem e amostragem é a sobrecarga limitada, que diminui linearmente conforme a configuração de filtragem e a taxa de amostragem.

No entando, apesar de serem adotadas como soluções, ambas as estratégias têm limitações inerentes. A filtragem carece de reutilização em diferentes sistemas, sendo personalizada para casos particulares, por exemplo, limitando a monitoração apenas à eventos de alto nível ou um conjunto predefinido de execuções. A amostragem geralmente se concentra na coleta de traços para uma finalidade específica ou depende de uma estratégia que não pode garantir que os traços coletados sejam representativos da população. Essas configurações podem ser inadequadas para lidar com picos de uso de software e são incapazes de lidar com cenários imprevistos.

Com base nestas limitações, a questão de pesquisa que guia esta tese é a seguinte. *Como monitorar sistemas de software para coletar traços de execução em tempo de execução que são relevantes e representativos, com um impacto de desempenho aceitável?*

Como a monitoração de sistemas consiste na execução de instruções adicionais, isso necessariamente implica em uma sobrecarga. Embora essa sobrecarga possa ser reduzida a um nível aceitável com o uso de técnicas de filtragem e amostragem, o trabalho existente se concentra apenas na coleta de traços para um propósito específico, são adotados com configurações predefinidas e fixas, ou se usa uma estratégia que não pode garantir que os traços coletados sejam relevantes e representativos da população. Ao abordar essas questões, é possível fornecer uma solução capaz de atingir um balanço adequado entre a sobrecarga de monitoração e a relevância ou representatividade das informações coletadas.

Como a filtragem e a amostragem são problemas essencialmente diferentes nos quais as soluções podem ser combinadas para resolver os problemas de monitoração e compor uma solução eficaz, essa tese aborda e endereça os problemas individualmente. Para resolver os problemas em torno da filtragem e fornecer uma solução que pode ser reutilizada em diferentes sistemas, propomos uma abordagem de monitoração de duas fases para filtrar os traços de execução em tempo de execução. Em sua primeira fase, se faz uso de uma monitoração de leve impacto para coletar metricas que possam auxiliar na identi-

ficação de partes relevantes da execução do sistemas. Como relevância é essencialmente um conceito específico de domínio, este processo é informado pela definição de *objetivo do monitoração*, dada pelo usuário, na forma de *critérios de relevância* de alto nível. Tal definição é expressa em uma linguagem específica de domínio proposta, chamada *Tigris-DSL*. Dessa forma, filtros de relevância podem orientar os componentes de monitoração para coletar um conjunto de traços relevantes que são analisados para atingir o objetivo de monitoração. Na prática, os critérios são traduzidos em métricas de software, que são coletadas e analisadas em tempo de execução para orientar uma monitoração detalhada e refinada na segunda fase da abordagem. Tanto os critérios de relevância quanto as métricas de software são derivados de uma revisão sistemática da literatura. Tal revisão sistemática da literatura foi realizada para compreender e identificar os critérios e métricas de relevância usados por abordagens que se utilizam de monitoração.

A abordagem proposta para filtragem foi implementada como um framework, chamado Tigris, que pode ser usado para integrar a solução proposta à sistemas de software existentes para oferecer suporte às atividades baseadas em monitoração. Consequentemente, nossa abordagem e framework podem ser usadas como um componente para monitorar um sistema de software e fornecer informações para diferentes fins, por exemplo, para identificar vulnerabilidades de segurança ou problemas de desempenho. Para avaliar nossa proposta, instanciamos o framework Tigris como o componente de monitoração de uma abordagem para identificação de oportunidade de cache em sistemas, a qual utiliza traços de execução como entrada para seu funcionamento. Nossa avaliação mostra que nossa proposta pode manter a eficácia da abordagem de cache, encontrando oportunidades relevantes, enquanto reduz a sobrecarga de monitoração.

Para lidar com as limitações associadas à amostragem, propomos um processo de amostragem adaptativo para coletar traços de execução com informações detalhadas em ambientes onde o impacto no desempenho é crítico, como ambientes de produção. Nosso objetivo é maximizar a representatividade das amostras de traços de execução ao ajustar a taxa de amostragem usada para monitorar um sistema de software para lidar com variações em sua carga de trabalho. O processo proposto é composto de três atividades: (1) *decisão de amostragem*, que decide se uma requisição (com traços de execução associados) deve ser registrada e incluída em uma amostra; (2) *adaptação da taxa de amostragem*, que ajusta a taxa de amostragem em tempo de execução de acordo com a carga de trabalho do sistema; e (3) *avaliação da amostra*, que avalia a representatividade da amostra para identificar o final de um ciclo de monitoração. Essas atividades incluem

algoritmos com fundamentos estatísticos para buscar que, ao final de cada ciclo de monitoração, a amostra coletada seja representativa da população. Para avaliar nossa proposta, utilizamos como alvo cinco sistemas do conhecido benchmark DaCapo, e os resultados mostram que nossa abordagem pode alcançar maior representatividade e menor impacto no desempenho do que as soluções existentes.

Portanto, ambas as soluções propostas aumentam a viabilidade prática de monitoração de software, reduzindo a sobrecarga de monitoração e buscando a relevância e representatividade dos traços coletados. As soluções propostas podem ser usadas individualmente ou combinadas para monitorar efetivamente da monitoração de modo a fornecer traços para abordagens com diferentes propósitos. Ambas as técnicas adaptativas foram implementadas em Java para avaliações empíricas, mas são genéricas e independentes da linguagem. Em relação à nossa questão de pesquisa, nossos resultados mostraram evidências de que nossas propostas podem coletar traços de execução que são relevantes e representativos com um impacto de desempenho aceitável. Assim, concluímos que nossas soluções adaptativas propostas podem ser usadas como componentes de monitoração de abordagens baseadas em monitoração existentes em diferentes domínios e aplicações.

Várias contribuições podem ser enumeradas como resultado do trabalho apresentado nesta tese. Juntas, elas representam a solução proposta para aumentar a viabilidade prática do monitoração de software.

Refinamento do tópico de pesquisa. No contexto desta tese, ao mesmo tempo em que exploramos as aplicações de monitoração de tempo de execução de software, realizamos um trabalho de pesquisa que auxiliou na compreensão dos problemas existentes em outras áreas e na idealização de possíveis soluções a serem aplicadas. Durante uma visita de pesquisa de três meses na Universidade de Grenoble (FR), investigamos as necessidades e oportunidades de monitoração em cenários de IoT, como casas inteligentes. Esta investigação resultou em dois artigos de pesquisa (LALANDA; MERTZ; NUNES, 2018; MERTZ et al., 2017), focados na monitoração de comunicações entre dispositivos e plataformas pervasivas para identificar oportunidades de cache e, consequentemente, em melhorar a resiliência e o desempenho do ambiente. Em colaboração com pesquisadores da TU Darmstadt e ETH Zurich, estendemos um trabalho anterior para definir cache de nível de aplicação como uma área de pesquisa e apontar trabalhos futuros (MERTZ et al., 2020), destacando a monitoração e compreensão do comportamento de sistemas como um dos principais desafios na área.

Critérios de relevância e TigrisDSL. Apresentamos os resultados de uma revisão sistemática da literatura para identificar critérios de relevância e métricas que podem ser usados em aplicações de monitoração. Esta revisão também analisou abordagens baseadas em monitoração existentes em termos de generalidade, escalabilidade e adaptabilidade (MERTZ; NUNES, 2019; MERTZ; NUNES, 2021). Com base nos resultados da revisão da literatura, propusemos o TigrisDSL, um DSL que permite a especificação de filtros de monitoração por meio de critérios de relevância de alto nível. Esses filtros de monitoração podem ser usados para capturar e expressar diferentes objetivos de monitoração de tal forma que podem ser fornecidos como entrada para outras abordagens para orientar a coleta de traços relevantes para um determinado objetivo.

Abordagem de filtragem adaptativa. Apresentamos uma abordagem de monitoração em duas fases para filtrar os traços de execução em tempo de execução (MERTZ; NUNES, 2019; MERTZ; NUNES, 2021). Em sua primeira fase, uma monitoração de baix impacto é aplicada para identificar as partes relevantes da execução do sistema. Ela recebe o objetivo de monitoração como entrada, expresso em TigrisDSL na forma de critérios de relevância de alto nível. Os critérios são traduzidos em métricas de software, que são coletadas e analisadas em tempo de execução para orientar a monitoração detalhada e refinada na segunda fase da abordagem. Essa abordagem de filtragem adapta o processo de monitoração de acordo com o objetivo de monitoração, mantendo sua sobrecarga em um nível aceitável enquanto coleta os traços relevantes. Consequentemente, a abordagem pode ser usada como um componente de monitoração para monitorar efetivamente um sistema de software e fornecer informações para diferentes fins.

Framework Tigris. Apresentamos um framework, chamado Tigris, que é uma instância da abordagem de filtragem adaptativa que pode ser integrada à sistemas para apoiar atividades baseadas em monitoração. O objetivo do framework é separar a fase de monitoração das abordagens baseadas em monitoração, que são independentes do domínio, da análise que as mesmas realizam (ou seja, o objetivo do monitoração), que é específico do domínio. Sendo assim, é possível reutilizar o mesmo processo de monitoração em domínios diferentes, definindo o objetivo de monitoração na forma de um conjunto de critérios e métricas de relevância. Assim, o framework fornece comportamento reutilizável com base na entrada mínima específica do domínio (MERTZ; NUNES, 2019; MERTZ; NUNES, 2021).

Processo de amostragem adaptativo. Apresentamos uma estratégia de amostragem que busca um balanço adequado entre sobrecarga de monitoração e representatividade da amostra de traços coletada (MERTZ; NUNES, 2022. Submitted.). Esse processo é realizado em ciclos de monitoração e é composto por algoritmos com fundamentos estatísticos para buscar que, ao final de cada ciclo de monitoração, a amostra coletada seja representativa da população. Durante os ciclos de monitoração, a amostragem adaptativa decide dinamicamente quais traços manter e adapta a taxa de amostragem para reduzir a sobrecarga de monitoração a um nível aceitável quando o sistema de software alvo precisa de seus recursos para processamento regular.

Por fim, esta tese avança a pesquisa sobre a redução do impacto do monitoração do tempo de execução do software, abrindo caminho para cobrir mais desafios de monitoração não resolvidos no futuro. É necessário mais trabalho para desenvolver uma solução geral que forneça um monitoração eficaz com custos praticamente aceitáveis, mas nosso trabalho é um dos passos nessa direção.