JOHANNES KEPLER
UNIVERSITY LINZ

Method Profiling and Lock Contention Profiling on the Java Virtual Machine Level

Submitted by
Dipl.-Ing. Peter Hofer

Submitted at
Institute for System Software

Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems

Supervisor and First Examiner
o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Second Examiner
Univ.-Prof. Dipl.-Ing. Dr. Walter Binder

October 2016

Doctoral Thesis
to obtain the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Large-scale software systems are frequently distributed over a network of computers in different locations, commonly rely on virtualization, and are often used by thousands of users at the same time. *Application Performance Management (APM)* software is used to continuously monitor such software systems. It uses profiling approaches to collect real-world performance data that can be used to identify and understand performance problems. However, unlike profiling tools, APM software must incur only minimal runtime overhead while still collecting sufficiently detailed performance data.

In order to address these challenges for Java, we propose a Java virtual machine implementation with sophisticated built-in profiling mechanisms. By integrating with the virtual machine, these mechanisms can efficiently collect accurate performance data. First, we propose new approaches for *sampling-based method profiling,* which attribute execution time to method calls. Inside the virtual machine, our approaches can directly decode and manipulate call stacks and can control how threads are paused to capture their states. Second, we propose a novel approach for *lock contention profiling,* which reveals concurrency issues that cause threads to wait for each other. Our approach integrates with the low-level implementation of locking in the virtual machine to efficiently collect accurate data while avoiding side effects. Finally, we propose a new approach to quantify *effects of virtualization,* which can misrepresent the resource usage of threads.

We implemented our approaches in the *HotSpot Virtual Machine,* a production-quality, high-performance Java virtual machine implementation. We evaluate our approaches using benchmarks with non-trivial, real-world workloads and demonstrate their efficiency and the accuracy of the data which they collect.

Contents

Chapter 1

Introduction

Troubleshooting performance bottlenecks in large-scale Java software systems is difficult and requires sophisticated tools. We characterize the challenges which these tools face, discuss how these challenges are currently addressed, point out open issues, and outline our approaches and scientific contributions to overcome these issues.

1.1 Problem Statement

Software developers use profilers during the development and maintenance of applications in order to troubleshoot performance problems. A profiler observes the execution of an application and collects performance data, such as how much time is spent executing individual methods. The collected data is typically summarized and visualized in a form in which a developer can use it to understand the behavior of the application and to subsequently apply optimizations.

However, profiling large-scale software systems is challenging. These software systems are commonly used by hundreds or thousands of users at the same time. They consist of communicating components that are distributed in a network of computers, often in multiple locations and running on different hardware. The use of virtualization and cloud computing services can further add to their complexity. Under such conditions, software can exhibit performance characteristics that are very different from those that are revealed by profiling on a testing infrastructure, or even those on a single developer workstation.

A straightforward approach to this problem is to collect performance data on the production system itself. Application Performance Management (APM) software provides this functionality and is typically used to monitor distributed web applications, which are often implemented in Java. APM software observes individual transactions in the software system which, for example, start out in a user's web browser and are handled by a frontend service, which in turn calls several backend services. For that purpose, APM software continuously monitors each component of the system as well as the communication between the components. The collected performance data can be used to detect when performance problems occur and to diagnose them. APM software collects this data using profiling approaches and faces challenges such as:

Impact on Performance. Unlike profiling tools, APM software is used on production systems, where it must not impose a significant impact on the performance of the software system and should incur only minimal runtime overhead. For this purpose, APM software depends on the availability of efficient profiling capabilities in the monitored environment.

Accuracy of Collected Data. Despite the goal to minimize its impact on the software system's performance, the primary objective of APM software must be to collect sufficiently detailed data which can be used to reliably detect performance problems, to accurately determine a problem's cause, and to effectively remedy a problem.

Detection of Concurrency Issues. Large-scale software systems rely on multi-core hardware to process many transactions in parallel. However, concurrency issues can limit the system's achievable degree of parallelism and can leave the hardware underutilized. APM software should identify and locate such problems, especially because they are more likely to occur in production systems with more powerful hardware than that in testing infrastructures or on developer workstations.

Accounting for Virtualization Effects. Distributed web applications are commonly deployed using virtualization and cloud computing services, so they share their resources with other software. APM software should determine how virtualization affects application performance and also how it distorts performance measurements.

1.2 State of the Art

Given the challenges which we identified above, we examined the profiling approaches that APM software currently uses to collect performance data. This thesis focuses on profiling approaches which APM software can use to monitor Java software systems.

Such profiling approaches must either be provided by the Java Virtual Machine (Java VM) or require its support. APM software can use multiple profiling approaches for different purposes.

Instrumenting approaches insert code snippets into the application code to collect data. This enables them to collect arbitrary data, such as method calls, object allocations, or the values of variables. However, by modifying the application code, instrumentation can significantly change the performance characteristics of the application. For example, instrumenting a method to measure its execution time also contributes to its execution time, and affects short-running methods more than long-running methods. The changes to the code can also interfere with optimizations of the Java VM. For those reasons, instrumentation can have a significant impact on the application's performance and on the accuracy of the collected data, and this impact can be hard to predict. Efficiently collecting accurate data for the detection of concurrency issues is difficult for the same reasons. Furthermore, frequently used concurrency features of Java are implemented inside the Java VM and do not expose their internals to application code, so instrumentation is ill-suited to observe those features. Nevertheless, instrumentation is useful to capture all occurrences of an event in an application, such as the begin and the end of transactions that are observed by APM software.

Profiling approaches which interact with the Java VM typically use the *JVM Tool Interface (JVMTI)* to collect performance data. JVMTI offers functions to inspect aspects of the Java application such as its threads, its loaded classes, and the objects on the heap. It also offers functions that provide information about the state of the Java VM itself, which is typically not observable with instrumentation. One profiling approach that is straightforward to implement with JVMTI is sampling-based method profiling, which periodically takes samples of the currently executing methods to create a statistical execution profile. For that purpose, JVMTI offers ready-to-use functionality to capture the executing methods and their callers in all threads. JVMTI also provides functionality for detecting concurrency issues with intrinsic locks, which are a frequently used Java concurrency feature that is often implemented inside the Java VM. However, many of JVMTI's functions pause all application threads to report fully consistent states to the profiler. This incurs significant runtime overhead because JVMTI can typically pause threads only at specific code locations, and all threads must first reach such a location. Moreover, because JVMTI can capture states only in those locations, it conceals the code and method calls between them from the profiler. This strongly affects the accuracy of the collected performance data. Moreover, using specific functions of JVMTI can disable optimizations of the Java VM, which can also have a significant impact on overhead and accuracy.

Profiling tools have also been integrated entirely into a Java VM. The *Java Flight Recorder (JFR)* is a profiling tool that is part of the production-quality Oracle Java HotSpot Virtual Machine and records performance-related events in the application and in the VM itself. Because of its integration into the VM, JFR can record many events more accurately and with less overhead than an external profiler that uses JVMTI or instrumentation. This includes events for the detection of concurrency problems. JFR can also collect information that is not available with instrumentation or with JVMTI, such as VM-internal events and their details. Overall, JFR's approach for collecting performance data inside the Java VM is well-suited for APM purposes. However, JFR is designed as a stand-alone, self-contained profiling tool with no officially supported interfaces. Therefore, APM software cannot make use of JFR to collect data.

Accounting for the impact of virtualization is generally not done on the application level or on the Java VM level. Current approaches which quantify virtualization effects do so for the entire system under virtualization or for its individual virtual processors, which does not show which threads or transactions of an application are affected by virtualization effects.

1.3 Novel Solutions

In order to address the challenges of APM software and the shortcomings of current approaches which we identified above, we propose a production-quality Java virtual machine implementation with sophisticated built-in profiling mechanisms that are designed to be harnessed by external profiling tools such as APM software. These mechanisms can efficiently collect accurate performance data by being part of the virtual machine: they have exact knowledge of the virtual machine's implementation, can influence its execution when that is beneficial, and have direct access to exhaustive metadata about the application and about runtime artifacts such as generated machine code.

In this thesis, we advance the state of the art with novel approaches for the following forms of profiling:

Sampling-based method profiling is an essential technique for attributing the execution time of an application to the executed method calls. We propose a set of novel approaches which are built into the Java VM where they can directly decode and manipulate call stacks, can access details about optimized code, and can control how threads are paused for sampling. Our approaches are more efficient than current approaches and can be equally or more accurate.

Lock contention profiling reveals concurrency issues which cause threads to wait for each other because they try to acquire locks at the same time. We propose a novel approach which integrates with the low-level implementation of locking inside the Java VM, where all details of a contention are readily available, including which thread holds the lock and therefore causes the contention. Inside the Java VM, our approach efficiently collects accurate information, refrains from collecting this information unless contention actually occurs, and avoids causing additional contention and interference with optimizations.

Virtualization is provided by separate virtualization software and is independent of the Java VM. This virtualization software is generally not accessible for profiling purposes, especially with cloud computing services. Therefore, to **measure virtualization effects,** we propose a novel sampling approach which only requires access to performance counters that are commonly available in virtualized systems. Our approach efficiently accounts for virtualization effects because accessing these counters is typically inexpensive.

We implemented our approaches in the *HotSpot* Virtual Machine in order to evaluate them and to demonstrate their advantages over current approaches. HotSpot is a production-quality, high-performance Java Virtual Machine implementation. It is part of the widely used Oracle Java Runtime Environment (JRE) and Oracle Java Development Kit (JDK).

1.4 Scientific Contributions

The novel scientific contributions of this thesis are the following:

Stack Fragment Sampling is a novel sampling-based method profiling approach which minimizes pause times for threads and eliminates redundant samples of unchanged execution states. With support from the operating system, it interrupts an application thread only briefly to copy a fragment of its stack to a buffer and then immediately resumes the thread's execution. A background thread asynchronously decodes the captured stack fragments to stack traces. We published this contribution in [Hofer14a, Hofer14b].

Partial Safepoints and Incremental Stack Tracing constitute a second novel approach for efficient sampling-based method profiling. Unlike Stack Fragment Sampling, this approach does not require support from the operating system. Partial safepoints reduce pause times for sampling by targeting only those threads which are currently running. Incremental stack tracing constructs stack traces lazily instead of walking the entire stack for each sample, examining unchanged stack frames only once and sharing the collected data between multiple stack traces. We published this contribution in [Hofer15b].

Our **Lock Contention Tracing and Analysis** approach efficiently records lock contention events in the application. Unlike common methods, our novel approach observes not only when a thread is blocked on a lock, but also which other thread blocked it by holding the lock, and records both their stack traces. We further present a versatile analysis tool which enables users to identify locking bottlenecks and their characteristics in an effective way. We published this contribution in [Hofer15a, Hofer16].

Finally, our novel approach to **Virtualization Steal Time Accounting** is a simple, yet effective sampling-based technique to estimate the effect of the suspension of virtual processors on the reported resource usage of application threads. The technique requires no changes to the Java VM, to the operating system, or to the virtualization software. We published this contribution in [Hofer15c].

1.5 Project Context

The research described in this thesis was done at the *Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems* and was partly funded by *Dynatrace Austria*. The laboratory was established in 2013 at the Johannes Kepler University Linz to conduct application-oriented research together with industry partners.

Also in cooperation with Dynatrace, Lengauer, Bitto et al. published an efficient approach for tracking object allocations and garbage collections to find memory-related performance problems in Java applications [Lengauer15, Lengauer16]. Lengauer further proposed an approach to automatically tune garbage collector parameters to the allocation behavior of a Java application to increase its performance [Lengauer14].

Working with industry partner *KEBA*, Lettner et al. analyzed the process of modeling features of large-scale industrial software systems and proposed to use multiple models for different purposes and for multiple levels of granularity [Lettner15]. Angerer et al. proposed techniques to determine the potential impact of code changes on individual configurations of features [Angerer15].

In cooperation with *Primetals Technologies*, Vierhauser, Rabiser et al. devised the *ReMinds* framework, a flexible high-level framework for monitoring systems of systems at runtime to ensure that their components meet pre-defined requirements [Vierhauser16]. They further proposed a model-based approach for describing requirements and relating them with architectural scopes and with events that occur at runtime [Vierhauser15].

Before the laboratory was established, the author has also been part of a long-running research cooperation between the Institute for System Software at the Johannes Kepler University Linz and Oracle Labs (formerly Sun Microsystems). The focus of this cooperation has been research on Java virtual machines, especially on HotSpot.

Numerous researchers who participated in this cooperation have worked on the *Client Compiler,* which is one of the two just-in-time (JIT) compilers of the Java HotSpot Virtual Machine. Mössenböck devised an intermediate representation in static single assigment form [Mössenböck00]. Mössenböck, Pfeiffer and Wimmer introduced a linear scan register allocator [Mössenböck02, Wimmer05]. Kotzmann extended the Client Compiler with object escape analysis [Kotzmann05]. Würthinger added an algorithm for the elimination of array bounds checks [Würthinger09].

More recently, Würthinger et al. devised the *Graal Virtual Machine,* a modification of the Java HotSpot Virtual Machine [Würthinger13, Graal15]. It includes the *Graal Compiler,* a JIT compiler for Java which is itself written in Java, and *Truffle,* a framework for implementing programming language interpreters. Such interpreters work on abstract syntax trees and benefit from *self-optimization* through partial evaluation and dynamic compilation performed by Truffle and the Graal Compiler. Truffle interpreters have been implemented for JavaScript, Ruby, C, and other languages. Recent research on Truffle has focused on interoperability between language implementations [Grimmer15].

1.6 Structure of this Thesis

This thesis is organized as follows: Chapter 2 describes the Java programming language and the HotSpot Java Virtual Machine, in which we integrated our proposed profiling approaches. Chapter 3 gives an overview of sampling-based method profiling in general and characterizes current implementations for Java. It then describes our approaches, which are *Stack Fragment Sampling* (in Section 3.2) and *Incremental Stack Tracing with Partial Safepoints* (in Section 3.3), followed by an evaluation of the two approaches and a discussion of related work. Chapter 4 characterizes locks in Java and describes and evaluates our approach for recording and analyzing lock contention events in HotSpot, followed by a discussion of related research and tools. Chapter 5 describes and evaluates our steal time accounting approach for virtualized environments and examines related work. Finally, Chapter 6 summarizes our contributions and concludes this thesis.

Chapter 2

Background

This chapter characterizes the Java programming language and its implementation in the HotSpot Java Virtual Machine, in which we integrated our profiling approaches in order to evaluate them.

2.1 Java

This thesis focuses on profiling approaches for *Java* applications. Java is a general-purpose, object-oriented, high-level programming language that was developed by Sun Microsystems and is currently maintained by Oracle Corporation. It was designed with simplicity, portability, safe execution, and security in mind [Gosling15]. Java has well-defined, hardware-independent data types, enforces automatic memory management, and does not permit potentially unsafe memory accesses, all of which prevents many types of errors and security vulnerabilities. Support for concurrent programming is built into the language. The Java class library provides a broad range of functionality such as collection types, input and output facilities, multi-threading utilities, networking and security support, database connectors, and user interface toolkits.

The source code of Java applications is organized in classes, which are compiled to *Java bytecode* ahead of execution. Java bytecode is the instruction set that can be executed by a Java Virtual Machine, which is an abstract machine that is not bound to a specific type of hardware or operating system [Lindholm15]. An *implementation* of the Java VM, such as the HotSpot VM that is introduced later in this chapter, takes care of specific aspects of the Java application's execution on a particular hardware architecture and on top of a specific operating system.

2.1.1 Native Code

Some Java applications need to call native functions of the operating system or code that is implemented in a different programming language. Profiling approaches must be able to handle such foreign calls. In Java, these calls are performed using the Java Native Interface (JNI), which allows Java code to interact with native code in other languages, typically in C and C++ [JNI15]. Moreover, JNI enables native code to call Java methods and to create and access Java objects. Typically, parts of the Java class library itself are implemented in native code, for example, file operations and network communication. Native code is executed in the same process as the virtual machine, but unlike Java code, it is generally non-transparent to the Java VM as well as to Java profiling tools.

2.1.2 JVM Tool Interface

As described in Section 1.2, Java profiling approaches commonly use the JVM Tool Interface (JVMTI) to interact with the Java VM and to collect performance data. JVMTI is a native programming interface that allows debuggers, profilers, and similar tools to interact with the Java VM and with the application running on top of it [JVMTI13]. Clients of JVMTI are called *agents* and run in the same process as the Java VM. Agents can invoke JVMTI functions to inspect aspects of the Java application such as its threads, its loaded classes, and the objects on its heap. An agent can also manipulate the application, for example by modifying the bytecode of classes or by suspending individual threads. Moreover, an agent can register callbacks to receive notifications about events in the application and in the VM, for example when classes are loaded or when threads are launched or terminated. JVMTI itself uses JNI to manage references to Java objects in native code.

The individual functions of JVMTI incur different overheads that depend on a function's implementation in a specific Java VM. Some functions may pause all application threads to safely capture a consistent state. Other functionality may temporarily or permanently disable optimizations of the Java VM, which can significantly affect the application's performance. When collecting performance data, this can also affect the accuracy of the collected data.

2.1.3 Other Languages on the Java VM

The Java VM also supports other programming languages besides Java. In this thesis, we also use software written in *Scala* and *Jython* for the evaluation of our approaches. Scala is a programming language which aims at unifying object-oriented and functional programming and has been designed to interoperate seamlessly with Java [Odersky04]. Jython is an implementation of the dynamic programming language Python for the Java VM [Juneau10].

Language implementations for the Java VM are commonly compiled to Java bytecode, either ahead of time or at runtime. Language constructs that have no equivalent in Java bytecode are typically compiled by generating helper methods and classes for them. This helper code becomes visible in Java profilers, debuggers and similar tools and can be difficult to comprehend for developers. Moreover, helper code can be less efficient than direct support for a language construct. However, there have been efforts to extend the Java VM to better support languages other than Java [Rose08].

2.2 The Java HotSpot Virtual Machine

The *Java HotSpot Virtual Machine,* or *HotSpot*, is a production-quality, high-performance implementation of the Java Virtual Machine [HotSpot15]. It is developed as part of the *OpenJDK* project, an open source implementation of the Java Development Kit (JDK), which also includes an implementation of the Java class library. OpenJDK is primarily developed by Oracle and is also the basis for Oracle's Java distributions. We used HotSpot as the foundation into which we integrated our profiling approaches in order to evaluate them.

Figure 2.1 shows a high-level overview of the architecture of HotSpot, which is described in the following.

Class Loader, Linker and System Dictionary. The class loader parses the bytecode of Java class files. The linker then resolves references to other classes as necessary and verifies that the bytecode and other data are well-formed and meet all the constraints of the specification. The system dictionary stores all loaded classes, their fields, their methods, and their bytecode instructions. Java applications can also provide custom class loaders which supply classes to the HotSpot class loader.

Interpreter. HotSpot initially executes methods in its efficient bytecode interpreter, which consists of hand-written assembly code for each bytecode instruction that is assembled to machine code at startup [Griesemer99]. The interpreter collects profiling data for the methods it executes. When a method is executed frequently enough, it is compiled to machine code by one of HotSpot's just-in-time compilers.

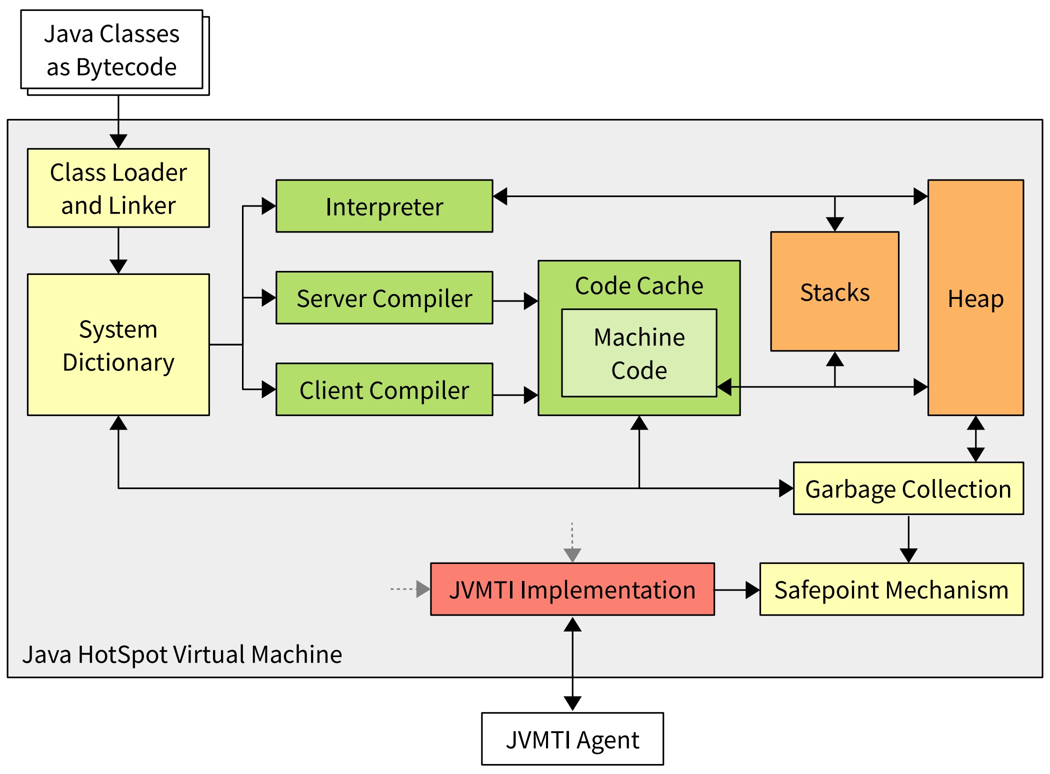


Figure 2.1: Architecture of HotSpot

Server Compiler. The server compiler is one of the two just-in-time (JIT) compilers in HotSpot which compile frequently executed methods to machine code. It is designed to generate highly efficient machine code by applying aggressive optimizations [Paleczny01]. Many of these optimizations are speculative, such as omitting code paths that are infrequently executed, or inlining a specific implementation of a polymorphic method. For these optimizations, the server compiler relies on the profiling information collected by the interpreter. The server compiler ensures that when a speculative optimization turns out to be invalid during execution, the code can be safely *deoptimized* so that its execution continues in the interpreter. The server compiler typically has long compilation times, but is intended for long-running server applications for which the initial compilation time is less relevant. For interactive applications, the long compilation times cause noticeable startup delays.

Client Compiler. The client compiler is the second JIT compiler in HotSpot and is designed for best startup performance. It applies fewer optimizations than the server compiler and incurs much shorter compilation times while still providing adequate performance [Kotzmann08]. The server compiler and the client compiler can be used together in *tiered compilation mode,* where a method is first compiled by the client compiler, and later by the server compiler. This approach achieves both a fast startup phase and high efficiency for long-running applications.

Code Cache. The code cache stores the compiled code generated by the JIT compilers as well as its metadata, which includes where methods have been inlined in the code and which assumptions have been made for speculative optimizations. The code cache also contains other machine code that is generated at runtime. This includes the assembled interpreter templates as well as *stub code* such as call wrappers for calls between compiled and interpreted code, call wrappers for native methods, compiler intrinsics, and other helper code.

Stacks. Each thread in HotSpot has its own stack. When a method is called in a thread, the called method creates its own *stack frame* on top of the thread's stack. The stack frame provides space for the method's local variables and also stores the return address, where execution continues in the caller when the method returns. The exact layout of a stack frame depends on its type. Interpreted methods, compiled methods, native code and stub code all use the same stack, but the layout of their frames is different. Stack walks in HotSpot support frames of different types and can determine which methods they belong to.

Heap and Garbage Collection. The heap is the memory area where Java objects are allocated. Because Java provides automatic memory management, a *garbage collector* scans the heap for objects that are no longer used by the application and reclaims their memory. The garbage collector uses the system dictionary to determine the layout of objects in order to find the pointers in them. It also uses the metadata of the code cache to locate pointers that are used by the currently executing code. The garbage collector further detects when a custom class loader of the Java application becomes unused. In that case, it unloads all classes in the system dictionary that were loaded by that class loader and also removes all compiled code for those classes from the code cache. This feature of HotSpot garbage collection can be used by plugin-based applications to completely unload plugin code.

HotSpot provides multiple garbage collector implementations, such as the Garbage-First (G1) collector that is intended for multi-processor machines with large memory [Detlefs04]. In most garbage collector implementations, some phases of garbage collection execute in parallel to the application. Other phases, such as moving objects that are still alive to avoid fragmentation, require the application's threads to be paused. For pausing threads, the garbage collectors use the virtual machine's *safepoint mechanism.*

Safepoint Mechanism. A *safepoint* is a state in which all application threads are paused in a well-defined state that allows the safe execution of certain operations. The safepoint mechanism is responsible for entering and leaving such a state. To accomplish this, the interpreter frequently checks whether a safepoint is currently pending, and the JIT compilers also insert checks for a pending safepoint at safe locations in the compiled code. When the safepoint mechanism indicates that a safepoint is pending, each application thread pauses itself as soon as it runs to its next safepoint check. Threads that are in a waiting state, such as those that are waiting to acquire a lock, are already paused in a safe state. The safepoint mechanism knows which threads are waiting because HotSpot tracks when threads enter a waiting state from Java code. Once all application threads are paused, the operation in question can execute, and as soon as it has finished, the safepoint mechanism resumes all paused threads.

JVMTI Implementation. The implementation of JVMTI in HotSpot allows agents to interact with HotSpot and with the application that it executes. For that purpose, numerous components of HotSpot implement specific functionality to provide agents with information, to allow agents to manipulate the application's state, or to report events to an agent. Many JVMTI functions use the safepoint mechanism to pause the application threads so that these functions can safely manipulate data structures or capture consistent states, especially when the functions operate on multiple threads.

Chapter 3

Sampling-based Method Profiling

This chapter introduces sampling-based method profiling and characterizes current approaches for profiling Java applications in HotSpot. We then describe our two novel approaches Stack Fragment Sampling and Incremental Stack Tracing with Partial Safepoints, followed by an evaluation of these approaches and a discussion of related work.

Method profilers measure the execution time of methods and generate a profile that shows those methods where the most time is spent. A performance engineer can use this profile to spot performance bottlenecks and to apply optimizations where they are most effective. For this reason, APM software and profiling tools commonly include a method profiler. Method profilers are also used to guide compiler optimizations, to determine test coverage, or to identify code that is never used.

In this chapter, we focus on *sampling-based method profilers.* We describe how current sampling-based method profilers for Java and specifically in HotSpot operate, and point out issues with their impact on application performance and their accuracy. Earlier in this thesis, we identified both of these issues as major challenges of APM software. In order to overcome these issues, we present two novel approaches to sampling-based method profiling, *Stack Fragment Sampling* and *Incremental Stack Tracing with Partial Safepoints.* We claim that these approaches are more efficient and more accurate than current approaches because they are integrated into the Java VM. In an evaluation of the two approaches, we back these claims.

3.1 Background

In this section, we describe how sampling-based method profilers collect data, which data they collect, and how this data is represented. We then describe implementations of such profilers for Java and specifically in HotSpot and analyze open issues.

3.1.1 Sampling and Instrumentation

There are two approaches to method profiling. *Instrumenting profilers* insert code snippets into methods to record calls. This yields a complete profile with all calls, but can cause substantial overhead. Instrumentation also introduces distortion because the additional code adds more relative overhead in short-running methods than in long-running methods, and because it influences compiler optimizations such as method inlining. Therefore, a profile from an instrumenting profiler can show behavior that is significantly different to that of an uninstrumented application.

Sampling profilers, on the other hand, periodically interrupt the application threads and record the currently executing methods, which requires no intrusive changes to the code. The resulting profile reflects the relative time spent in each observed method as the number of its samples out of all collected samples. The overhead from sampling can be easily adjusted by changing the sampling interval, which is the time between two subsequent samples. However, sampling can miss method calls between samples and therefore results in an approximate profile with only statistically significant information.

Our research focuses on non-intrusive profiling techniques with minimal overhead that are suitable for APM software, which is why we focused on sampling approaches in our work.

3.1.2 Calling Context Trees

"Flat" execution profiles break down an application's execution time to individual methods. However, a performance problem is often not confined to a single method. For example, when a flat profile indicates that an application spends too much time in a library method for sorting an array, it is unlikely that that library method itself is inefficient. Instead, the application likely sorts arrays more often than necessary and could be optimized by using more efficient data structures. Such performance problems are typically caused by very few locations in the application, but the sorting method could be called from hundreds of locations, and trying to examine or even optimize all of them is impractical. Therefore, prior research has demonstrated the importance of recording

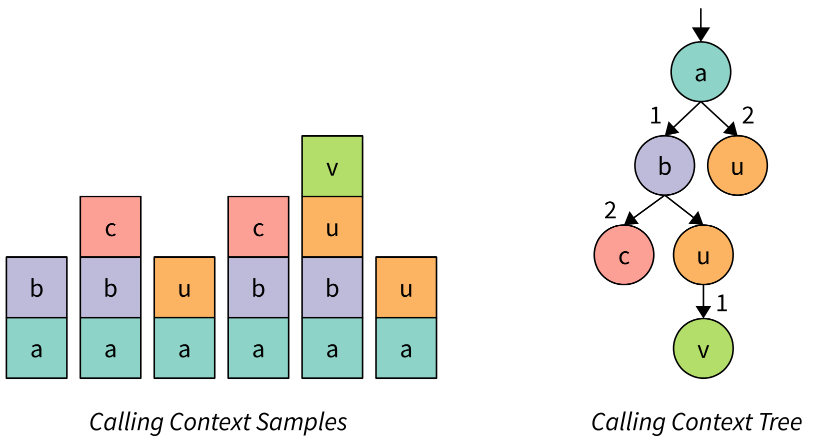
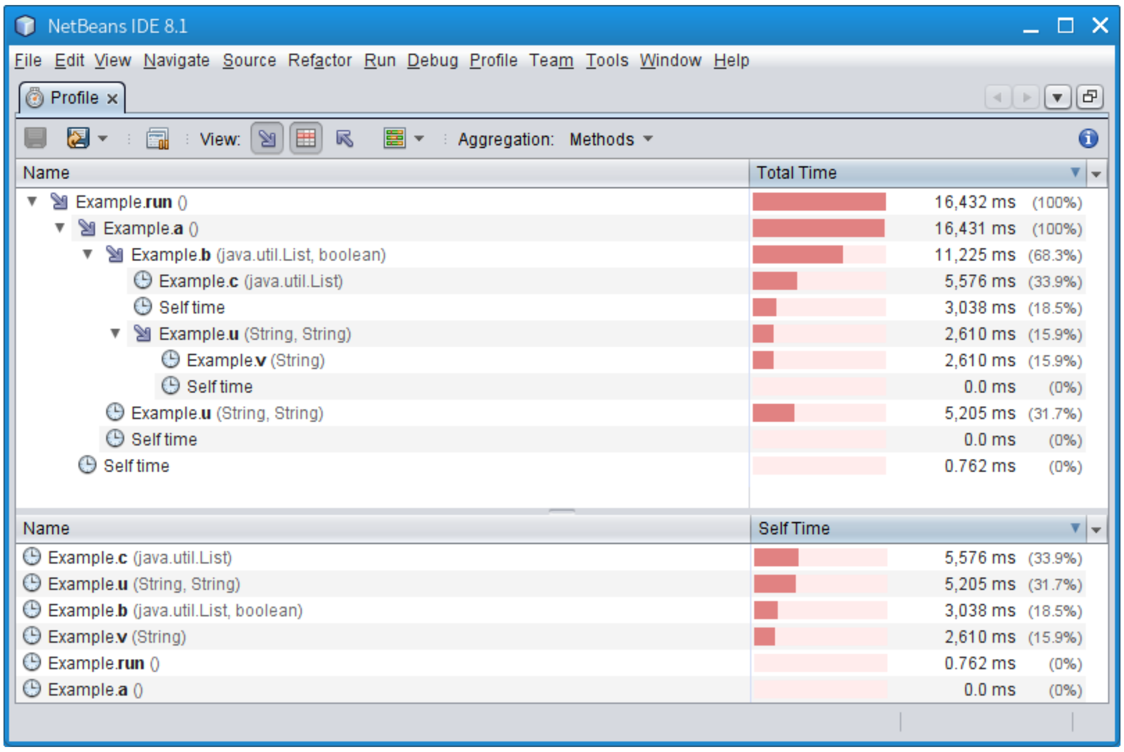


Figure 3.1: Calling contexts and calling context tree (CCT)

not just executing methods, but entire *calling contexts*, which are stack traces that include the executing method and its callers [Ammons97, Arnold05, Whaley00, Zhuang06].

Profilers typically merge the recorded calling contexts into a *calling context tree* (CCT, [Ammons97]). Figure 3.1 shows a series of calling contexts that a sampling profiler recorded and their resulting CCT. The calling contexts reflect calls from the entry method a to method b, from b to c, from a to u, from b to u, and from u to v. The CCT represents the called methods as tree nodes that have their callers as ancestor nodes, and merges multiple calls to the same method from the same callers into a single node, such as the two calls from a to u. Each edge from a caller to a callee can therefore represent multiple calls. The edges in a CCT are commonly assigned weights which reflect the execution time of the callee. In the depicted example, the weight of an edge is the number of samples in which the callee was the executing (top-most) method. Relative to the total number of captured samples (which is the sum of all edge weights), the edge weight therefore provides the fraction of the overall observed execution time that was spent directly in the callee. For example, u, when called from a, has an edge weight of 2, and the total number of samples is 6, so the execution of this calling context constitutes a third of the total observed execution time.

Figure 3.2 shows the profiler of the NetBeans IDE [NetBeans14] displaying a profile which is similar to the profile depicted in Figure 3.1. The hierarchical table at the center shows the CCT with the entry method *Example.run()* on the top level. The *Total Time* column shows the total times that were spent in each calling context. When a calling context has further callees, this total time includes the time that was spent in all its callees. In that case, an additional tree node labeled *Self time* shows the time that was spent executing only the method itself. In contrast to the CCT view, the table at the bottom shows a flat profile that displays only the self times of methods and provides no information about their callers.


NetBeans IDE 8.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Profile
View:
Aggregation: Methods

Name	Total Time		
▼ Example.**run** ()		16,432 ms	(100%)
▼ Example.**a** ()		16,431 ms	(100%)
▼ Example.**b** (java.util.List, boolean)		11,225 ms	(68.3%)
Example.**c** (java.util.List)		5,576 ms	(33.9%)
Self time		3,038 ms	(18.5%)
▼ Example.**u** (String, String)		2,610 ms	(15.9%)
Example.**v** (String)		2,610 ms	(15.9%)
Self time		0.0 ms	(0%)
Example.**u** (String, String)		5,205 ms	(31.7%)
Self time		0.0 ms	(0%)
Self time		0.762 ms	(0%)

Name	Self Time		
Example.**c** (java.util.List)		5,576 ms	(33.9%)
Example.**u** (String, String)		5,205 ms	(31.7%)
Example.**b** (java.util.List, boolean)		3,038 ms	(18.5%)
Example.**v** (String)		2,610 ms	(15.9%)
Example.**run** ()		0.762 ms	(0%)
Example.**a** ()		0.0 ms	(0%)

Figure 3.2: NetBeans profiler displaying a CCT

3.1.3 Running and Waiting Threads

The thread states in which samples are taken are relevant for the meaning of the profile and for the profiler's efficiency. At any time, an alive application thread can be in one of three basic states:

Running, when the thread is currently executing and consuming CPU time.

Ready, when the thread is currently not executing, but is ready to continue execution.

Waiting, when the thread is currently not ready to continue execution, for example because it was blocked while trying to acquire a lock, because it is waiting for a notification from another thread, or because it is sleeping.

A profile that is composed of samples of threads in state *running* reflects most accurately where CPU time is spent. It provides the most insight for a user who tries to understand why a CPU-bound task takes too much time, or to identify inefficient algorithms and data structures.

Profiles which further include samples of threads in state *waiting* can be useful to detect problems such as locking bottlenecks or slow network communication. However, such profiles can be more difficult to analyze for a user because threads in a typical application are expected to spend a significant amount of time waiting for incoming network connections, for user input, or for work to perform, which is simply idle time and not an indicator of inefficiency. Even after identifying contexts in which waiting actually causes delays in the execution of operations, it can be difficult to judge whether the time spent waiting in that specific context is unreasonably high. Moreover, because the time which a typical application spends waiting vastly outweighs the time spent running, including it can conceal performance problems in CPU-bound code.

A profiler can reduce its performance overhead by not taking samples of threads in states *ready* or *waiting* when such samples are not needed. This also avoids taking redundant samples of a thread when it has not been running since the last sample, which frequently occurs with short sampling intervals. However, it is the operating system which keeps track of the states of application threads. Most importantly, the operating system decides when application threads in the state *ready* are "scheduled in" and enter the state *running*, and when they are scheduled out again. Applications – including profilers – are mostly oblivious of those scheduling decisions and state transitions. Therefore, a profiler requires support from the operating system in order to be able to take samples only in a specific state.

3.1.4 Sampling with JVMTI in Safepoints

Sampling-based method profilers for Java are typically implemented as agents using the JVM Tool Interface (JVMTI), which we introduced in Section 2.1.2. JVMTI provides ready-to-use functionality for sampling-based method profiling, such as the function *GetThreadListStackTraces*, which pauses multiple threads, walks their stacks, and decodes the stack frames to an array of method identifiers. A straightforward sampling profiler can launch a separate thread with a sampling loop in which it calls this function to obtain stack traces for the application threads it is interested in, merges them into a CCT, and then sleeps for the sampling interval. However, the implementation of JVMTI in HotSpot and in other Java VMs suffers from problems that affect the efficiency and accuracy of sampling-based method profiling.

When an agent calls a JVMTI function to take stack traces, the function enters a task into the work queue of the VM thread. The VM thread can be considered the main thread of

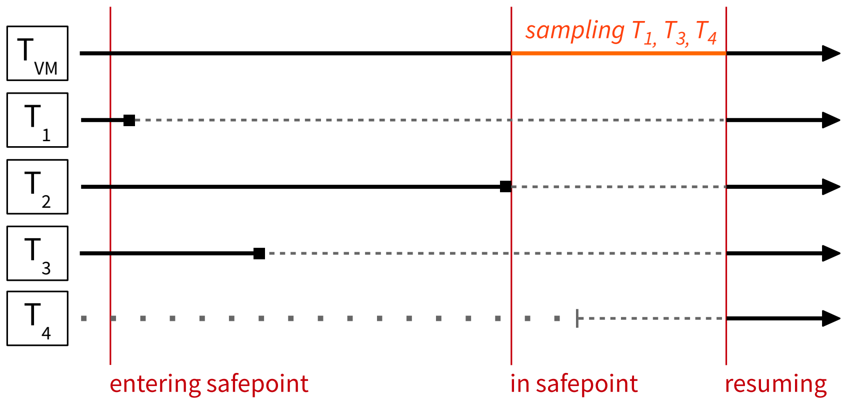


Figure 3.3: Sampling threads in a safepoint

HotSpot (which is different from the Java application's main thread). If higher-priority tasks in the queue are pending, the VM thread completes them first before it begins taking stack traces.

In order to safely walk the stacks of application threads, the JVMTI implementation of HotSpot uses the safepoint mechanism, which pauses *all* application threads. Figure 3.3 shows an example of how JVMTI uses safepoints to sample application threads: Thread T_{VM} is the VM thread, threads T_1, T_2 and T_3 are running application threads, and thread T_4 is an application thread which is waiting to receive data via a socket. In this example, the agent requests samples of the threads T_1, T_3 and T_4. The VM thread then indicates that a safepoint is pending. Soon after that, thread T_1 reaches its next safepoint check and enters a paused state (indicated by the square and the now dashed line), followed by thread T_3. Thread T_4 does not have to enter a safepoint because it is already blocked. At this point, thread T_2 delays the process, although no sample was requested for it, and threads T_1 and T_3 remain paused and unproductive. When thread T_2 finally pauses and hence, all threads have entered the safepoint, T_{VM} can walk the stacks of the three threads requested by the agent. In the meantime, thread T_4 becomes unblocked because its socket has received data, but instead of resuming its execution, a safepoint check ensures that it also enters a paused state. After the VM thread has finished taking the stack traces, it ends the safepoint and resumes all application threads.

The main problem with using safepoints for sampling is that they pause all application threads, including threads for which no samples have been requested. JVMTI also does not support skipping threads in states *ready* or *waiting* when performing stack walks. Therefore, a profiler cannot focus only on the currently running threads to reduce its overhead, and also cannot avoid taking redundant samples of threads that have not been running since the last sample was taken.[1] Although JVMTI includes the thread state in each sample, a profiler also cannot reliably filter samples in a state *waiting* after they are taken. The reason is that applications are frequently waiting in native functions, and because a Java VM generally cannot determine what happens outside of Java code, JVMTI reports a special thread state of *in native code* for such threads. This is common even in pure Java applications because the Java SDK uses native functions to implement network and filesystem operations. Therefore, a profiler that is trying to filter samples in state *waiting* can, at best, attempt to check them against a set of native functions that are well-known for waiting.

Optimizations performed by a JIT compiler can further affect the performance impact of JVMTI sampling, as well as its accuracy. By default, the compiler places safepoint checks at the exit points of methods and at the end of loop iterations. Although the overhead of safepoint checks is very low, it can become significant in hot loops. Safepoint checks also prevent certain kinds of optimizations because they must correspond to a specific location in a method's bytecode and thus constrain the order of instructions. Therefore, the compiler can decide to move safepoint checks out of loops to increase the code's performance and can even decide to eliminate safepoint checks in inlined methods. With fewer safepoint checks, it can take longer until all threads reach a safepoint check and pause, so the overhead of sampling increases. Moreover, when safepoint checks are eliminated, there are fewer locations where samples can be taken, which reduces the accuracy of the profile. Because safepoint checks are eliminated especially in hot code and its inlined callers, that code can then be particularly misrepresented in the profile, although hot code is of particular interest to a developer.

3.1.5 Sampling with AsyncGetCallTrace

HotSpot also contains the function *AsyncGetCallTrace* that enables profilers to take stack traces without using the safepoint mechanism. Unlike JVMTI, AsyncGetCallTrace is neither officially supported nor documented, and is not exposed through a public interface. AsyncGetCallTrace was developed for Sun Microsystem's Forté Performance Tools, a predecessor of Oracle Solaris Studio. Java Flight Recorder also uses a variant of AsyncGetCallTrace for sampling-based method profiling [Hirt13].

[1] A profiler could query the states of threads ahead of time and then decide whether to request samples of them. It could also keep track of their consumed CPU times to assess whether there has been activity in a thread after the last sample was taken. However, doing so would incur additional overhead and would be unreliable because threads can still change their states until the samples are actually taken. Moreover, excluding threads from sampling would only reduce the effort for stack walks while the safepoint mechanism would still pause all threads during the entire sampling operation.

AsyncGetCallTrace is designed to be used with signals, which are a form of inter-process communication that is defined in the POSIX standard [POSIX13]. Each process can register a custom function as its signal handler for a specific type of signal. When a process sends another process a signal, the operating system interrupts the execution of the receiving process and calls the appropriate signal handler for the type of signal. After the signal handler completes, execution continues in the code that was interrupted. With the POSIX threads extensions, signals can also be sent to a specific thread of a process.

A sampling profiler that uses AsyncGetCallTrace needs to be running as an agent in the HotSpot process and must register a signal handler for a specific type of signal, such as *SIGPROF*, which is intended for profiling purposes. The profiler then launches a sampling thread, but instead of using JVMTI to obtain stack traces, the sampling thread sends SIGPROF signals to threads of interest. Each receiving thread is interrupted and enters the profiler's signal handler, which calls AsyncGetCallTrace to walk the thread's stack and create a stack trace. The stack trace is then passed to the sampling thread and the signal handler returns to resume execution in the interrupted thread.

Unlike the safepoint mechanism, signals can interrupt a thread in an arbitrary state where performing certain actions is unsafe. For example, if the profiling signal interrupts a thread while it is allocating memory and the profiler's signal handler tries to allocate memory for the stack trace, it could corrupt the allocator's data structures. Therefore, signal handlers are only allowed to call a limited set of "asynchronous-safe" functions, which does not include functions for memory allocation or thread synchronization. This makes it difficult for the profiler's signal handler to allocate memory for the stack trace from AsyncGetCallTrace and to pass that stack trace to the sampling thread. We implemented our own sampling profiler with AsyncGetCallTrace for our evaluation later in this chapter. In this profiler, we used a custom memory allocator and a custom linked queue implementation, both of which rely only on atomic compare-and-set instructions to prevent corrupt or lost data.

The implementation of AsyncGetCallTrace itself is also significantly more complex than JVMTI's stack walk because it must handle many possibly unsafe states in which a thread can be interrupted. When AsyncGetCallTrace is called, it first checks whether the thread was executing Java code (as opposed to a native function). If so, it locates the top frame on the stack using the stack pointer and the instruction pointer from before the thread was interrupted. AsyncGetCallTrace then performs a series of checks to ensure that the top stack frame is in a consistent, walkable state. The top frame must have valid metadata from the JIT compiler or interpreter that includes the frame's size, which is used to locate the next frame below. When execution was interrupted in a method's prologue where the frame is still being set up, the frame size is considered unreliable and the stack walk is considered unsafe. Pointers that link frames are checked whether they point into the thread's stack, and return addresses are checked whether they point into valid code regions. While AsyncGetCallTrace walks the stack, these checks are repeated for each frame. If anything indicates that a stack walk could lead to a wrong stack trace or even to an illegal memory access, the stack walk is terminated without a result. Therefore, not every request for a stack trace is fulfilled, which reduces the accuracy of profiling with AsyncGetCallTrace.

The profiling signal can also interrupt a thread while it is executing (or waiting in) a native function that was called from Java code. As a compiler optimization, native code often does not establish valid linkage between frames, so the native stack cannot be walked in that case. However, when Java code calls native functions, HotSpot stores the location of the last Java frame (which called the native function) in the thread's thread-local storage area. AsyncGetCallTrace tries to retrieve this information and to start the stack walk from this frame.

AsyncGetCallTrace also cannot be used to reliably sample only running threads. However, a profiler may be able to use implementation details of the signal delivery mechanism to attempt to exclude threads in states *waiting* or *ready*. The POSIX standard specifies that a signal that is directed to a process (rather than to a specific thread) should be handled by a single, arbitrarily selected thread of that process. Some implementations, such as the current implementation in Linux, try to select a thread which is currently running to receive the signal [Linux15, *Version 4.6, kernel/signal.c*]. In that case, a profiler can send signals to the HotSpot process rather than to its individual threads and the signal handler will preferably be executed in a currently running thread. Nevertheless, this behavior is not guaranteed and may change between versions.

3.2 Stack Fragment Sampling

JVMTI and AsyncGetCallTrace require the application threads to remain paused while their stack traces are taken and cannot reliably restrict sampling to running threads in order to avoid redundant samples. Additionally, JVMTI can capture samples only at the locations of safepoint checks in the code, which affects its accuracy.

In order to address these shortcomings, we devised a novel approach which we named *Stack Fragment Sampling*. Our approach integrates with the operating system so that samples are captured only in running threads and at arbitrary locations in the code. This reduces the overhead by eliminating redundant samples and improves accuracy by avoiding the bias of sampling only in safepoint checks. Moreover, instead of pausing

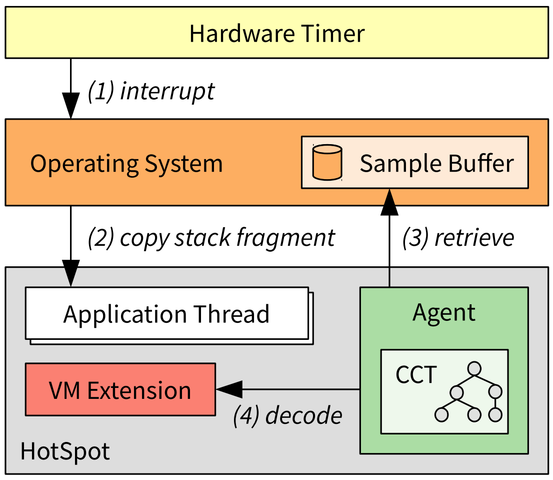


Figure 3.4: Stack Fragment Sampling components and their interaction

a thread until its stack trace has been taken, we only briefly copy a fragment of the thread's stack to a buffer and immediately resume the thread. The stack fragment is then asynchronously decoded to a stack trace. This minimizes the pause times for threads and further reduces the overhead. We described Stack Fragment Sampling in [Hofer14b] after presenting early results in [Hofer14a].

Figure 3.4 shows the components of our sampling technique and how they interact to take samples. We configure a hardware timer to periodically trigger interrupts while any thread of the Java application is currently running. When a timer interrupt occurs, (1) it is handled by the operating system, which (2) copies a fixed-size memory fragment from the top of the running thread's stack as well as the values of several processor registers into a sample buffer. The operating system then immediately resumes the interrupted thread. A profiling agent running within HotSpot (3) asynchronously retrieves the collected samples from the buffer and (4) passes each sample to a VM extension that decodes the stack fragments to a stack trace which the agent can then merge into its CCT.

3.2.1 Scheduling-Aware Sampling

A key aspect of Stack Fragment Sampling is that it is aware of scheduling and does not take redundant samples of threads that are not currently running. This requires support from the operating system, because it is the operating system that schedules which threads are running at a particular time, whereas applications – including profilers – are mostly oblivious of scheduling decisions.

In our implementation, we used *perf* to trigger samples and collect fixed-size stack fragments. perf is a subsystem of the Linux operating system kernel which profilers can use to collect profiling data when specific hardware or software events occur [perf15, Weaver15, Weaver13]. It integrates with the process scheduler and can restrict sampling of a thread to times when the thread is executing on a CPU. perf already supports copying a memory fragment from the top of a thread's stack as well as collecting the values of registers when taking a sample. In principle, the perf subsystem can even walk a thread's stack and construct its stack trace, but this feature was designed for native code and does not work for the more complex stacks of Java code in HotSpot, for which a stack walk requires additional metadata.

perf provides a single system call to set up sampling for an existing thread. In the parameters to this system call, the profiler specifies how samples should be triggered and what information should be collected for a sample. Samples can be triggered by a variety of hardware events such as CPU cycles or cache misses, as well as by software events such as context switches. When using an event as a trigger, a counter is incremented each time the event occurs. When the counter exceeds a threshold, perf takes a sample and resets the counter. The counter threshold is referred to as the sampling period. With hardware events, perf uses a counter of the CPU's hardware performance monitoring unit (PMU) that overflows when it reaches the sampling period and then triggers an interrupt. When perf has been set up successfully, it provides a file descriptor from which the collected samples can be read sequentially. This file can be mapped into the profiler's memory and then acts as a fixed-size ring buffer of the collected samples. The profiler must read the samples from the ring buffer in a timely manner to avoid that the buffer becomes full and new samples are dropped.

We use an agent running within HotSpot to interact with perf. Our agent is loaded at VM startup and uses JVMTI events to be notified when the Java application's main thread is launched. The agent then uses the perf system call to enable sampling for the main thread and, by enabling an inheritance option, for the entire hierarchy of application threads that the main thread launches. We configure perf to create one sample buffer for each CPU, and to use the CPU's reference cycle counter to trigger taking samples.[2] We set the sampling period to the number of cycles which the CPU can execute in the desired sampling time interval. If waiting periods should be included in the profile, we configure perf to take an additional sample each time when a thread is scheduled out.

[2]Unlike the regular cycle counter, the reference cycle counter is not affected when the operating system adjusts the CPU's frequency to reduce power usage.

3.2.2 Retrieving Collected Samples

After sampling has been enabled, our agent launches a separate thread that processes the collected samples. The thread waits until one of the perf buffers is filled to a "watermark" and then processes each of the samples that that buffer contains, as well as the samples in the the other buffers. When processing a sample, we first copy the sample's stack fragment from perf's sample buffer to a local buffer where it can be modified. Next, we scan the fragment for addresses that point into the live stack of the sampled thread and adjust the pointers to point into the fragment instead. We then call our VM extension and pass the modified stack fragment as well as the sample's stack pointer, its frame pointer (both adjusted to point into the fragment) and its instruction pointer. Our extension finally returns a stack trace to the agent, which merges it into its CCT.

In some situations, metadata that is required to decode stack fragments to stack traces can become unavailable, for example when HotSpot unloads classes or JIT-compiled methods. Stack fragments which were captured before such an event and which contain an unloaded method can then no longer be decoded to stack traces. Hence, our agent registers callbacks for the corresponding JVMTI unload events and processes all samples in all perf buffers before the metadata is discarded.

3.2.3 Decoding the Stack Fragments

Decoding stack fragments to stack traces requires knowledge of HotSpot's different types of stack frames and their layouts as well as specific metadata about JIT-compiled methods and other generated code. Because this knowledge is not available to an agent, we decided to implement decoding stack fragments directly in HotSpot and to make the functionality available to our agent via a JVMTI extension method which we named *GetStackTraceFromFragment*. Using JVMTI's extension mechanism has the advantage that a profiling agent can probe whether the VM supports the extension method and can fall back to a different sampling technique when it is not available.

We based the implementation of GetStackTraceFromFragment on that of AsyncGetCall-Trace because its code already handles some of the difficulties of walking stacks in an arbitrary state in which stack walks are potentially unsafe. To start a stack walk, we use the stack pointer recorded in the sample to locate the top frame. We then look up the instruction pointer in HotSpot's code cache to determine whether the thread was executing compiled Java code or other generated code. If so, we use the available metadata for that code to continue the stack walk, otherwise we try to use the frame pointer and frame linkage to find the top Java frame on the stack.

While walking the stack, we validate each frame before trying to advance to the next frame because the stack fragment can contain invalid data if the sample was taken in an unsafe state, for example, while the stack was being unwound during exception handling. If we would not handle such cases correctly, the resulting stack trace could be invalid, or worse, HotSpot could crash. Therefore, we always check whether any address which we inspect correctly points into the stack fragment or to valid code, and we further perform additional checks that are specific to the type of the frame.

Unlike AsyncGetCallTrace, we must be able to walk stacks asynchronously and therefore cannot access the internal state of HotSpot that existed at the time when the sample was taken. Therefore, we had to devise heuristics and implement changes in HotSpot to successfully decode most stacks, which we describe below.

Native Code

If a sample was taken while the thread was executing native code, locating the top Java frame to begin the stack walk is difficult.[3] In favor of efficiency, native code commonly does not establish a proper chain of frame pointers and does not provide additional debugging information which would allow walking its stack frames to locate the top Java frame. However, decoding samples in native code is vital even for profiling pure Java applications because the Java class library itself relies on native code to implement network communication and file operations. AsyncGetCallTrace is able to decode such samples by reading the location of the top Java frame from a thread-local data structure where HotSpot records it at the time of the native call. However, we decode samples asynchronously and can only use the data that was captured in the stack fragment.

Therefore, we adapted HotSpot to place an *on-stack frame anchor* at the base of each Java thread's stack. This anchor is similar to the thread-local data structure which we mentioned above, but the anchor is captured in the stack fragments. We modified the interpreter and the JIT compilers so that when Java code performs a native call, it records the location of the top Java frame in the anchor, and when the native call returns, it clears the anchor. For native calls from JIT-compiled Java code, we record the frame's stack pointer and instruction pointer. For native calls from the interpreter and for calls to certain VM-internal native methods, we also record the frame pointer.

When decoding a stack fragment, we first inspect the anchor that it contains. If it holds the location of a Java frame, we start the stack walk from that frame, otherwise we attempt the stack walk using the stack pointer, frame pointer and instruction pointer

[3]Java profilers typically only capture the Java stack trace including the call to the native method, but not the stack frames of native code.

captured in the sample. However, because stack fragments are captured from the top of the stack and the frame anchor is at an unknown location below the top, we must find the anchor first. Therefore, we place known "magic" values before and after the anchor on each Java stack and scan for them when decoding a stack fragment. Because these magic values could also occur on the stack by chance, we perform additional checks to ensure that an anchor's contents are valid.

Native code invoked from Java can again call Java code via the Java Native Interface. In such cases, the stack consists of alternating sequences of Java frames and native frames. The resulting stack trace should contain all Java frames on the stack and not just the topmost sequence of Java frames, which again requires finding the boundaries between Java code and the typically unwalkable native code. In this case however, we do not need multiple on-stack anchors: when native code calls a Java method, HotSpot already places an *entry frame* on the stack with a pointer to the last Java frame below the native caller. Using the pointers in the entry frames, the stack walk can reliably skip over sequences of native frames.

Stub Code

Another problem occurs when frames from so-called stub code are on top of the stack fragment. Stub code refers to snippets of machine code that HotSpot dynamically generates as helper code, as call wrappers, or for compiler intrinsics. Stub frames have no common layout and some do not even have a known frame size. If a fragment contains a stub frame with an unknown size, we scan the words below the stack pointer for an address that could be a return address into Java code. If we find such an address and additional checks reassure that it is at the boundary to a Java frame, we ignore the stub frame and start the stack walk from the Java frame.

Method Prologues

Samples can also be taken in a method's prologue, which is the method's entry code that creates its stack frame. The prologue pushes the caller's frame pointer on the stack, makes the current stack pointer the new frame pointer and changes the stack pointer to point to the end of the new frame. Before these instructions have completed, the frame is incomplete and examining it is unsafe. However, HotSpot records at which instruction the prologue of each JIT-compiled method ends. We can check whether the captured instruction pointer is within a method prologue, and if so, we use the captured stack pointer and frame pointer to determine if the caller's frame pointer has been pushed on the stack yet. If so, the word below the pushed frame pointer, or otherwise, the word on top of the stack, should be a return address to Java code. If this is the case, we have located the boundary to the frame below and we start the stack walk from that frame, acting as if the call had not happened yet.

Incomplete Stack Fragments

A regular stack walk continues until it encounters the initial entry frame at the stack's base, which indicates the end of the Java stack. With large stacks, however, the fixed-size fragments which we capture from the top of the stack can be missing the base part of the stack, and our stack walk can arrive at the end of the fragment before reaching the initial entry frame. Therefore, we verify that all memory accesses remain within the fragment's boundaries. When we reach the end of the stack fragment early, we return the frames that we decoded until then and set a flag to indicate to the agent that the stack trace is incomplete. Nevertheless, for samples in native code, an incomplete stack fragment can prevent a successful stack walk when the fragment is missing the on-stack anchor that is located at the stack's base.

Example

Figure 3.5 shows an example of a stack walk with GetStackTraceFromFragment. The captured Java stack fragment is on the figure's left-hand side, with the stack base at the bottom of the figure. The thread was executing native code when it was sampled, so its stack has native frames at its top. The right-hand side of the figure shows the code cache, where HotSpot stores its generated code together with metadata. The stack walk is performed in the following steps:

(1) We locate the on-stack anchor near the stack base by scanning for its magic constants. The anchor consists of a stack pointer, a frame pointer, and an instruction pointer. The sample was taken while the thread was executing native code, so the stack pointer and instruction pointer have been set and are non-zero.

(2) We follow the anchor's stack pointer to locate the top Java frame. (If the anchor's pointers were zero, we would instead use the captured register values from the sample to locate the top Java frame.) To determine the type of the top frame, we look up the instruction pointer from the anchor in the code cache (2b). The corresponding metadata entry tells us that the frame is from a call wrapper for the method *Object.wait()*, which is a Java method that is implemented in native code, so we add *Object.wait()* to the stack trace. Call wrappers are pieces of code that

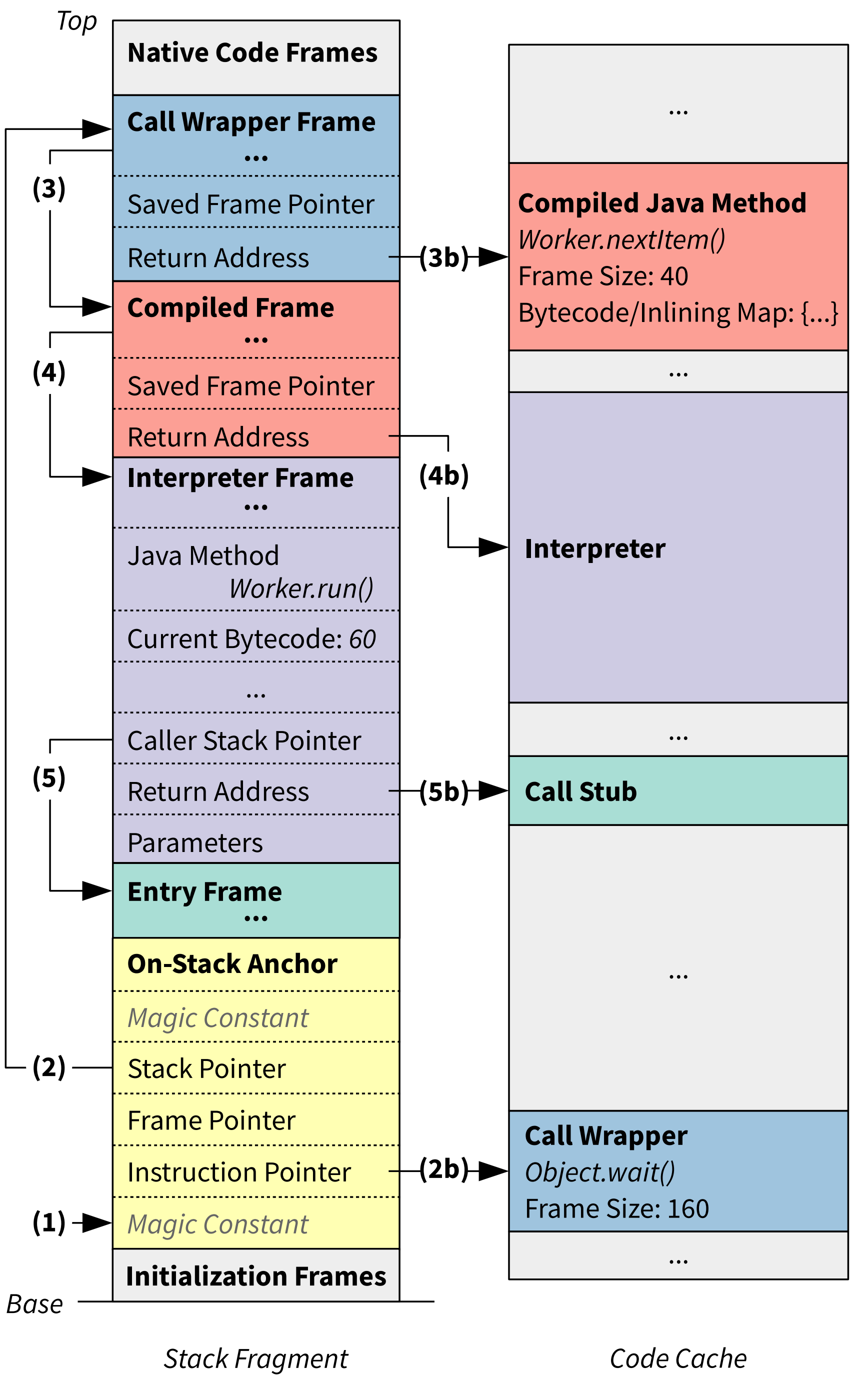


Figure 3.5: Example of decoding a stack fragment in native code

HotSpot generates for calls from Java code to native code and which take care of correctly passing the parameters and the return value. Frames of call wrappers have a fixed size, which HotSpot stores in the code cache.

(3) We use the call wrapper's frame size to advance to the next frame. In step (3b), we look up the return address of the call wrapper frame and find out that the frame belongs to the JIT-compiled method *Worker.nextItem()*. For each JIT-compiled method, HotSpot keeps metadata which maps instructions to locations in the Java bytecode and also stores which methods were inlined at which locations. We look up the return address from (3b) in that metadata and determine that it lies within code of *Queue.poll()*, which was inlined into *Worker.nextItem()*, and also determine the current bytecode positions in both methods. We add the two methods to the stack trace.

(4) We use the compiled frame's size which we know from its metadata in the code cache to advance to the next frame. In step (4b), we look up the return address of the compiled frame and find that it points to the bytecode interpreter. This does not tell us the executed Java method, but interpreter frames are very descriptive and from the frame itself, we can determine that the executed method is *Worker.run()* and that the current bytecode position is 60. We therefore add this method to the stack trace.

(5) Interpreter frames do not have a fixed size, but they store the caller's stack pointer in their frame, so we use this pointer to advance to the caller frame. We look up the return address from the interpreter frame in the code cache (5b) and find out that the caller frame was created by the so-called call stub. The call stub is a piece of generated code that performs the initial call to Java code in new Java threads. It also handles the case when native code calls a Java method, but in our example, it marks the end of the Java stack and thus the end of the stack walk.

Figure 3.6 shows the decoded stack trace for this example in text form. The actual data returned by GetStackTraceFromFragment consists of a result code that indicates if the stack walk walk was successful or why it failed, an array with the identifier of each Java method and its location in the bytecode, and flags that indicate whether the stack fragment is incomplete or which heuristics have been used.

3.2.4 Merging and Weighting Samples

After a stack fragment has been decoded to a stack trace, our agent merges it into its CCT. If the stack trace is incomplete because the stack fragment is missing frames from the

Method	Bytecode Position
`Object.wait()`	(native)
called from `Queue.poll()`	92
called from `Worker.nextItem()`	12
called from `Worker.run()`	60

Figure 3.6: Decoded stack trace for the example in Figure 3.5

base of the stack, we insert it below a special *unknown* node in the CCT that represents an unidentified sequence of frames. If the stack walk failed, we count the entire sample toward the unknown node itself.

We assign a weight to each sample which corresponds to the time between samples. When merging a sample's stack trace into the CCT, this weight contributes to the edge weight of the call to the sample's executing method. When threads are sampled only while they are running, the time intervals between samples – which are based on CPU cycles – are equal, so we assign a uniform weight to each sample. Therefore, the CCT shows a statistically accurate picture of the active methods while threads are running.

If periods of waiting in the application are sampled as well, weighting samples becomes more complex. Using the thread identifiers and timestamps that perf includes in its samples, we process the samples of each thread individually and in the order in which they were taken. We determine the weight of a sample as the time span until the next sample of the thread. For "regular" samples which are taken while the thread is running, this results in uniform weights that correspond to the sampling period. However, when a thread begins waiting and is scheduled out, we capture an additional sample of the thread's stack. When the thread takes a long time until it continues execution and therefore, further samples are taken, this sample of the thread in a waiting state is assigned an accordingly higher weight.

Figure 3.7 shows an example of samples of a thread which first executes on CPU_1, then spends time waiting, and finally continues execution on CPU_2. When our agent processes these samples, it first gathers all samples from the perf buffers of all CPUs and decodes their stack fragments to stack traces. We then process the thread's samples in the order of their timestamps and start with the first sample s_1. At this time, we cannot merge s_1 into our CCT because we do not know its weight yet, which is the time span until the next sample. We move on to the next sample, s_2. Using the timestamp of s_2 and of the previous sample s_1, we now compute the weight w_1, assign it to sample s_1, and merge s_1 into our CCT. We move on to s_3, which was triggered because the thread began waiting and was scheduled out. We compute w_2 and merge s_2 into our CCT. We then

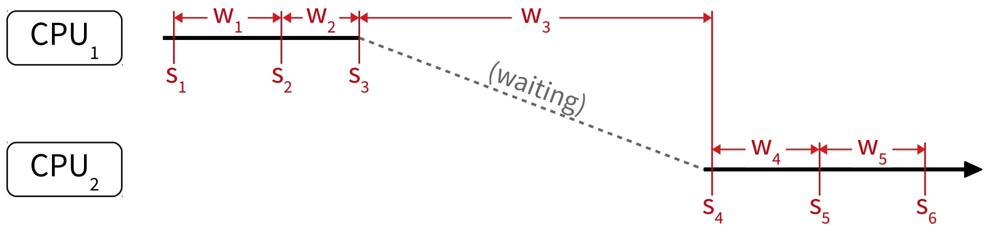


Figure 3.7: Example of processing samples of a thread with a period of waiting

move on to s_4, which was taken after the thread finished waiting and was scheduled in again. We compute w_3, which corresponds to the time span that the thread spent waiting. Accordingly, we assign w_3 to sample s_3, which contains the stack trace of the thread during the time it spent waiting, and merge it into our CCT. We move on to s_5, compute w_4 and merge s_4 into the CCT, and move on to s_6 and process s_5 the same way. We keep s_6 in memory until we later process the next sample when it becomes available.

3.3 Partial Safepoints and Incremental Stack Tracing

Stack Fragment Sampling is a highly efficient and accurate technique for profiling Java applications, as we will demonstrate later in its evaluation in Section 3.4.3. However, it relies on very specific capabilities of the operating system, which might not be available in a specific environment. Therefore, we devised *Partial Safepoints*, *Self-Sampling* and *Incremental Stack Tracing*, which form an alternative set of techniques that also significantly reduce the overhead of sampling Java applications, but are independent of operating systems and hardware. Partial safepoints and self-sampling reduce sampling pause times and can be used to target those threads that are actually running. Incremental stack tracing constructs stack traces lazily instead of walking the entire stack for each sample, examining unchanged stack frames only once and sharing the collected data between multiple stack traces. We presented these techniques in [Hofer15b] and describe them in this section. Part of this work has been done by David Gnedt and is described in his bachelor's thesis [Gnedt14].

3.3.1 Partial Safepoints

For a profiler, it is often sufficient to sample only a subset of the application's threads, especially those which are currently running and consuming CPU resources. Doing so reduces the profiler's runtime overhead. However, as we described in Section 3.1.4, sampling with JVMTI in HotSpot uses global safepoints which always pause *all* threads.

Taking samples only of running threads is not supported and even filtering the samples of non-running threads cannot be done reliably.

To target only running threads and to reduce the performance impact of sampling profilers, we implemented a variation of safepoints which we call *Partial Safepoints*. Partial safepoints require only a certain number of application threads to enter a safepoint state, and samples are taken only of those threads. We allow the profiling agent to choose the number of threads to sample. By using the number of CPUs in the system, the threads currently running on them ideally enter the partial safepoint first and are sampled, and no other threads are affected. With no scheduling data available, this is a best-effort approach. In practice, some of the system's CPUs might be executing threads of other processes. Also, the operating system might interrupt a running thread, and instead schedules another thread which then enters the partial safepoint in its place. In either case, however, a sample is taken of a thread which was in state *ready* when the profiler requested the samples, which we consider acceptable.

As soon as the intended number of threads has entered the partial safepoint, we can walk their stacks. Because some threads can enter a waiting state and block before reaching a safepoint check, we observe such thread state transitions to avoid a deadlock caused by waiting for more threads than can possibly enter the safepoint. While the stacks are walked, the safepoint must remain in effect. During that time, more threads than anticipated can enter the partial safepoint. Our implementation must consider which threads have entered the safepoint late and must finally resume all of them.

When a profiler is interested only in certain threads, it can restrict sampling with partial safepoints to a specific set of threads. We then wait for enough threads from that set to enter the partial safepoint and take samples only of those threads. The profiler can also request samples of all threads in the set. Thus, profilers that use a specific strategy to select threads for sampling can still benefit from partial safepoints, which do not affect all threads and remain in effect only for as long as necessary.

Waiting Threads

We extended our partial safepoints mechanism with an option to take samples also of waiting threads and of threads that are in native code. If this option is active, we determine the number of non-waiting application threads (i.e., running or ready) and the number of threads that are waiting or executing native code, and accordingly divide the number of requested samples between these two groups of threads. We randomly select which of those threads that are waiting or executing native code we sample. For example, if a profiling agent requests samples of four threads of an application that has 21 threads,

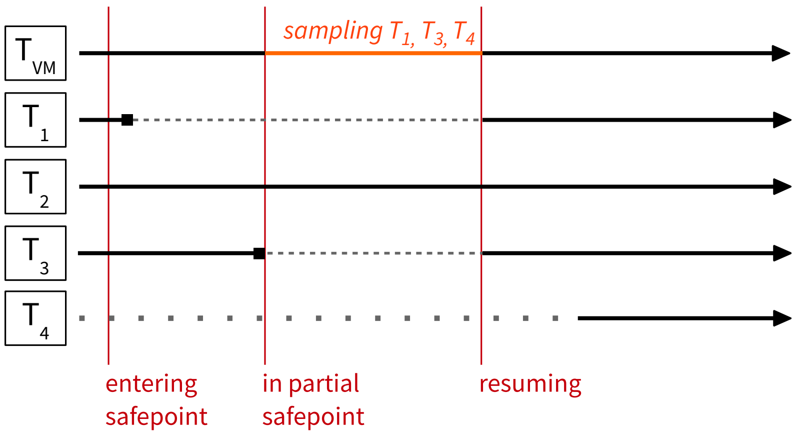


Figure 3.8: Sampling threads in a partial safepoint

out of which 15 threads are non-waiting and six threads are waiting, our approach will take samples of the first three threads that enter the safepoint, and one sample of another, randomly selected thread that is waiting. If the profiling agent restricts sampling to a specific set of threads, we determine the ratio of non-waiting to waiting threads for those threads and select the waiting threads to sample from the set.

Example

Figure 3.8 shows an example of how non-waiting and waiting threads are sampled with partial safepoints, using a similar scenario as the example in Section 3.1.4. In our example, a profiling agent has requested three samples. The VM thread T_{VM} determines that there are three non-waiting threads and one waiting thread. Therefore, it chooses to sample two non-waiting threads and the waiting thread T_4 and then initiates a safepoint. As soon as two threads, T_1 and T_3, have paused, it takes samples of T_1, T_3, and T_4, and then immediately resumes T_1 and T_3. By the time when thread T_2 reaches its next safepoint check and when T_4 becomes unblocked, the safepoint is no longer in effect, and their execution remains entirely unaffected by sampling.

3.3.2 Self-Sampling

With a straightforward implementation of partial safepoints, the safepoint remains in effect while the VM thread walks the stack of each thread. This is to ensure that the stack walks can be done safely. During that time, further threads can enter the partial safepoint. Although these threads will not be sampled, their execution is paused, and the VM must also keep track of them in order to resume them later, all of which causes unnecessary overhead.

To minimize the time the partial safepoint must remain in effect, we combined partial safepoints with *Self-Sampling.* When a thread enters a partial safepoint, it takes a ticket which tells it whether it is among the threads which should be sampled. If it is, the thread immediately walks its own stack. However, if the profiler restricted sampling to a set of threads, only threads in that set get a ticket and can sample themselves, and any other threads that enter the safepoint simply wait for it to end. When the last thread that should be sampled enters the partial safepoint, it notifies the VM thread, which then ends the safepoint so no further threads (which would not be sampled) can enter it. After a thread has completed sampling itself, it places the stack trace in a designated buffer and notifies the VM thread. The VM thread waits until all threads have provided their samples, and then resumes all sampled threads and returns the samples to the agent.

Blocked threads do not enter a safepoint, so they cannot sample themselves. Instead, the VM thread takes their samples, which increases the time the safepoint must remain in effect. However, the VM thread can take the samples while other threads are still running to their next safepoint check.

Figure 3.9 shows an example of a partial safepoint with self-sampling threads. As in Figure 3.8, the VM thread T_{VM} inspects the states of the application threads, decides to sample two running threads and the waiting thread T_4, and then initiates a safepoint. It then immediately begins to take a sample of the waiting thread T_4. Meanwhile, thread T_1 enters the partial safepoint and examines its ticket. Because it is the first of the two non-waiting threads that to be sampled, it walks its own stack, places the stack trace in the designated buffer and notifies T_{VM}. When T_3 enters the safepoint, it also examines its ticket and recognizes that it is the second and last of the two non-waiting threads to be sampled, so it notifies T_{VM}, which ends the safepoint. After T_3 has sampled itself and placed the stack trace in the buffer, it notifies T_{VM} again, which resumes T_1 and T_3 and returns the collected stack traces to the agent.

3.3.3 Incremental Stack Tracing

Self-sampling and partial safepoints reduce the time that an application must pause for sampling. To further lower the overall overhead, we looked at the cost of stack walks. In many cases, the stack frames from the stack base up to a certain stack depth remain unchanged for most of a thread's execution. Nevertheless, these unchanged frames are examined during every stack walk.

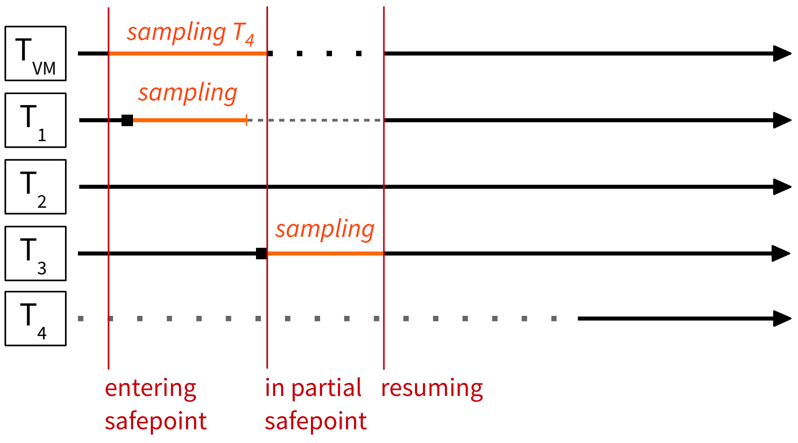


Figure 3.9: Self-sampling threads in a partial safepoint

Figure 3.10 shows an example of redundantly sampled frames, starting with a call to the method a. Method a calls b, which in turn calls c. While execution is in c, the profiler takes a sample. The stack walk visits the frames of the three active methods as well as all frames below a. When the profiler takes the next sample in v, c has returned, the frame of b has changed because b continued its execution, and two new frames from the calls to u and v are on the stack. Although the frame of a and all frames below it remained unchanged, the stack walk needlessly walks and decodes them again. When the profiler takes a third sample, the frames of u and v have disappeared and only the frame of b has changed, but the stack walk again visits all other frames as well.

To avoid redundant sampling of frames, stack walks could be limited to a certain number of frames below the frame of the executing method. However, the resulting incomplete stack traces would not be suitable to be correctly merged into a CCT, which has the entry method as its root. Therefore, we devised an approach that builds stack traces incrementally when methods return, and does not examine an unchanged stack frame more than once. We based our technique for incremental stack tracing on an approach that our research group developed for implementing continuations in a Java VM [Stadler09].

Data Structures

To share frame information between stack traces of multiple samples, we store the traces in a tree structure. Figure 3.11 shows what this tree looks like for the example from Figure 3.10. We maintain a linked list of *stack trace objects* for the stack traces that were taken, which is shown on the right-hand side of the figure. Stack trace objects are assigned numeric identifiers, which the profiling agent can use to keep track of the stack traces that it has requested. Each stack trace object has a pointer to a *frame object* that

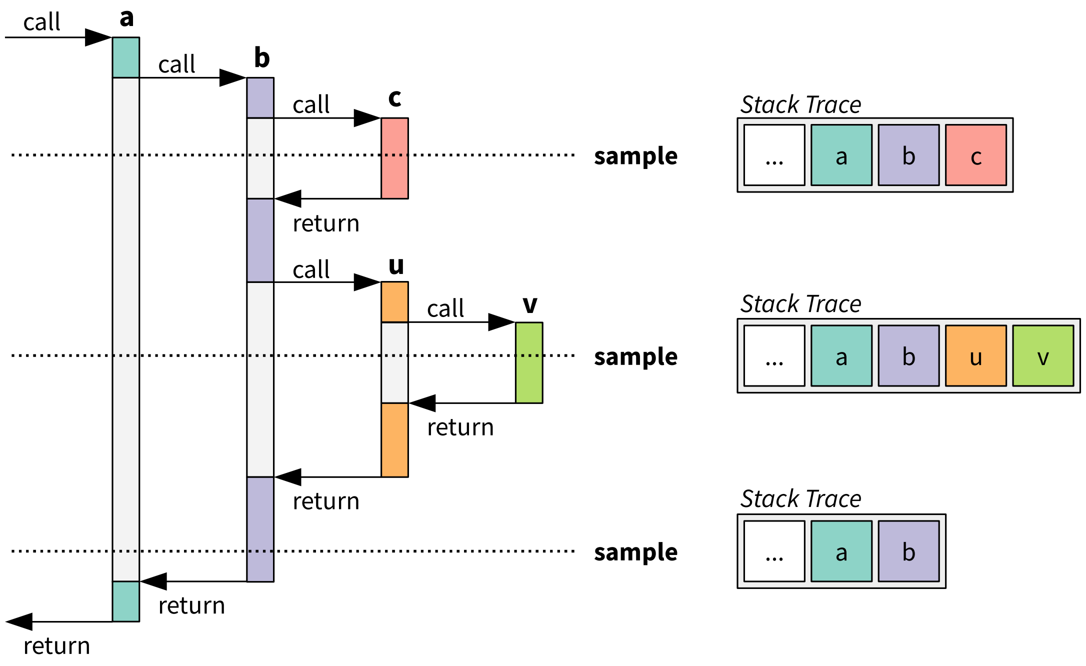


Figure 3.10: Samples with separate, complete stack traces

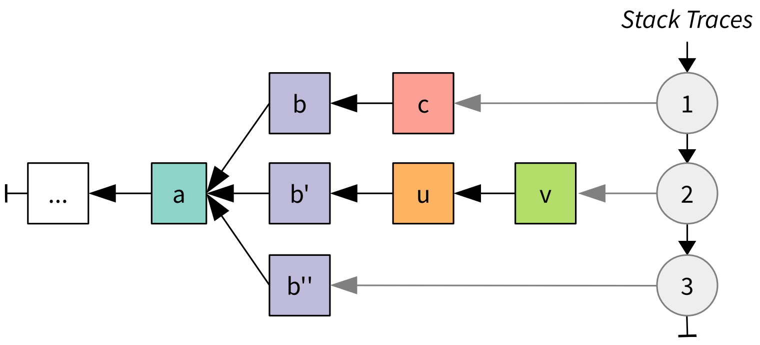


Figure 3.11: Stack traces in a tree with shared frame information

represents the stack frame which was on top of the stack when the stack trace was taken. For the first stack trace, which has the identifier 1, this is the frame object representing the frame for *c*, for stack trace 2 it is the frame object for *v*, and for stack trace 3 it is the frame object for *b*. Each frame object has a pointer to its caller frame object. The frame objects b, b' and b'' refer to the same invocation of *b*, but the duplication is necessary because the frame objects store different execution positions (i.e., bytecode indices) within the method. This information is useful to a profiler, for example, to distinguish between different call sites in a method.

Frame objects store the details of a captured stack frame in the following attributes:

Parent: the pointer to the caller's frame object.

Method: the identifier of the Java method that the stack frame belongs to.
BCI: the index of the current instruction within the Java bytecode of the method.

The following attributes of a frame object are not intended for the profiler, but are required for capturing frames and managing the tree of frame objects (see the next section).

Filled: a value that indicates if the frame object has been filled with valid data, or if it is an empty *skeleton frame object.*
Frame Address: the frame's exact location on the stack.
Saved Return Address: the original return address of the callee.

Capturing Frames

We maintain a list of stack trace objects for each thread. When we take a new stack trace, we first create a new stack trace object and insert it into the respective thread's list. We then create a new frame object for the top frame on the stack, which we call *top frame object (TFO).* We decode the top frame and fill the TFO with the determined method identifier and bytecode index. The *frame address* and the *saved return address* attributes are not required for the TFO. We set the TFO's *filled* attribute and link it with the stack trace object that we created earlier.

In a second step, we deal with the caller frame. The caller frame remains unchanged until the top frame's method returns, so we do not capture it immediately. Instead, we create a *skeleton frame object* for the caller frame that we can fill later, and make this skeleton object the parent of the TFO. We store the caller frame's address in the skeleton object's *frame address* attribute so we can match it to the frame later. To intercept when the top frame's method returns, we patch the top frame's return address on the stack with the address of a piece of trampoline code that we generate during HotSpot's startup phase. The original return address is stored in the skeleton object's *saved return address* attribute.

When the top method returns, it returns to our trampoline instead of to its caller, and the trampoline in turn calls our stack tracing code. In this code, we decode the caller's frame into its skeleton object, patch the caller's return address on the stack, and create another skeleton object for the caller's caller frame. Finally, we do the actual return by using the saved return address that we stored in the skeleton object before.

Figure 3.12(a) shows an example of capturing a new sample of a stack with the top method *c* and callers *b* and *a*. We create a frame object as TFO, decode the top frame, and fill the TFO with the method identifier of *c* and the current bytecode index, *42*. We then deal with the caller frame *b*. We create a skeleton object for the frame and make

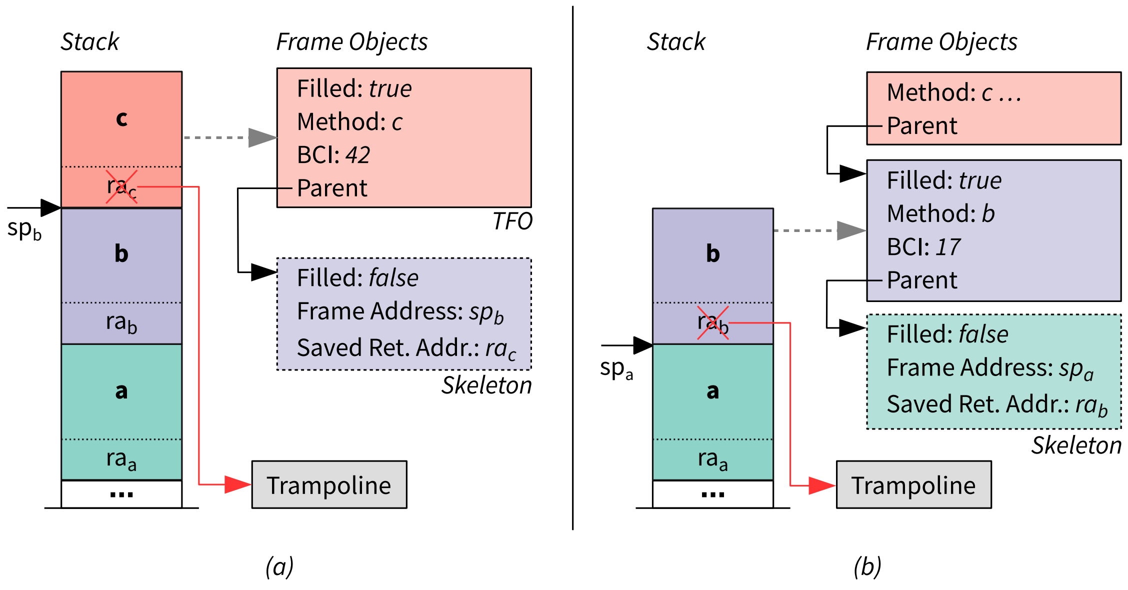


Figure 3.12: Capturing a frame (a) when taking a new sample, and (b) when intercepting a method return

it the parent of the TFO. We store the frame's address, sp_b, in the skeleton object so we can match the object to the frame later. To intercept when *c* returns, we patch its return address on the stack with the address of our trampoline code. We store the original return address ra_c in the skeleton object. Figure 3.12(b) shows how we capture the next frame when *c* returns. We intercept the return of *c* to *b* using our trampoline and decode the frame of *b* into its existing skeleton object. We then patch the return address of *b* on the stack, and create a new skeleton object for the frame of *a*, the caller of *b*. Finally, we do the actual return to *b* using the saved return address from the frame object of *b*.

To know which frame object must be filled when we intercept the return of a method, we maintain a thread-local pointer to the next skeleton object that needs to be filled, which we call *current skeleton object (CSO)*. We also use the CSO to implement sharing of frame objects between multiple stack traces. We distinguish the following situations:

Taking a sample. When we take a sample, we create a new TFO and fill it with the decoded top frame. We then examine the CSO as follows:

- If the CSO is not set yet, we create a new CSO and make it the parent of the TFO.
- If there already is a CSO, we check whether it refers to the frame of the TFO's caller by comparing their frame addresses. If they match, we make the CSO the parent of the TFO. Otherwise, we create a new CSO and insert it between the TFO and the former CSO.

Intercepting a return. When we intercept a method return, the CSO always refers to the frame object of the caller, so we decode the caller frame into the CSO. We then inspect the CSO's parent:

- If the CSO has no parent, we create a new CSO as the parent of the former CSO and associate it with the frame of the caller's caller.
- If the CSO's parent refers to the frame of the caller's caller, we make that parent the new CSO.
- If the CSO's parent refers to some other frame, we create a new CSO and insert it between the former CSO and its parent.

Figure 3.13 demonstrates how our technique incrementally builds stack traces for the example from Figure 3.10. Initially, the list of stack traces is empty and there is no CSO. The stack traces are then built in the following steps (for simplicity, we use the name of the methods to also refer to their frames).

(1) To take the first sample, we create a new TFO and decode the top frame c into it (the object that is filled in each step is highlighted in bold). Because there is no CSO yet, we create a skeleton object for the caller b (skeleton objects are indicated with a dashed frame). We make the new skeleton object the CSO and also make it the parent of the TFO. Finally, we patch the return address of c and save the original return address in the CSO.

(2) When c returns, the trampoline is executed, and we fill the CSO with the decoded stack frame of b. Because the CSO does not have a parent yet, we create a new skeleton object for a as parent. We then make that skeleton object the CSO, patch the return address of b and do the actual return from c to b.

(3) When we take the second sample, we decode the top frame v into a new TFO. We then check whether the CSO corresponds to the caller frame u. Since the CSO actually corresponds to a, we create a new CSO for u and insert it between a and v. Finally, we patch the return address of v.

(4) We intercept the return from v to u and fill the CSO with the decoded frame of u. Because the CSO's parent, which is a, does not match u's caller, which is b, we create a new CSO b' for b and insert it between a and u. We finally patch the return address of u and do the actual return from v to u.

(5) We intercept the return from u to b and fill the CSO with the decoded frame b. Because the CSO's parent, which is a, now corresponds to b's caller, we make the parent the CSO and do not need to create a new one. We also need not patch the

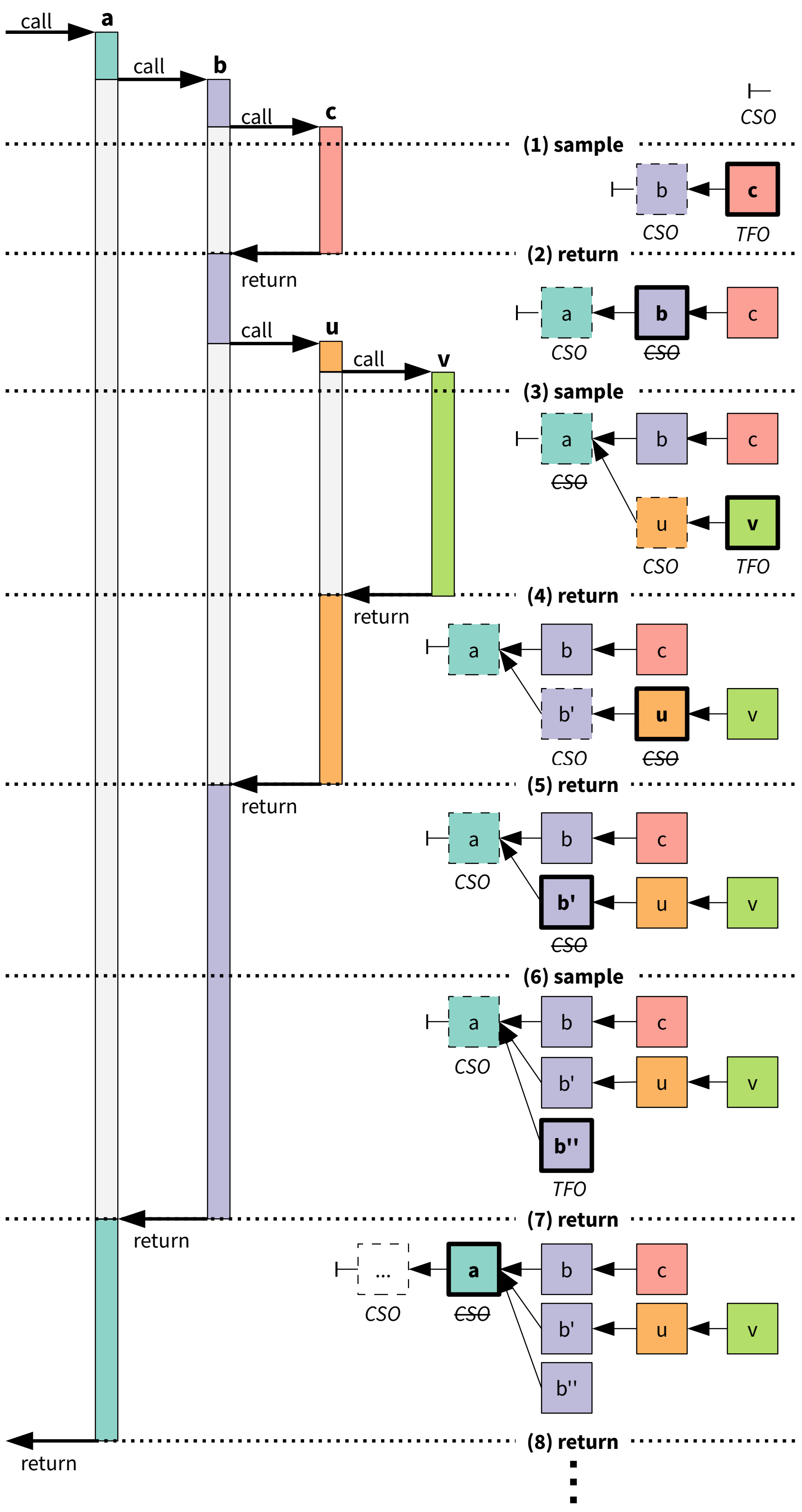


Figure 3.13: Incremental construction of stack traces

return address of *b* because it was already patched in step (2), and do the actual return from *u* to *b*.

(6) When we take a third sample, we fill the top frame *b* into a new TFO denoted by b''. Because the CSO corresponds to the caller frame *a*, we make it the parent of the TFO. The return address of *b* is still patched and does not need to be modified.

(7) We intercept the return from *b* to *a* and fill the CSO with the decoded frame *a.* Since the CSO does not have a parent here, we create a new CSO for *a*'s caller. Because all three stack traces join at the frame object of *a,* they share this object and all further frame objects below. Thus, we examine their stack frames only once.

Interface

JVMTI offers a single operation that walks the stack of a thread and returns a complete stack trace. Our approach does not create a complete stack trace right away, but incrementally builds stack traces and requires the profiler to collect them later. Therefore, we provide two operations for using our technique:

sample. The profiler can use the *sample* operation to record a stack trace, or rather, to initiate recording it incrementally. It can specify a numeric identifier which is assigned to the stack trace. The identifiers of stack traces need not be unique, and a profiler could also simply assign timestamps to the stack traces it requests.

retrieve. The profiler can use the *retrieve* operation to collect all recorded stack traces for a set of threads. The stack traces are returned in a tree structure that is similar to the described internal representation. When stack traces are still incomplete, the operation examines the remaining frames on the stack, completes the tree and reverts the patched return addresses on the stack. The retrieve operation empties the tree of stack traces kept in the VM. It always enters a global safepoint, but due to its infrequent use, the introduced overhead is negligible.

We implemented these two operations as JVMTI extension methods, which has the advantage that a profiling agent can check whether the VM supports incremental stack tracing. Typically, an agent would periodically request samples by calling the *sample* method, and infrequently use the *retrieve* method to collect the tree of stack traces. It can then merge this tree into a CCT and update the CCT's edge weights accordingly. The agent must retrieve the samples of a thread before the thread exits, or otherwise the stack traces would be released together with the thread's resources. It can accomplish this by subscribing to the *ThreadEnd* event that JVMTI offers.

3.3.4 Implementation Aspects

When we implemented our techniques in the highly optimized HotSpot VM, we had to handle several cases where thread synchronization or taking a correct stack trace is not as straightforward as described in the previous sections.

Frame Types

HotSpot starts out by executing Java bytecode in an interpreter, but compiles frequently executed methods to machine code. Therefore, the stack can contain frames of both interpreted and compiled methods, which differ in their layout. Moreover, Java code can call native methods of the VM, which again use different types of frames. When walking stacks and particularly when patching return addresses, we must handle each type of frame differently.

Inlining

The compiler aggressively tries to inline the code of called methods, and attempts to also inline those methods that are called by the inlined callees. Therefore, a particular location in compiled code can actually lie within multiple inlined methods that share a single stack frame. The compiler stores information about inlined methods and their ranges within other methods as metadata. When filling the frame object of a compiled frame, we must read this metadata and create extra frame objects for the inlined methods.

Exceptions

When a method throws an exception which must be handled by a caller, the method does not return in the usual way, using the return address on the stack. Instead, the VM unwinds the stack and pops frames until it reaches a method which can handle the exception. We modified HotSpot's exception handling code to capture a frame before it is popped from the stack (again considering any inlined methods).

Deoptimization

Deoptimization occurs when a method was compiled under an assumption that turned out to be false at runtime. When deoptimization occurs, the stack frame of the compiled method is transformed into one or more interpreted frames, and execution is continued in the interpreter. During this transformation, patched return addresses are lost, so we altered the deoptimization code to preserve patched return addresses.

On-stack Replacement

A method can start out executing in the interpreter, but can then execute long enough so that it would profit from JIT compilation. HotSpot can decide to compile the executing method and then perform on-stack replacement by transforming its interpreted stack frame into a compiled frame and continuing the execution of the method in the compiled code. Because the resulting compiled frame can have a different location than the interpreted frame, we have to update our data structures when on-stack replacement is performed.

Safepoint Synchronization

The safepoint checks that HotSpot injects into application code simply read a dummy value from a specific page in memory that is called *polling page.* When no safepoint is pending, these reads are inexpensive. To enter a safepoint, the VM thread acquires the global *threads lock.* Holding this lock prevents threads from starting and exiting and blocks thread state transitions, such as when a thread resumes execution after waiting. Therefore, acquiring the lock ensures that the VM thread has a consistent view of the existing threads and their states. Next, HotSpot instructs the operating system to read-protect the polling page. This causes the safepoint checks to trigger page faults in each thread, and the fault handler then pauses the thread. HotSpot finally waits until all threads are paused or are in a safe state guarded by the threads lock.

For partial safepoints and self-sampling, we use a modified safepoint mechanism that waits only until enough threads have entered the safepoint, and then immediately unprotects the polling page again. However, when the profiler restricts sampling to a set of threads, other threads can also enter the safepoint during that time. Therefore, before read-protecting the polling page, we set a flag for each thread which indicates whether the thread should sample itself if it enters the safepoint. Threads which have their flag set then sample themselves in the fault handler, while the other threads simply wait for the safepoint to end. When including waiting threads for sampling, we acquire the threads lock before computing the ratio of waiting to non-waiting threads, so no threads can change their state at that point.

avrora	microcontroller grid simulation
fop	PDF generation from XML
h2	in-memory database
jython	pybench benchmark suite
luindex	document indexer
lusearch	search in document index
pmd	Java source code analyzer
sunflow	raytracing renderer
tomcat	local webserver
tradebeans	trading simulation
tradesoap	trading simulation with SOAP messages
xalan	HTML generation from XML
(batik)	vector graphics generation
(eclipse)	performance tests of the Eclipse IDE

Figure 3.14: The benchmarks of the DaCapo 9.12 "Bach" Benchmark Suite

3.4 Evaluation

We claim that our sampling-based method profiling approaches are more efficient and more accurate than current approaches. In order to back this claim, we evaluated their runtime overhead, their accuracy and other characteristics. We independently evaluated Stack Fragment Sampling in [Hofer14b], and Partial Safepoints with Self-Sampling and Incremental Stack Tracing in [Hofer15b]. In this section, we describe our evaluation methodology and present our results.

3.4.1 General Methodology

We examine the overhead and other aspects of our approaches with benchmarks. We use the publicly available benchmarks of the DaCapo 9.12 "Bach" benchmark suite and of the Scala Benchmarking Project 0.1.0-20120216. Both suites consist of open-source, real-world applications with pre-defined, non-trivial workloads. The DaCapo suite provides 14 benchmarks of Java applications [Blackburn06]. However, we were unable to use its *batik* and *eclipse* benchmarks because they do not run on OpenJDK 8, which we used throughout this thesis. We briefly describe the individual DaCapo benchmarks in Figure 3.14. The Scala Benchmarking Project suite consists of 12 benchmarks of Scala applications [Sewe11], which we describe in Figure 3.15.

actors	trading simulation with actors
apparat	optimizer for Adobe Flash files
factorie	probabilistic topic extraction
kiama	language processing
scalac	Scala compiler
scaladoc	Scala documentation tool
scalap	Scala class file decoder
scalariform	Scala source code formatter
scalatest	Scala testing toolkit
scalaxb	XML data binding
specs	behavior-driven design framework
tmt	topic model learning

Figure 3.15: The benchmarks of the Scala Benchmarking Project 0.1.0-20120216

The harnesses of both benchmark suites are launched with a parameter to select which specific benchmark to run, and with a parameter for the number of *iterations*. The harness then repeatedly executes that benchmark's constant workload for the specified number of iterations in the same Java VM instance and reports the execution time of each iteration. The harnesses support registering callbacks that are invoked at the start and end of an iteration, which we use to measure and record additional metrics for each iteration.

Benchmarks executing on HotSpot are affected by a warm-up phase during which classes are loaded, methods are compiled to machine code, and the heap and the garbage collector adjust to the application's memory allocation behavior. Only after this warm-up phase, a benchmark reaches a steady performance level that is representative for a longer use of an application. The duration of the warm-up phase depends on the benchmark and on the hardware. We execute a fixed number of warm-up iterations that is sufficient to reach a steady state of performance in all benchmarks on the used hardware, followed by a fixed number of steady-state iterations. We record execution times and other metrics for each iteration, but discard the execution times and metrics for the warm-up iterations.

The steady-state performance of an application further depends on factors that are determined during the warm-up phase, such as optimization decisions of HotSpot as well as the memory locations of machine code in the code cache and of long-living objects on the heap. To avoid biases from such factors, we run multiple *rounds* of each benchmark, each of which executes an entire sequence of warm-up and steady-state iterations in a new HotSpot instance.

We report the runtime overhead of our profiling approaches relative to executions with no profiling. For that purpose, we also execute multiple rounds of the benchmarks with an unmodified HotSpot build[4] and without an agent. We report the *median* execution time of all steady-state iterations of each benchmark *with profiling, normalized* to the median execution time *without profiling*. We further summarize the runtime overhead for multiple benchmarks by reporting *geometric means* over their normalized medians, which is an indicator for their overall runtime overhead that weights each benchmark equally [Citron06, Fleming86]. Moreover, we report typical deviations in our measurements with the first and third quartiles for medians, and with a 50% confidence interval for the geometric means.

3.4.2 Accuracy Analysis

Determining the accuracy of a profiler is challenging. Ideally, we would like to compare the CCT generated by the profiler with a perfectly accurate CCT of an execution under the exact same conditions. However, obtaining a perfectly accurate CCT would require a profiler that can record each method call and measure its exact execution time while not interfering with the execution of the profiled application at all. While instrumenting profilers can record each method call and its execution time in a CCT, the instrumentation slows down short-running methods more than long-running methods and interferes with optimizations of the JIT compiler, especially with inlining. Therefore, when using instrumentation to measure the execution times of methods, the results are distorted and not representative for the unaltered application.

Therefore, we decided to analyze the accuracy of a profiler by determining whether it consistently generates similar CCTs for the same benchmark, and by comparing the CCTs produced by different profilers for the same benchmark to test whether the profilers agree. To compare two CCTs with each other, we use the *degree of overlap* and *hot-edge coverage* metrics, which rely on the following equivalence rules between nodes and edges of a CCT [Zhuang06]:

1. The root nodes $n_1 \in CCT_1$ and $n_2 \in CCT_2$ are equivalent if they represent the same method.
2. Nodes $n_1 \in CCT_1$ and $n_2 \in CCT_2$ are equivalent if they represent the same method, and if their parent nodes are equivalent.
3. Edges e_1 in CCT_1 and e_2 in CCT_2 are equivalent if the start nodes of both edges are equivalent and if the end nodes of both edges are equivalent.

[4]We use the same HotSpot versions and compile both the modified and unmodified builds with the same toolchain.

4. When two nodes or two edges from different CCTs are equivalent, we consider them part of both CCTs (regardless of their respective edge weights).

Degree of Overlap

The degree of overlap assesses how many edges of two CCTs are equivalent and how close their edge weights are to each other. It has been used extensively in related research [D'Elia11, Feller98, Moret09, Zhuang06]. Figure 3.16 shows its formal definition, where $relweight(e, CCT)$ is the relative edge weight of e in CCT, i.e., the percentage of e's edge weight relative to the sum of all edge weights in CCT. The degree of overlap considers only edges that are part of both CCTs, and adds the smaller of the two relative edge weights to the result. The result ranges from 0, where the two CCTs have no equivalent edges, to 1, where CCT_1 and CCT_2 share all edges and have exactly matching edge weights.

$$overlap(CCT_1, CCT_2) = \sum_{e \in CCT_1 \cap CCT_2} min(relweight(e, CCT_1), relweight(e, CCT_2))$$

Figure 3.16: Formal definition of the degree of overlap between CCT_1 and CCT_2

Hot-Edge Coverage

While the degree of overlap reflects all edges of two CCTs and the differences in their exact edge weights, only the hottest edges of a CCT are relevant for identifying performance bottlenecks. The hot-edge coverage metric determines whether two CCTs identify a similar set of edges as hot according to a relative threshold and puts less emphasis on the exact edge weights. This metric was introduced in [Zhuang06] and also applied in [D'Elia11].

Figure 3.17 shows a formal definition of hot-edge coverage. $weight(e, CCT)$ is the absolute edge weight of e in CCT, and $hottest(CCT)$ is the maximum edge weight of any edge in CCT. $hotset(CCT, T)$ is the set of the hottest edges in CCT, where T is a threshold relative to the hottest edge that the weight of an edge must exceed to be included in the set. The hot-edge coverage, $hotcover(CCT_1, CCT_2, T)$, is the number of equivalent edges in the hot sets of CCT_1 and CCT_2 divided by the number of edges that are only in the hot set of CCT_2. Hence, the hot-edge coverage metric computes the percentage of hot edges of CCT_2 that are covered by the hot edges of CCT_1.

$$hotset(CCT,T) = \{e : e \in CCT, weight(e,CCT) \geq T \cdot hottest(CCT)\}$$

$$hotcover(CCT_1, CCT_2, T) = \frac{|hotset(CCT_1, T) \cap hotset(CCT_2, T)|}{|hotset(CCT_2, T)|}$$

Figure 3.17: Formal definition of the hot-edge coverage of CCT_2 by CCT_1

Unlike the degree of overlap, hot-edge coverage is not commutative, i.e. CCT_2 might cover the hot edges of CCT_1 better than vice versa. It is also vital to recognize that the threshold, for example $T = 0.6$, is relative to the single hottest edge. This is different from selecting the hottest 40% of edges, or the hottest edges that make up 40% or more of the sum of edge weights.

Analysis Methodology

For each profiler and benchmark, we merged the CCTs from all (non-warmup) iterations into a single CCT. [5] This merged CCT contains all edges that exist in any of the CCTs, with edge weights that are the arithmetic means of the relative edge weights in all CCTs. The merged CCT is therefore the *average CCT* over all CCTs from the individual iterations. We then used the average CCTs to analyze the consistency of the CCTs generated by a profiler for a specific benchmark, and to determine the agreement of different profilers over a CCT.

Consistency Analysis: The behavior of our benchmarks generally does not deviate significantly between iterations, so a profiler should also consistently produce similar CCTs for different iterations of a benchmark. We analyzed this consistency property by determining the degree of overlap between the CCT from each individual iteration and the average CCT, and by determining the hot-edge coverage of each individual CCT by the average CCT. This analysis also shows whether the average CCT is representative for the individual CCTs.

Profiler Agreement: Consistency between the CCTs of a profiler does not prove that these CCTs are accurate because a profiler can also consistently generate inaccurate CCTs. Therefore, we determined the degree of overlap and the hot-edge coverage between the

[5]Some of our benchmarks dynamically generate classes which are assigned different names in different iterations, such as call wrappers or web service handlers. We adapted our analysis tools to match identical generated methods when merging CCTs.

average CCTs from the different profilers for each benchmark. When profilers agree on a CCT, we take that as an indication that the CCT is accurate.

3.4.3 Stack Fragment Sampling

We implemented Stack Fragment Sampling in OpenJDK 8u5-b13 and evaluated our implementation with the DaCapo 9.12 benchmark suite and with the benchmarks of the Scala Benchmarking Project 0.1.0, as described in Section 3.4.1. We further included the SciMark 2.0 benchmark [Pozo04], which is a Java benchmark for scientific and numerical computing. [6]

We compared the runtime overheads and the generated CCTs of the following sampling approaches, for each of which we implemented a profiler that runs as an agent in the Java VM and constructs a CCT:

- Conventional sampling with JVMTI
- Sending signals to threads to take stack traces with AsyncGetCallTrace (AGCT)
- Stack Fragment Sampling (SFS)

We performed all tests on a system with a quad-core Intel Core i7-4770 processor with 16 GB of memory running openSUSE Linux 13.1. We disabled hyperthreading, turbo boost and dynamic frequency scaling to avoid their often irreproducible effects on the execution of an application. With the exception of vital system services, no other applications were running while the benchmarks were executed.

We executed 30 successive *iterations* of each benchmark with each profiler in a single HotSpot instance. To generate metrics and a CCT for individual iterations, our profilers track the start and the end of benchmark iterations. We discarded the data from the first 20 iterations to adequately compensate for HotSpot's startup phase on our system. We further executed multiple *rounds* of each benchmark (with 30 iterations each) with each profiler to ensure that the results are not biased by factors that are determined during HotSpot's startup phase, such as early optimization decisions.

Stack Fragment Size

For our SFS profiler, we first had to choose the size of the sampled stack fragments. The fragments should be large enough to include all frames on the Java stack as well as our

[6] We adapted SciMark 2.0 to use a constant workload instead of measuring operations per second over a fixed time period.

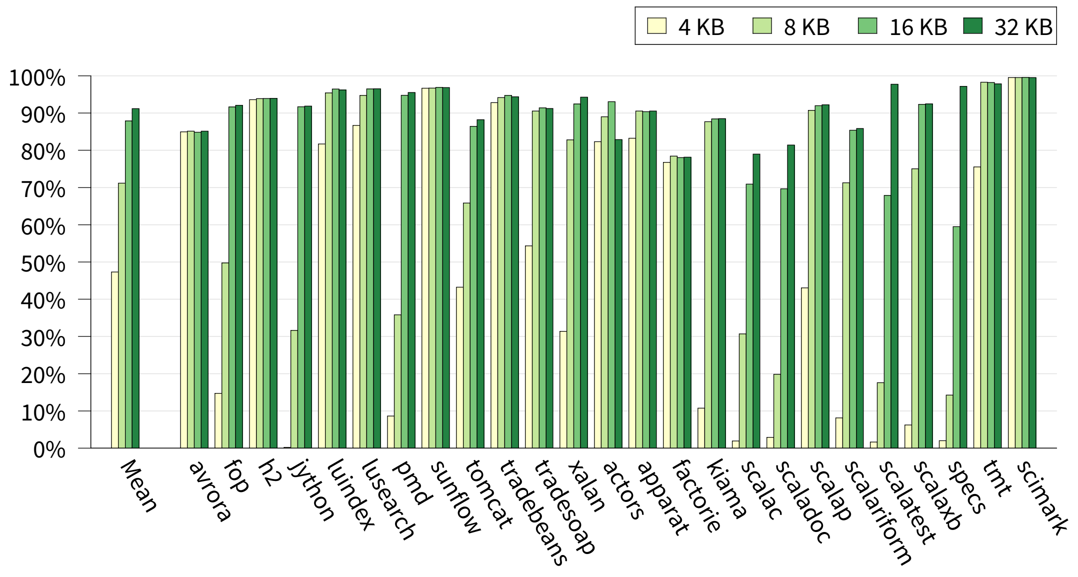


Figure 3.18: Percentage of resolved and complete stack traces when sampling stack fragments of different size

anchor near the Java stack's base, which is vital to decode the stacks of samples in native code. On the other hand, the fragments should not be too large so that copying them is fast and samples do not have to be dropped due to overflowing sample buffers.

We experimented with stack fragment sizes of 4, 8, 16, 32 and 64 KB for our set of benchmarks. Figure 3.18 shows the mean percentage of samples that could be decoded to a complete stack trace, for each benchmark and for each fragment size up to 32 KB. While fragment sizes of 4 KB or 8 KB are insufficient for most benchmarks, a stack fragment size of 16 KB achieves good overall results. Still, the *scalac, scaladoc, scalatest* and *specs* benchmarks benefit significantly from a larger fragment size of 32 KB. Further increasing the fragment size to 64 KB had no effect (not shown). Even with large fragment sizes, not all stack fragments could be decoded when the samples were taken in an unsafe state.

We did not measure a significant slowdown when using larger stack fragments instead of smaller ones and conclude that most of the overhead in taking the samples is caused by interrupting the application, which requires a context switch to kernel space, and another context switch back to the application after the sample is taken. Therefore, we decided to use a fragment size of 32 KB for our further tests.

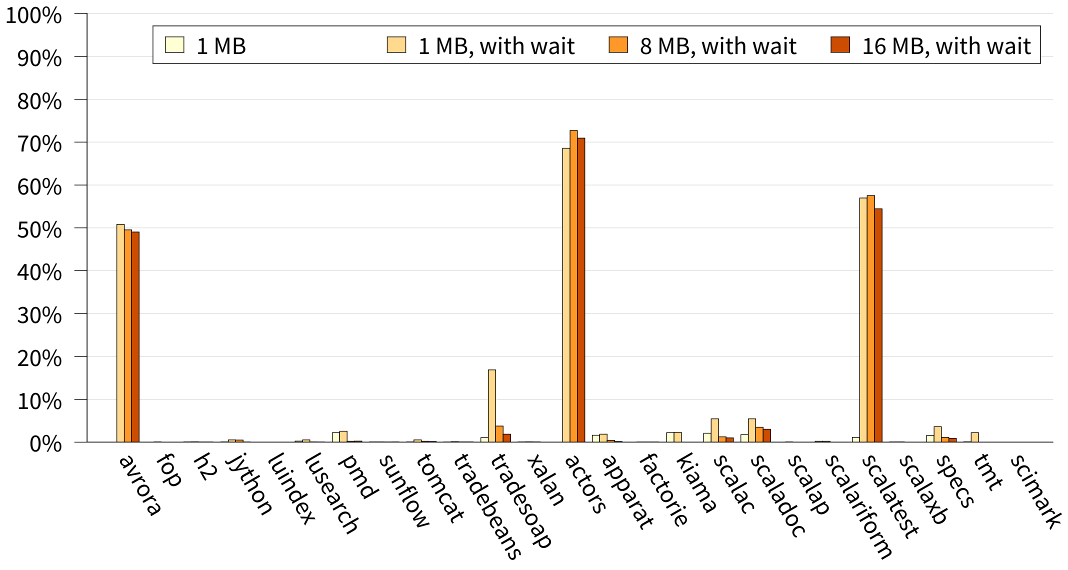


Figure 3.19: Percentage of lost samples using Stack Fragment Sampling with different buffer sizes, with and without sampling waiting periods

Buffer Size and Lost Samples

perf places the samples into a buffer for the respective CPU, from where our agent retrieves them and decodes them to stack traces. When samples are generated so rapidly that the buffer contents cannot be processed in time, the buffer can become full and new samples are dropped. We examined how often this situation occurs, and which buffer sizes result in the fewest lost samples.

Figure 3.19 shows the mean fraction of lost samples at a sampling interval of 0.1 ms when using buffer sizes of 1 MB, 8 MB and 16 MB, with and without sampling waiting periods. When not sampling waiting periods and using 1 MB buffers (which fit around 30 samples), the number of lost samples is almost negligible for most benchmarks, with a maximum of 2.2% lost samples for the *pmd* benchmark. However, when also sampling waiting periods, the number of lost samples increases due to the extra samples taken at context switches. For the *avrora*, *actors* and *scalatest* benchmarks, the fraction of lost samples even exceeds 50%. We found that the thread interactions in these benchmarks trigger an excessive number of context switches and therefore generate many additional samples which cannot be processed quickly enough. Increasing the buffer size does not significantly reduce the number of lost samples for these benchmarks. However, using a buffer size of 8 MB significantly reduces or eliminates lost samples for *tradesoap*. Further increasing the buffer size to 16 MB has no substantial impact.

For our further tests, we therefore used a buffer size of 1 MB per CPU when sampling only running threads, and a buffer size of 8 MB per CPU when additionally monitoring waiting periods.

Undecodable Samples and Effectiveness of Heuristics

Not all stack fragments can be decoded to stack traces because samples can be taken at a time when the top Java frame cannot be located or when a stack walk is not safe, for example when the stack is being unwound during exception handling. We measured how often these situations occur, and how effective our heuristics are in these cases.

Figure 3.20 shows the mean fraction of samples that could be decoded, broken down by the mechanism used to decode the stack. In the figure, *Java* refers to stack fragments which could be decoded with a regular Java stack walk from the current stack pointer. *Anchor* indicates samples in native code for which the on-stack anchor was used to locate the topmost Java frame. *Stub* refers to samples in stub code for which the top Java frame was successfully located by scanning for a return address to Java code. *Prologue* refers to successfully decoded samples that were taken in a method's prologue.

The chart shows that for most benchmarks, a significant fraction of the samples cannot be decoded with only a regular Java stack walk. Using the on-stack anchor substantially improves the fraction of decoded samples, which is also an indicator for how much time is spent native code or in VM code. The stub heuristic also brings a considerable improvement for most benchmarks. The prologue heuristic contributes less than we expected, which we attribute to the JIT compiler's eager inlining policy that reduces the number of method calls and thus decreases the chance of taking a sample in a method prologue. On average, 91.3% of the stack fragments can be successfully decoded to stack traces, which is an indicator for the effectiveness of our heuristics.

Performance Impact

Figure 3.21 shows the median execution time of each benchmark without profiling and with our Stack Fragment Sampling profiler, using 32 KB stack fragments and 1 MB buffers at sampling intervals of 10 ms, 1 ms and 0.1 ms. We sampled only running threads and ignored waiting periods. The times for each benchmark are normalized to its median execution time with no sampling, which is indicated by the horizontal line (note that the y-axis starts at 95%). The error bars indicate the first and third quartiles. On the left-hand side, *G.Mean* shows the geometric means over all benchmarks. For comparison, *G.Mean with wait* shows the geometric means when waiting threads are sampled as well

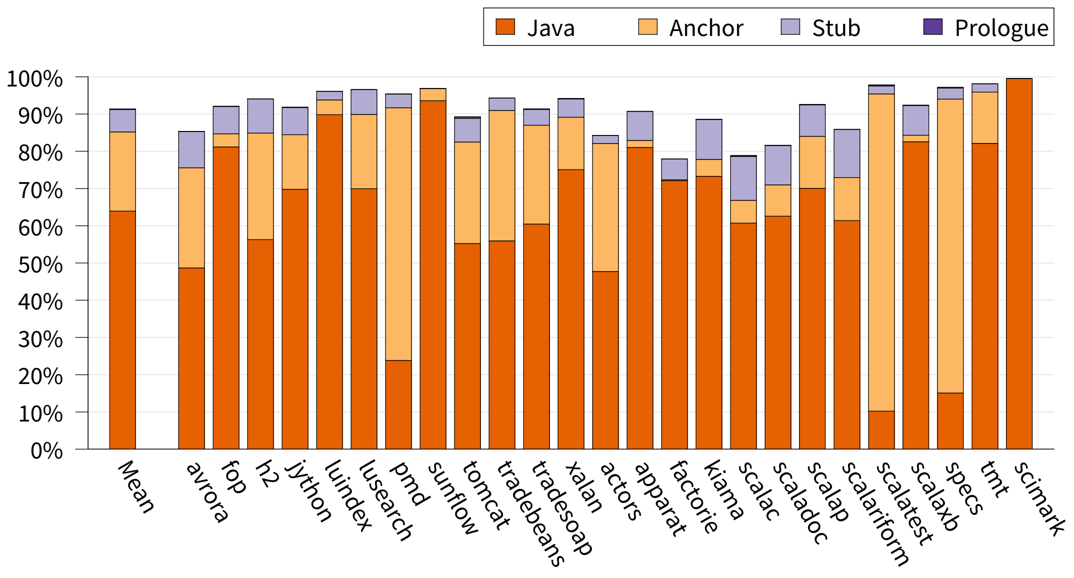


Figure 3.20: Percentage of decodable stack fragments by used stack walk heuristics

(using 8 MB buffers), and *AGCT G.Mean* and *JVMTI G.Mean* show the geometric means of AsyncGetCallTrace sampling and JVMTI sampling for the same set of benchmarks.

The geometric mean overheads of Stack Fragment Sampling over all benchmarks are 1%, 2.1% and 9.7%, at sampling intervals of 10 ms, 1 ms and 0.1 ms, respectively. For 15 out of the 25 benchmarks, the overhead is below 10% even with a sampling interval of 0.1 ms. We found that the other ten benchmarks are more CPU-intensive and thus, our scheduling-aware sampling technique generates more samples for them, especially at small sampling intervals. Notable outliers in our results are the *avrora* and *actors* benchmarks, which show significant overheads even with 10 ms and 1 ms sampling intervals. We attribute this to the excessive number of context switches that these benchmarks trigger because perf adds a slight overhead to each context switch of a monitored thread. In rare cases, such as with the *fop* and *scalap* benchmarks, the execution even appears to become slightly faster with sampling. We suspect that this anomaly results from effects of the sampling on scheduling.

Because of the low performance impact, we consider Stack Fragment Sampling to be well-suited for APM software to continuously monitor production systems.

Accuracy

For analyzing the accuracy of our profilers, we used the CCTs that we collected with 0.1 ms sampling intervals and with sampling of waiting periods enabled for Stack

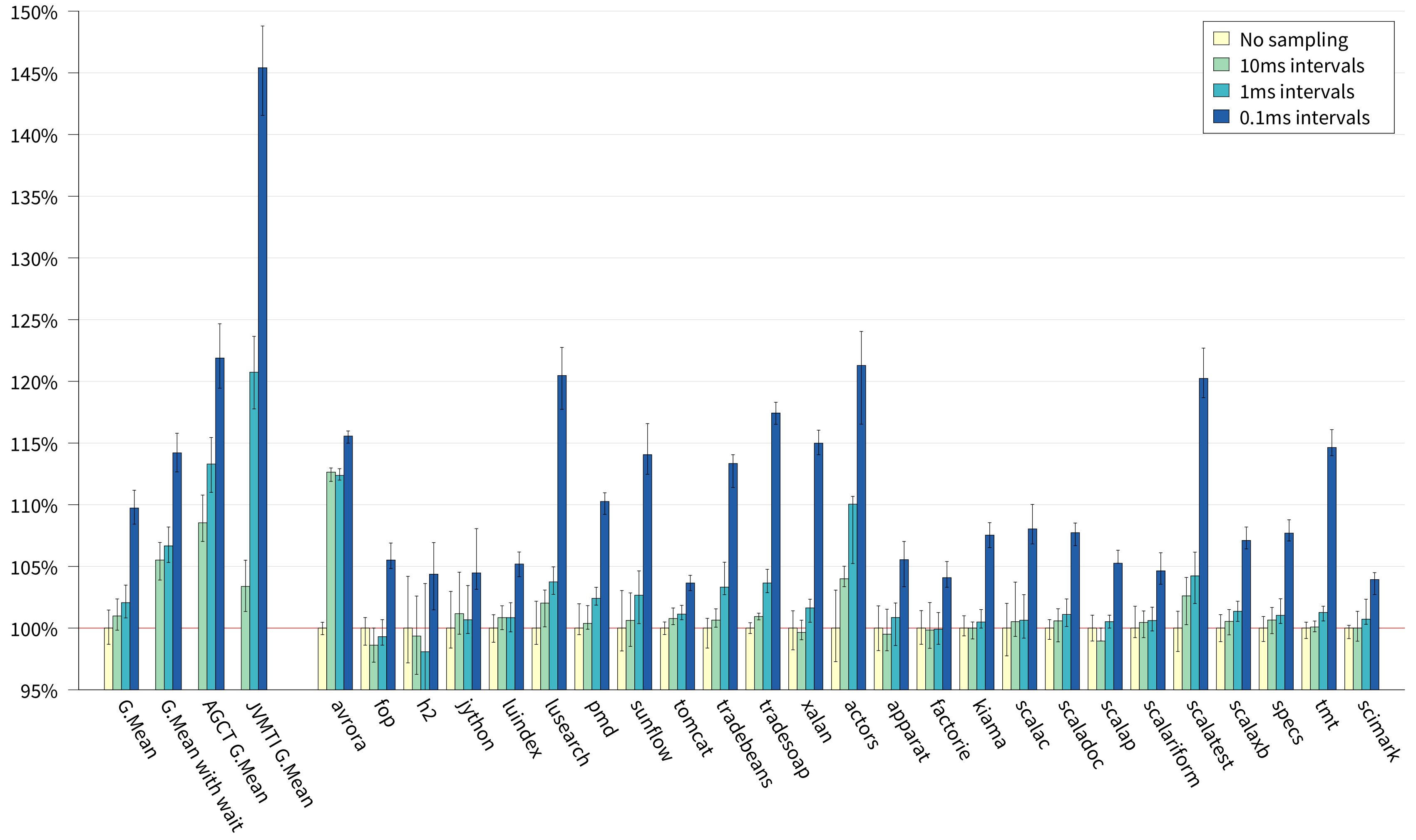


Figure 3.21: Runtime overhead of Stack Fragment Sampling (SFS) with different sampling periods, mean overheads also shown for conventional sampling with JVMTI and with AsyncGetCallTrace (AGCT)

Fragment Sampling because waiting periods cannot be reliably excluded with AsyncGetCallTrace and JVMTI sampling. We further discarded all undecodable or incomplete samples for comparability.

Consistency Analysis. Figure 3.22 shows the median degree of overlap of the average CCT with the individual CCTs for each profiler and benchmark. The error bars indicate the first and third quartiles. The plot demonstrates that the consistency between CCTs is very similar for all three profilers. It also shows that the CCTs from all profilers for *fop, kiama, scalac, scaladoc* and *scalaxb* vary significantly between iterations. We found that these benchmarks spend over 40% of their execution time in many different calling contexts that each make up less than 0.05% of the execution time, in many cases even less than 0.01%. Hence, these calling contexts are seen in only very few samples and even slight shifts in sampling times add up to a significant difference in the resulting overlap.

However, typically only the hottest edges of a CCT are relevant. Figure 3.23 shows the median hot-edge coverage of the individual CCTs by the average CCT for all benchmarks and all three profilers. This analysis also shows whether the average CCT is representative for the individual CCTs. We used a threshold of $T = 0.1$ for the hot-edge coverage, which means that we consider an edge to be hot if its weight is within a tenth of that of the hottest edge. The error bars indicate the first and third quartiles. The hot-edge coverage demonstrates that for most benchmarks, all three profilers consistently identify the same or a very similar set of edges as hot.

These results demonstrate that all three profilers typically generate CCTs that are consistent between different iterations of benchmarks, which also implies that the average CCT is representative for the individual CCTs. However, this does not prove that the profiles are accurate.

Profiler Agreement. Figure 3.24 shows the degree of overlap between the average CCTs from the three profilers. For 17 out of the 25 benchmarks, the overlap between Stack Fragment Sampling and AsyncGetCallTrace exceeds 70%. In general, the CCTs from Stack Fragment Sampling and AsyncGetCallTrace tend to overlap more than those from AsyncGetCallTrace and JVMTI sampling, or than those from Stack Fragment Sampling and JVMTI sampling. We consider this an indicator for the accuracy of Stack Fragment Sampling because AsyncGetCallTrace is not limited to sampling at safepoints and is thus potentially more accurate than JVMTI. A notable example of the agreement of Stack Fragment Sampling and AsyncGetCallTrace is *scimark*, where we confirmed that compiler optimizations significantly affect the accuracy of JVMTI sampling. *pmd* is a counterexample, where AsyncGetCallTrace could not decode a significant number of

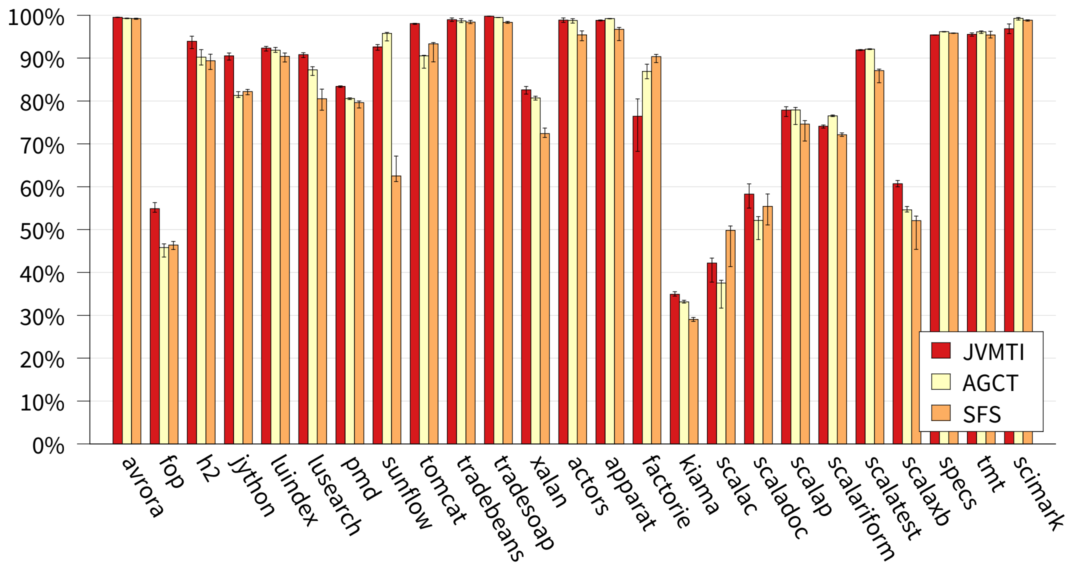


Figure 3.22: Overlap of individual CCTs with the average CCT of profilers using JVMTI, AGCT and SFS

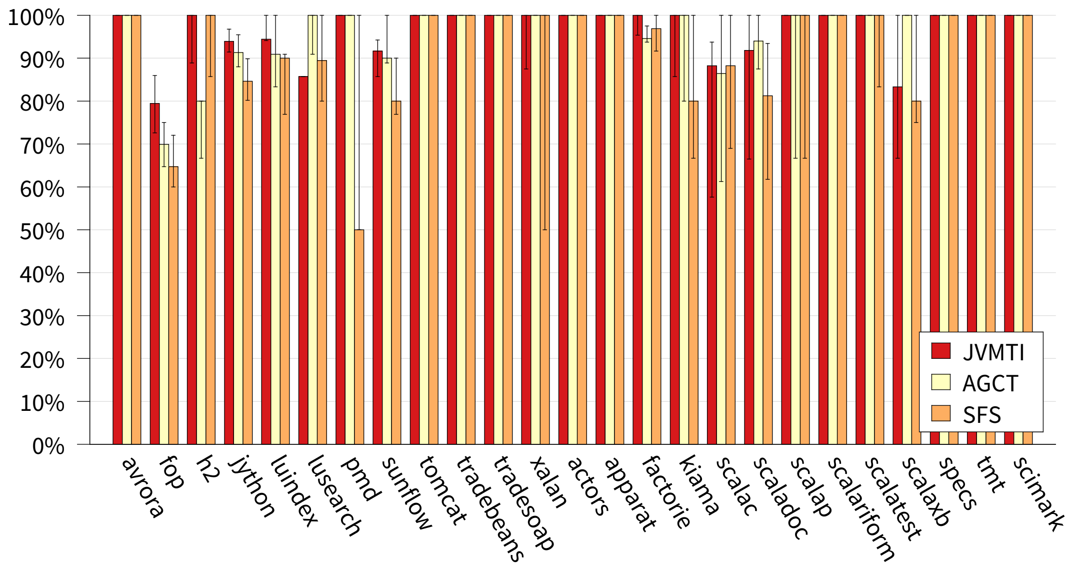


Figure 3.23: Hot-edge coverage of individual CCTs by the average CCT of profilers using JVMTI, AGCT and SFS

stacks and therefore shows a low overlap with the other two profilers, which, on the other hand, agree well with each other. For some benchmarks such as *kiama, scalac, scaladoc* and *tmt*, all three profilers arrived at different CCTs. In the case of *actors,* we found that the amount of lost samples due to excessive context switches impacts the accuracy of the CCTs from Stack Fragment Sampling. With *avrora* and *scalatest* however, all three profilers agree with each other despite the high number of lost samples with Stack Fragment Sampling.

Figure 3.25 shows the hot-edge coverage (using $T = 0.1$) of the average CCTs from AsyncGetCallTrace and JVMTI sampling by the average CCT from Stack Fragment Sampling, and of the average CCT of AsyncGetCallTrace by that of JVMTI sampling. The results are similar to the results with the overlap metric. Stack Fragment Sampling generally agrees even better with AsyncGetCallTrace over the hottest edges of *pmd, xalan* and *scaladoc* than over the entire CCT. All three profilers agree significantly better on the hot edges of *pmd, kiama, scalac* and *scalariform.*

3.4.4 Partial Safepoints and Incremental Stack Tracing

We implemented Partial Safepoints with Self-Sampling and Incremental Stack Tracing in OpenJDK 8u5-b13. We evaluated our implementation with the DaCapo 9.12 benchmark suite and with the benchmarks of the Scala Benchmarking Project 0.1.0, as described in Section 3.4.1. For the evaluation, we implemented a profiler for each of the following sampling approaches which runs as an agent in the Java VM and constructs a CCT:

- Conventional sampling with JVMTI
- Self-sampling with Partial Safepoints (SPS)
- Incremental Self-sampling with Partial Safepoints (ISPS)

We performed all tests on a system with a quad-core Intel Core i7-3770 processor with 16 GB of memory running Ubuntu Linux 14.04 LTS. We disabled hyperthreading, turbo boost and dynamic frequency scaling to avoid their often irreproducible effects on the execution. With the exception of vital system services, no other applications were running while the benchmarks were executed.

As in our evaluation of Stack Fragment Sampling in Section 3.4.3, we executed 30 successive *iterations* of each benchmark with each profiler in a single HotSpot instance. Our profilers track the start and the end of benchmark iterations to generate metrics and a CCT for individual iterations. We discarded the data from the first 20 iterations to adequately compensate for HotSpot's startup phase on our system, and executed multiple *rounds* of each benchmark (with 30 iterations each) with each profiler to ensure

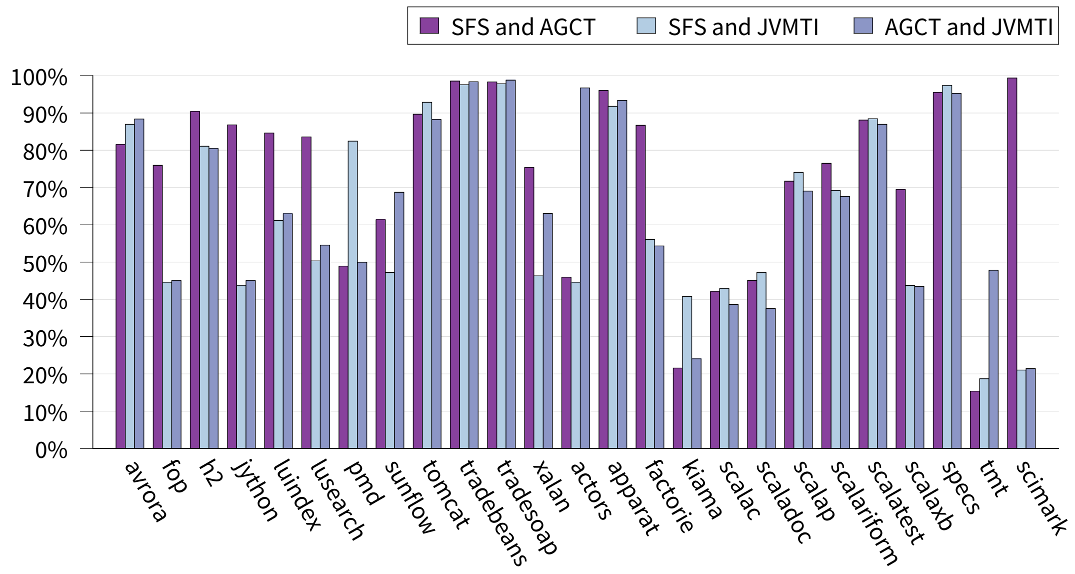


Figure 3.24: Overlap of average CCTs of profilers using JVMTI, AGCT and SFS

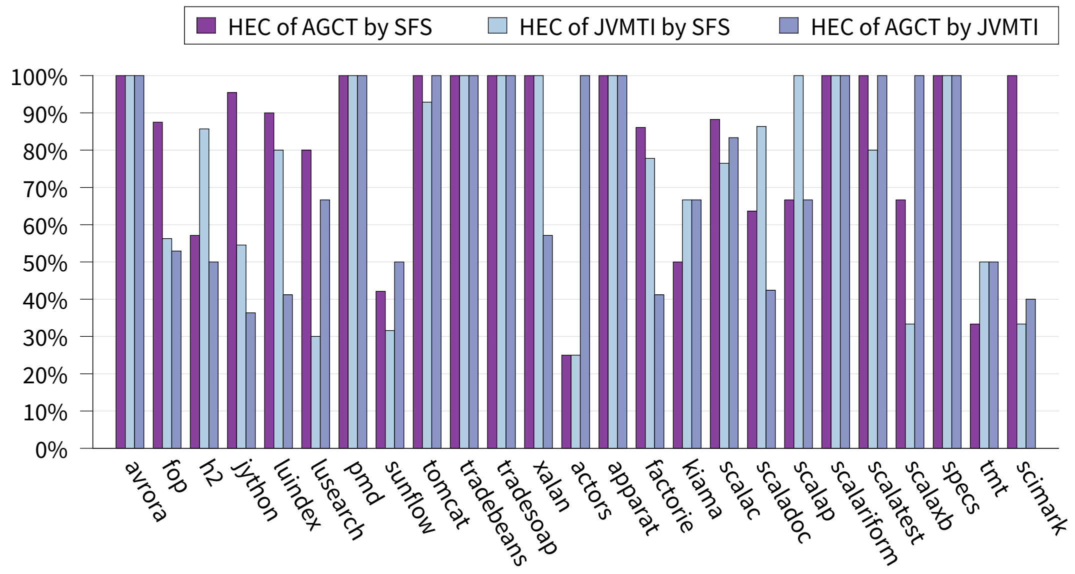


Figure 3.25: Hot-edge coverage of average CCTs of profilers using JVMTI, AGCT, SFS

that the results are not biased by factors that are determined during HotSpot's startup phase.

Unlike in our earlier evaluation of Stack Fragment Sampling, we took the latency of taking samples into account for the sampling interval. For example, when using a sampling interval of 1 ms and taking a sample takes 0.2 ms, our profilers only pause for 0.8 ms instead of 1 ms until they take the next sample. Therefore, all profilers take samples at a similar rate and comparing their overhead and accuracy becomes fairer.

Performance Impact

Figure 3.26 shows the median overheads which we measured for all three profilers, using sampling intervals of 10 ms, 1 ms and 0.1 ms. The error bars indicate the first and third quartiles. The *G.Mean* bars show the geometric means for a sampling interval, and their error bars indicate a 50% confidence interval. The top part of Figure 3.26 shows the overheads for the benchmarks of the DaCapo suite. With 10 ms sampling intervals, JVMTI sampling already has a considerable mean overhead of 10.7% while that of SPS is 2.9% and that of ISPS is even lower at 2.1%. Our techniques perform significantly better than JVMTI sampling for the *lusearch, sunflow, tradebeans, tradesoap* and *xalan* benchmarks. In general, ISPS achieves significantly lower overheads than SPS. This also applies with 1 ms sampling intervals, where JVMTI sampling has a mean overhead of 45.5% while that of SPS is 7.2% and that of ISPS is 6.7%, and for 0.1 ms intervals, where the mean overhead of JVMTI sampling is 107.6% while SPS and ISPS have lower overheads of 42.3% and 27.8%. Unexpectedly, the overhead of JVMTI sampling for *lusearch, sunflow* and *xalan* is reproducibly lower with 0.1 ms sampling intervals than with 1 ms intervals. As we describe in the following section, we found that the latency of JVMTI sampling for these three benchmarks is very unstable and we consider that scheduling effects cause the reduced overhead at shorter sampling intervals.

The bottom part of Figure 3.26 shows the overheads for the benchmarks of the Scala Benchmarking Project. With 10 ms sampling intervals, JVMTI sampling has a mean overhead of 4.8%, while that of SPS is 3.6% and that of ISPS is 3.4%. The improvements from our techniques are more significant with 1 ms intervals, for which JVMTI sampling has a mean overhead of approximately 25.4% while that of SPS is 12.5% and that of ISPS is 9.8%. With 0.1 ms intervals, we measured mean overheads of 98.3% for JVMTI sampling, 68.5% for SPS and 32.2% for ISPS. Overall, ISPS achieves lower overheads than SPS, except for the *tmt* benchmark, which creates a large number of short-lived threads. The ISPS profiler must retrieve all samples of a thread when it ends, which causes significant extra overhead for *tmt*. We were unable to measure the overhead of *actors* with JVMTI sampling with 0.1 ms intervals because that benchmark has an internal timeout which causes it to terminate early due to the high overhead.

Latency

We examined the latency of each sampling technique, which is the time it takes to pause threads and take samples. Figure 3.27 shows box plots of the latencies for all sampling techniques with 1 ms sampling intervals. The boxes show the first quartile, median and third quartile, and the whiskers indicate the 2.5% and 97.5% percentiles. We grouped benchmarks with similar characteristics in *Others.* For all those benchmarks, the latency of JVMTI sampling is low, and SPS and ISPS have slightly lower latencies with less variance.

For the remaining benchmarks, we found notable differences. *actors, tradebeans* and *tradesoap* use a large number of threads and JVMTI sampling has a high median latency for them because it takes longer until all threads have entered a safepoint. With SPS and ISPS, the latencies remain very low because the two techniques require only some of the threads to enter a safepoint. The *lusearch, sunflow* and *xalan* benchmarks use fewer threads, and the median latency of sampling them with JVMTI is not excessively high. However, those three benchmarks execute hot code which the compiler aggressively optimizes and in which it eliminates safepoint checks. The time that it takes for all threads to reach a safepoint check depends strongly on their current locations, so the latencies with JVMTI sampling vary significantly and can even exceed 10 ms. With SPS and ISPS, the median latency and variance are as low as for all other benchmarks because the two techniques wait only until a certain number of threads has entered the safepoint.

The latency measurements show that SPS and particularly ISPS commonly achieve sampling latencies of 0.1 ms or below, which makes them suitable for sampling in very short intervals. Compared to them, sampling with JVMTI has higher latencies which depend on the number of existing threads and on compiler optimizations and can even exceed 1 ms.

Accuracy

To analyze the accuracy of the profilers, we used the CCTs that we collected with a sampling interval of 1 ms. We consider this a reasonable sampling interval because even JVMTI sampling typically has a sampling latency of less than 1 ms, and because it still provides enough samples for short-running benchmarks.

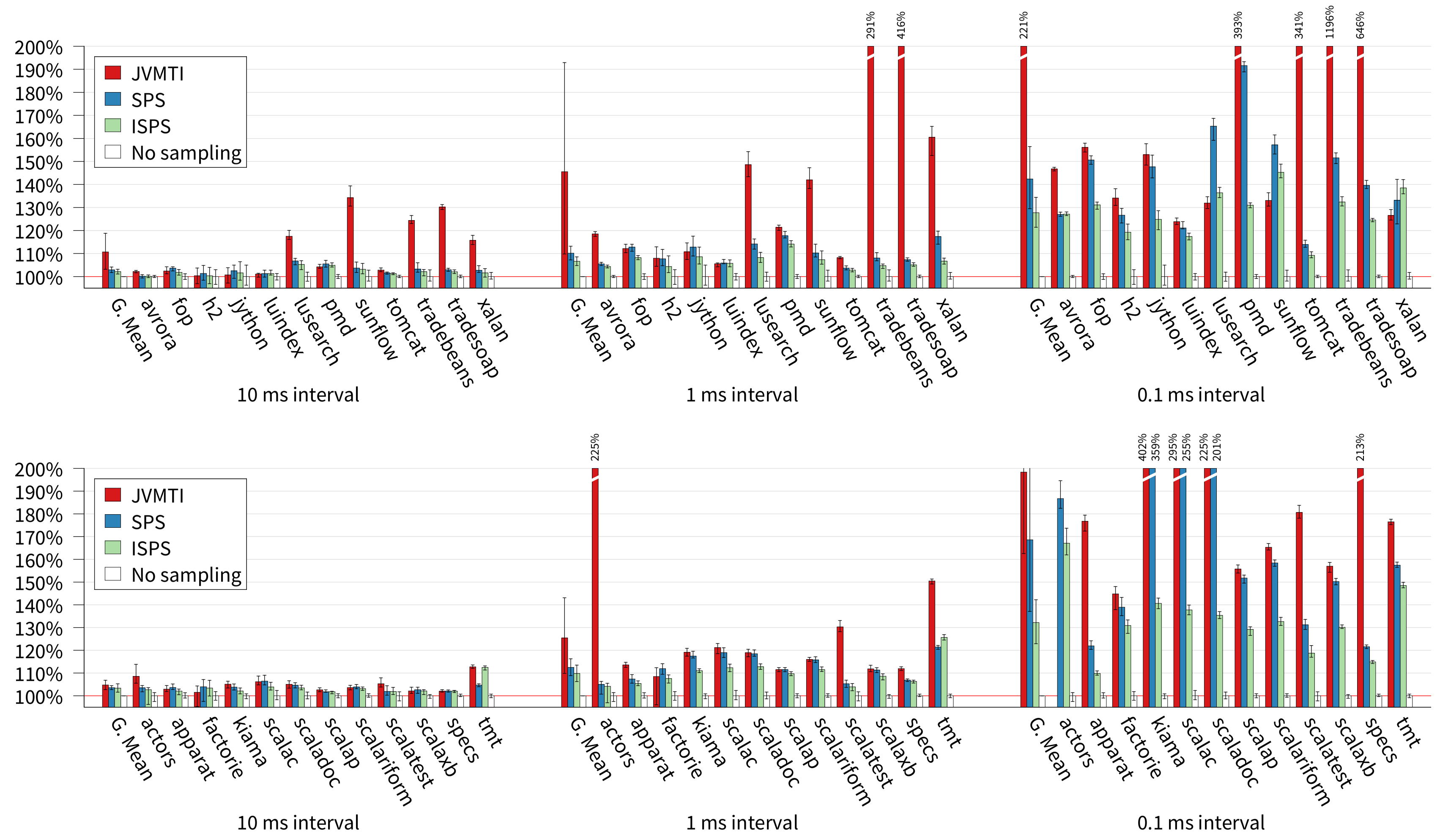


Figure 3.26: Runtime overhead of conventional sampling with JVMTI, Self-sampling with Partial Safepoints (SPS), and Incremental Self-sampling with Partial Safepoints (ISPS), with the benchmarks of the DaCapo suite (top) and of the Scala Benchmarking Project (bottom)

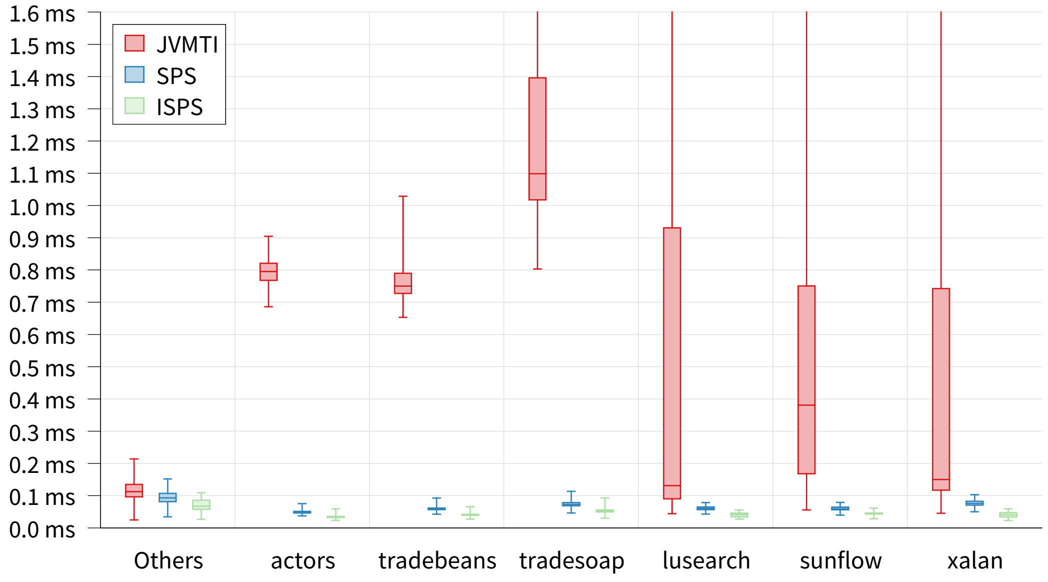


Figure 3.27: Latencies of sampling with JVMTI, SPS and ISPS

Consistency Analysis. Figure 3.28 shows the median overlap of the individual CCTs with the average CCT, while Figure 3.29 shows the median hot-edge coverage of the individual CCTs by the average CCT, for each profiler and benchmark. We used a threshold of $T = 0.1$ for the hot-edge coverage, so we consider an edge to be hot if its weight is within a tenth of that of the hottest edge. The error bars indicate the first and third quartiles. The two plots demonstrate that for every benchmark, the consistency between CCTs is very similar for all three profilers. The lower overlaps of *fop*, *kiama*, *scalac*, *scaladoc* and *scalaxb* suggest that their CCTs vary significantly between iterations. These are the same benchmarks that stood out in the evaluation of profiler consistency for Stack Fragment Sampling in Section 3.4.3, where we found that they spend over 40% of their execution time in calling contexts that are seen in only very few samples and even slight shifts in sampling times add up to a significant difference in the resulting degree of overlap. The hot-edge coverage for these benchmarks is significantly better, with the exception of *fop*.

Profiler Agreement. Figure 3.30 shows the degree of overlap between the average CCTs from all three profilers. The overlaps exceed 70% for all benchmarks, with the exception of *kiama*, *scalac* and *scaladoc*, which also were among the benchmarks with the least consistency between individual CCTs, as we showed earlier in Figure 3.28.

Figure 3.31 shows the hot-edge coverage (using $T = 0.1$) of the average CCT from JVMTI sampling by the average CCTs from SPS and from ISPS, and of the average CCT of SPS

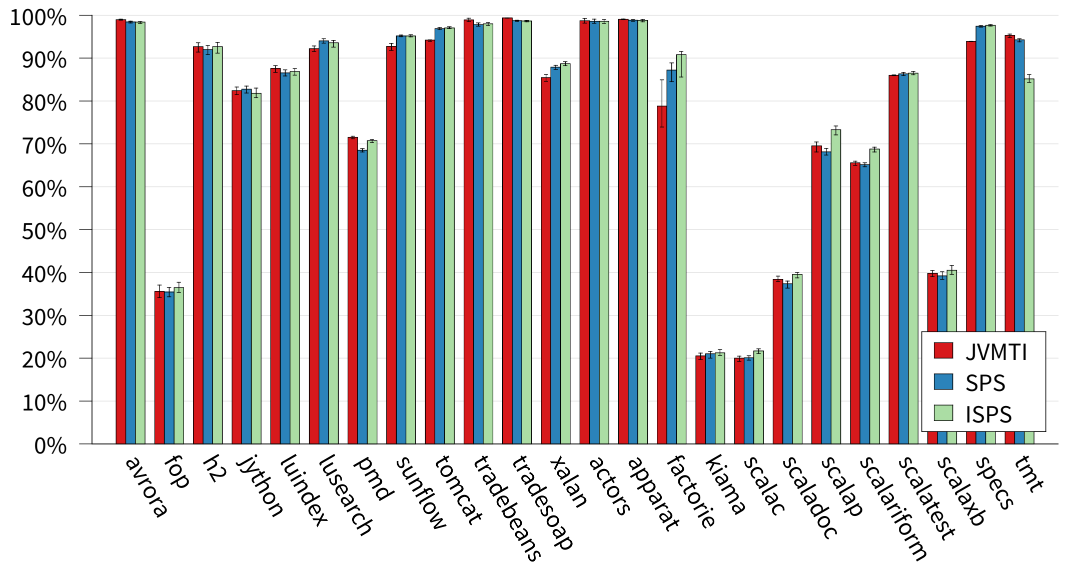


Figure 3.28: Overlap of individual CCTs with the average CCT of profilers using JVMTI, SPS and ISPS

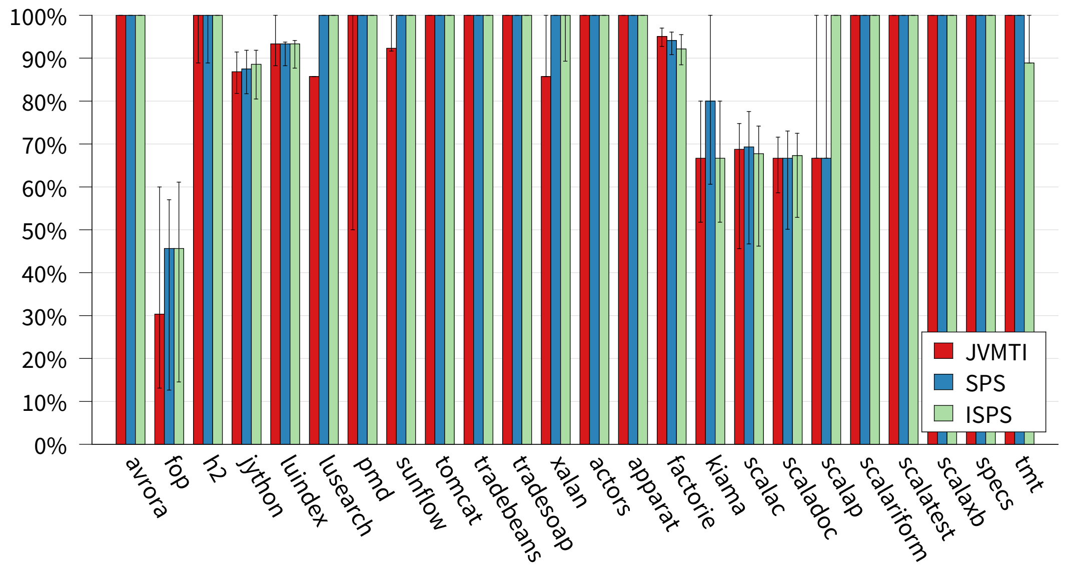


Figure 3.29: Hot-edge coverage of individual CCTs by the average CCT of profilers using JVMTI, SPS and ISPS

by that of ISPS. It demonstrates that SPS strongly agrees with the hot calling contexts identified by JVMTI sampling, with over 90% coverage for all benchmarks. The hot-edge coverage of JVMTI sampling and SPS by ISPS is slightly lower for some benchmarks, which is caused by deep stacks which exceed the otherwise adequate limit of 256 frames that our profilers use for SPS and JVMTI sampling. Such a limit is required for the preallocation of data structures and we found that using a limit that is high enough to fit every stack trace significantly increases the overhead even for shallow stacks. While SPS and JVMTI sampling truncate long stack traces, ISPS allocates data structures on demand and always provides complete stack traces. Although this is actually an advantage of ISPS, it reduces its hot-edge coverage with the other techniques because the complete stack traces do not match their truncated stack traces.

Validity. With partial safepoints, we sample only a subset of all threads, which could cause lower accuracy. Moreover, we sample only those threads that enter a safepoint first, which could lead to a bias toward code that checks for a pending safepoint more frequently than other code. However, the results we presented above support that the profiles obtained with our approaches do not have a lower accuracy nor do they significantly disagree with the profiles of conventional JVMTI sampling.

3.5 Related Work

In this section, we describe related research on sampling-based method profilers, on capturing calling contexts for other dynamic analyses, and on visualizing and analyzing CCTs.

3.5.1 Sampling-based Method Profiling

Mytkowicz et al. demonstrate that four commonly used Java sampling profilers often produce incorrect profiles due to safepoints and optimizations [Mytkowicz10]. They propose a profiler that pauses threads not only at safepoint locations, but at arbitrary locations in the code. However, their profiler samples only the executing method instead of the current calling context. Moreover, the profiler is implemented outside of the VM, where it suffers from limited information on inlined code and on interpreted methods. In contrast to that, Stack Fragment Sampling records calling contexts, does so at arbitrary locations, and leverages the metadata that is available in the VM to profile even heavily optimized code.

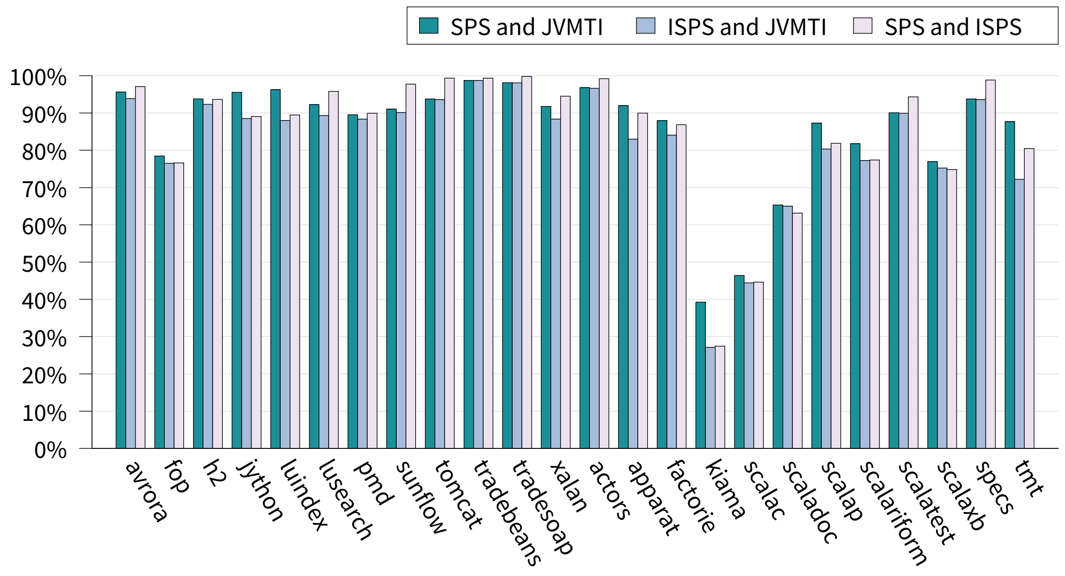


Figure 3.30: Overlap of average CCTs of profilers using JVMTI, SPS and ISPS

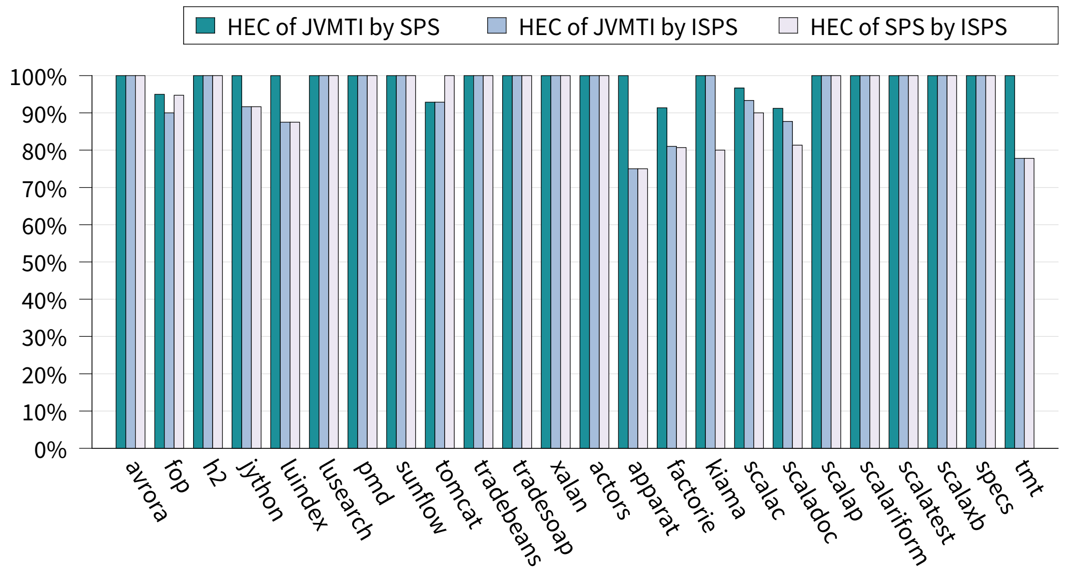


Figure 3.31: Hot-edge coverage of average CCTs of profilers using JVMTI, SPS, ISPS

Whaley describes a VM-internal Java profiler which samples threads at arbitrary code locations and avoids complete stack walks [Whaley00]. Unlike our incremental stack tracing approach, it examines stacks eagerly and uses a spare bit in each stack frame's return address to mark whether a frame has already been examined. Whaley claims a low overhead of 2-4% at 1000 samples per second, but the used VM performs thread scheduling itself ("green threads"), which permits certain assumptions and direct access to thread states. Green threads are uncommon in modern Java VMs because of their disadvantages in systems with multiple CPUs. Our techniques have only slightly more overhead with the high-performance HotSpot VM and work well for multi-processor systems.

Zhuang et al. describe a Java profiler that does not sample calling contexts, but instead instruments the code to sample a sequence ("burst") of calls and returns, and uses heuristics to disable and re-enable sampling to reduce redundant samples [Zhuang06]. The resulting CCTs are claimed to have more than 80% hot-edge coverage and overlap with exhaustive CCTs, but the stated overhead of 20% for 10 ms sampling intervals is significantly higher than that of our techniques.

Binder proposes a Java profiler that instruments methods to maintain a shadow method stack and to periodically capture samples of this stack, and claims a higher accuracy than that of JVMTI sampling [Binder06a]. Unlike our techniques, this profiler can be implemented in pure Java, but its overhead is higher and comparable to that of JVMTI sampling.

Inoue and Nakatani describe a Java profiler that uses the processor's hardware performance monitoring unit to take samples of only the executing method and its stack depth [Inoue09]. The profiler asynchronously builds a CCT by matching the stack depths and caller information of the samples. Samples can be triggered by hardware events, such as memory store instructions, or by a special instruction that can be added in arbitrary code locations. The profiler is reported to achieve an overhead of 2.2% when capturing 8000 samples per second. Unlike our approaches, the profiler's accuracy can suffer from situations in which a method's caller cannot be unambiguously determined.

Serrano and Zhuang propose a Java profiler that captures short traces of method calls and returns and attempts to merge them optimally into approximate CCTs [Serrano09]. The profiler captures these traces by using hardware branch tracing, which records a sequence of recently executed branches in a buffer, and by sampling this buffer. The profiler is claimed to produce highly accurate CCTs at negligible overhead. Nevertheless, its accuracy can suffer when traces cannot be unambiguously merged into a CCT.

3.5.2 Dynamic Analysis Tools

Calling context information is not only useful for method profilers, but also for other dynamic analysis tools.

Ansaloni et al. propose a framework for composing dynamic analysis tools from reusable components [Ansaloni13]. The components allow maintaining a nontrivial shadow state that corresponds to observed program states. This shadow state can capture calling contexts and information that is specific to calling contexts. The framework builds on the earlier research of Marek et al. on a domain-specific language for Java bytecode instrumentation [Marek12]. Marek et al. also describe an approach for offloading a shadow state and its analysis to a process that is separate from the observed application to avoid interference with the application [Marek13]. Sun et al. apply this approach for observing an application that is running on a mobile device while analyzing the collected data on a separate system [Sun15].

Zheng et al. address the issue that many instrumenting dynamic analysis tools produce incorrect results in the presence of optimizing compilers [Zheng15]. They propose a novel technique that makes such tools aware of optimizations and implemented it in the Graal JIT compiler [Graal15]. Among other capabilities, the technique allows tools that capture method calls to avoid instrumenting inlined calls, which incur less runtime cost than non-inlined calls and for which instrumentation tends to disturb optimizations.

Some dynamic analysis tools can capture calling contexts very frequently. In that case, continuously tracking the current calling context and capturing that state is often faster than walking the stack for each sample.

Bond and McKinley describe an approach which encodes the current calling context in an integer variable [Bond07]. They instrument the application code so that it updates the variable on each method call and return. The updates to the variable are deterministic, so that the same calling context is always encoded as the same integer value. A profiler can then sample the current value of the variable at points of interest in the code, such as memory allocations. The encoding of each calling context is probabilistically unique, but collisions can occur and can affect the accuracy of the approach. Continuously tracking the current calling context with this approach is claimed to have an overhead of 3%.

Sumner et al. describe a similar approach for encoding calling contexts as integers, but can do so without collisions [Sumner10]. They report an overhead of about 2%, but their approach has limitations with object-oriented applications and dynamic class loading.

Huang and Bond claim that the accuracy of encoding calling contexts does not scale well with program complexity [Huang13]. They propose an approach that continuously builds a CCT-like data structure through instrumentation. It creates tree nodes eagerly and relies on a modified garbage collector to release unused nodes and to merge duplicate nodes. Using this technique to add calling context information to a memory leak detector or to a data race detector is claimed to introduce around 30-40% extra overhead.

We believe that our incremental stack tracing approach could enable dynamic analysis tools to capture calling contexts very efficiently because it examines parts of the stack that remain the same between samples only once. Moreover, unlike techniques that continuously track the current calling context, incremental stack tracing does not introduce overhead in methods where calling contexts are never captured.

3.5.3 Profile Visualization and Analysis

The CCTs of real-world applications can be very large and complex so that it can be difficult to identify performance bottlenecks in them.

Moret et al. propose calling context ring charts for visualizing large and complex CCTs [Moret10]. Adamoli and Hauswirth describe a CCT visualization and analysis framework that uses ring charts and further supports comparing and clustering CCTs [Adamoli10].

Maplesden et al. describe an approach for finding patterns in a CCT in order to partition it into areas of related functionality [Maplesden15]. They claim that these partitions and their aggregated runtime cost better show optimization opportunities than the individual methods of the CCT.

D'Elia et al. describe algorithms to continuously maintain a "Hot CCT" during profiling that includes only hot calling contexts [D'Elia11]. They claim that it is orders of magnitudes smaller than a regular CCT at comparable accuracy. These algorithms could be used with our sampling techniques to build a memory-efficient profiler.

Han et al. describe an approach to identify performance problems in vast amounts of performance data through pattern mining [Han12]. The low overheads of our profiling techniques make them suitable for continuous use in production systems, and such a pattern mining approach could be used to examine their continuously collected data for performance problems.

3.6 Summary and Future Work

In this chapter, we presented efficient novel approaches for sampling-based method profiling in a high-performance Java VM.

Stack Fragment Sampling relies on the operating system to sample application threads only while they are running and to interrupt them only briefly to copy stack fragments. The fragments are then asynchronously retrieved from a buffer and decoded to stack traces. We described a set of effective heuristics that enable walking the stack even when execution is in native code or in VM-internal code. The performance impact of Stack Fragment Sampling is very low even at high sampling rates and significantly lower than that of comparable profilers. In an analysis of the resulting calling context trees, we found that our technique consistently generates an accurate picture of the program's behavior.

Further work on Stack Fragment Sampling could focus on improving its performance when also taking samples in waiting states. Instead of taking a sample every time the thread is scheduled *out*, a likely more efficient approach is to take samples when a thread is scheduled *in* again, and doing so only if sufficient time has passed since the previous sample. In our implementation, this would require modifications to perf. Another optimization would be to dynamically adjust the fragment sizes to match the size of the stacks of the executed code, which could reduce the costs of copying the fragments for some applications.

Partial Safepoints, Self-Sampling and Incremental Stack Tracing form an alternative novel approach that is independent of the operating system and hardware. This approach also reduces the overhead of sampling Java applications and allows a profiler to target just the running threads. The evaluation with our implementation in HotSpot demonstrates that the approach significantly reduces the sampling overhead while providing similar accuracy as a profiler that uses the common JVMTI interface.

Future work on our approach could focus on addressing limitations to the accuracy of safepoint-based profiling techniques. When the JIT compiler eliminates safepoint checks to optimize hot code regions, it prevents profilers from taking accurate samples in these regions, which can severely distort the profile of some applications. A possible solution could be introducing light-weight "sampling points" that are primarily intended for profiling and make fewer guarantees about safety and therefore do not obstruct compiler optimizations. Instead of eliminating a safepoint check, the JIT compiler could then downgrade it to a sampling point check and still generate fast code. Alternatively, we have also experimented with using facilities of the operating system such as POSIX signals to interrupt individual threads for sampling with incremental stack tracing. However, patching return addresses on the stack is much more complex and error-prone when a thread is not in a known safe state, and a correct implementation in HotSpot would require substantial modifications to other parts of the VM. Finally, a possible enhancement of incremental stack tracing would be to track the values of variables in stack frames. This should be possible at less runtime and space overhead than with exhaustive stack tracing and could be particularly valuable as input for profile-guided optimization.

Chapter 4

Lock Contention Tracing and Analysis

In this chapter, we introduce locks and lock contention in Java. We then describe our novel approach for efficiently tracing lock contention events on the virtual machine level, and for analyzing these events to find the causes of locking bottlenecks instead of just their symptoms. We evaluate different aspects of our approach and discuss related work.

So far, we have presented approaches for sampling-based method profiling that attribute the execution time of an application to individual methods and their callers. This enables a performance engineer to find problems where the application spends more time in a method than what would be expected. Earlier in this thesis, we further identified finding concurrency-related performance issues as a major challenge of APM software. These performance issues are commonly caused by the inefficient synchronization of accesses to shared resources. However, method profilers are not well-suited for finding concurrency-related performance issues because they do not capture these synchronization operations.

Synchronizing accesses to shared resources is commonly done with locks. However, implementing correct and scalable locking is challenging and locks become bottlenecks when multiple threads contend to acquire them. In this chapter, we present a novel approach for analyzing lock contention in Java applications by tracing locking events using mechanisms in the virtual machine. Our approach provides developers with exhaustive information to understand and resolve locking bottlenecks in an application, yet incurs very low overhead which makes it feasible for use by APM software. Most notably, our approach not only records the call chains of threads that are blocked, but also accurately reports the call chains of threads that block other threads by holding a requested lock. This reveals the causes of lock contention instead of showing only symptoms. We published this approach in [Hofer16] after presenting early results in [Hofer15a]. Part of this work has been done by David Gnedt for his master's thesis [Gnedt16], and by master's student Andreas Schörgenhumer.

4.1 Background

The synchronization of accesses to shared resources is the main challenge in concurrent programming and is typically addressed with locks. Each shared resource is protected by a lock. When a thread intends to access a shared resource, it must first acquire the resource's lock. With commonly used mutual exclusion locks, no other threads can then acquire the lock and only the owner thread is permitted to access the resource until it releases the lock again. *Lock contention* occurs when multiple threads try to acquire the same lock at the same time. Typically, the unsuccessful contending threads are blocked (suspended) and remain unproductive until they can acquire the lock.

Locking can be implemented at different granularities. An application can use a single coarse-grained lock to protect all of its shared resources, but it can also use many fine-grained locks to protect small units, such as parts of a data structure. Because subtle mistakes in locking can cause anomalies that are difficult to detect and to debug, coarse-grained locking can be favorable because it is easier to implement correctly. However, coarse-grained locking tends to suffer from frequent lock contention, which can eliminate any gains from parallelization. During development, it is difficult to judge in which cases more fine-grained locking would significantly improve performance and in which cases it would just make the application more complex and error-prone. Therefore, analyzing lock contention at runtime is vital to measure its effect on the performance of an application and to identify bottlenecks where more fine-grained locking is worth the additional complexity.

Lock contention analysis is valuable both during development and in production. Server applications in particular are deployed on machines with significantly more processor cores and memory than a developer workstation has, and must handle workloads that are often orders of magnitude larger than the workloads used for testing. Under such conditions, concurrent code can behave very differently, making it difficult to reproduce and to debug bottlenecks in locking on a smaller scale. A lock contention analysis approach that is feasible for use in a production environment should have minimal overhead while still providing information suitable to identify and comprehend bottlenecks in an effective way.

The following sections give an overview over Java's locking capabilities and their implementation. Java has intrinsic support for locking, but it also provides the *java.util.concurrent* package that contains explicit locks and advanced synchronization mechanisms.

4.1.1 Java Intrinsic Locks (Monitors)

Each Java object has an intrinsic mutual-exclusion lock associated with it, which is also called the object's *monitor.* Developers can insert *synchronized blocks* in their code and specify an object to use for locking. That object's lock is then acquired before entering the block and executing its statements, and released when exiting the block. When threads contend for a lock, that is, when one thread has acquired a lock by entering a synchronized block, and other threads are trying to enter a synchronized block using the same lock, those threads are blocked until the owner exits its synchronized block and releases the lock. Developers can also declare entire methods as synchronized, which then acquire the lock of the *this* object when called, and release it again when returning. A notable property of synchronized blocks and synchronized methods is that a lock is guaranteed to be released in the same method and scope in which it was acquired, even when an exception is thrown.

Intrinsic locks further support conditional waiting with the *wait, notifyAll* and *notify* methods. These methods may only be called on an object while holding its lock. The *wait* method releases the lock and suspends the calling thread. When another thread calls *notifyAll* on the same object, all threads that are waiting on its lock are resumed. Each resumed thread then attempts to acquire the lock, and once successful, it continues execution. In contrast to *notifyAll,* the *notify* method wakes up only a single waiting thread. This mechanism is typically used in producer-consumer scenarios, such as threads in a thread pool that wait for tasks to execute.

Figure 4.1 shows an example of using intrinsic locks to implement a thread-safe blocking queue. The field *q* holds the list of queue items, and the list object's intrinsic lock is used to ensure mutually exclusive queue accesses. The *enqueue* method has a synchronized block to acquire the lock of *q* before it appends an item and calls *notifyAll* to resume any threads waiting on *q*. The *dequeue* method also uses a synchronized block and calls *awaitNotEmpty,* which invokes *wait* on *q* as long as the queue is empty. The *wait* method releases the lock of *q* and suspends the calling thread until another thread resumes it by calling *notifyAll* from *enqueue*. Alternatively, when the thread is interrupted while waiting, *wait* throws an *InterruptedException,* which we catch. In either case, *wait* first attempts to reacquire the lock of *q* and waits until it is successful. When execution continues in *awaitNotEmpty,* it checks again whether the list is not empty. When *awaitNotEmpty* finally returns, the list is guaranteed to be not empty, and *dequeue* can remove and return an item.

The semantics of intrinsic locks are implemented entirely in the Java VM, usually in a very efficient way so their use incurs only significant overhead when threads actually

```
class BlockingQueue {
  private final List<Object> q = new LinkedList<>();

  void enqueue(Object item) {
    synchronized(q) {
      q.add(item);
      q.notifyAll();
    }
  }

  Object dequeue() {
    synchronized(q) {
      awaitNotEmpty();
      return q.remove(0);
    }
  }

  void awaitNotEmpty() {
    while (q.isEmpty()) {
      try {
        q.wait();
      } catch (InterruptedException e) { }
    }
  }
}
```

Figure 4.1: Blocking queue with Java intrinsic locks

contend for locks [Bacon98, Pool14, Russell06]. Implementations are typically *non-fair* and allow threads to acquire a recently released lock even when there are queued threads that requested that lock earlier. This increases the throughput by avoiding additional overhead that is caused by suspending and resuming threads, and by better utilizing the time periods between when one thread releases a lock and when a queued thread is scheduled and can acquire the lock [Goetz06].

4.1.2 The java.util.concurrent Package

Java 5 introduced the *java.util.concurrent* package with classes that provide useful synchronization mechanisms, such as concurrent collections and read-write locks [Lea05]. Most of these classes do not use Java's intrinsic locks, but rather rely on the newly introduced *LockSupport* facility, which provides a *park* method that a thread can call to park (suspend) itself, and an *unpark* method that other threads can call to resume a parked thread. Using these two methods as well as atomic compare-and-set operations, the semantics of java.util.concurrent classes can be implemented entirely in Java. The *AbstractQueuedSynchronizer* class further provides a convenient basis for implementing synchronization mechanisms with wait queues. Because these classes are public, application developers can also implement custom synchronization mechanisms on top of them.

ReentrantLock is an example of a mutual exclusion lock in java.util.concurrent that is semantically similar to intrinsic locks, but is implemented entirely in Java on top of AbstractQueuedSynchronizer. Code that uses ReentrantLock must explicitly call its *lock* and *unlock* methods to acquire and release the lock. ReentrantLock also supports conditional waiting by calling *await, signalAll* and *signal* on an associated *Condition* object. Unlike intrinsic locks, an arbitrary number of such condition objects can be created for each lock. Moreover, ReentrantLock has a fair mode which guarantees first-come-first-serve ordering of lock acquisitions and conditional wake-ups. This mode reduces the variance of lock acquisition times, typically at the expense of throughput.

Figure 4.2 shows an example of a thread-safe blocking queue that uses ReentrantLock, with the same behavior as the example in Figure 4.1. However, instead of using the intrinsic lock of the list object *q*, the code explicitly creates a ReentrantLock object *lock* and a Condition object *notEmpty* that is associated with the lock object. The methods *enqueue* and *dequeue* explicitly call *lock* and *unlock* on the lock to acquire and release it. The implementation of *lock* attempts to acquire the lock, but if the lock is already owned by another thread, the method adds the current thread to the lock's internal queue of blocked threads and then calls *LockSupport.park* to suspend the thread until the lock becomes available. The implementation of *unlock* releases the lock, checks the lock's queue of blocked threads, and when necessary, calls *LockSupport.unpark* to resume a blocked thread which can then acquire the lock. The methods *enqueue* and *awaitNotEmpty* call the methods *await* and *signalAll* on the condition *notEmpty*. These methods must be called while holding the condition's associated lock and are similar to the intrinsic lock methods *wait* and *notifyAll*, but use *LockSupport.park* and *LockSupport.unpark* to suspend and resume threads which are waiting for a condition.

4.2 Lock Contention Event Tracing

Analyzing lock contention in an application requires observing individual locking events and computing meaningful statistics from them. Maintaining those statistics in the synchronizing threads interferes with the execution of the application, and requires synchronization as well, so it can become a bottleneck itself. Therefore, we decided to

```java
class BlockingQueue {
  private final List<Object> q = new LinkedList<>();
  private final ReentrantLock lock = new ReentrantLock();
  private final Condition notEmpty = lock.newCondition();

  void enqueue(Object item) {
    lock.lock();
    try {
      q.add(item);
      notEmpty.signalAll();
    } finally {
      lock.unlock();
    }
  }

  Object dequeue() {
    lock.lock();
    try {
      awaitNotEmpty();
      return q.remove(0);
    } finally {
      lock.unlock();
    }
  }

  void awaitNotEmpty() {
    while (q.isEmpty()) {
      try {
        notEmpty.await();
      } catch (InterruptedException e) { }
    }
  }
}
```

Figure 4.2: Blocking queue with ReentrantLock from java.util.concurrent

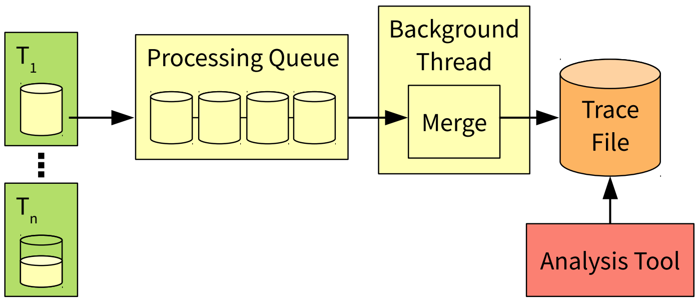


Figure 4.3: Writing and processing trace events

record only those events in the application threads and to analyze them later. In this section, we describe how we efficiently record events and metadata, which events we record, and how we reconstruct thread interactions from the recorded events.

4.2.1 Writing and Processing Events

Figure 4.3 shows an overview of how we write and process trace data in our approach. Naturally, we trace locking events in different application threads. Writing those events to a single shared trace buffer would require synchronization and could become a bottleneck. Therefore, each application thread T_i allocates a *thread-local trace buffer* where it can write events without synchronization. We modified the locations in the HotSpot code where relevant events occur so that they record these events in the current thread's trace buffer. When the trace buffer is full, the thread submits it to the *processing queue* and allocates a new buffer. A *background thread* retrieves the trace buffers from the queue and processes their events. In the depicted scenario, it merges them into a single *trace file*. This trace file can be opened in an *analysis tool* for offline analysis and visualization.

We encode the events in an efficient binary representation to facilitate fast writing, to reduce the amount of generated data and to keep the memory usage of the trace buffers low. Because each buffer is written by just a single thread, we can store the identifier of that thread once per buffer instead of recording it in each event. Submitting a full buffer to the queue requires synchronization, but we assign a random size between 8 KB and 24 KB to the individual buffers, which is large enough so that this is an infrequent operation. The randomization avoids that threads which perform similar tasks try to submit their buffers at the same time and contend for queue access.

Our design supports writing a trace file for offline analysis as well as analyzing the recorded events online. We process the recorded events in Java code, while we write events and manage trace buffers in native code. Therefore, the event processing code uses a thin native code interface to dequeue the trace buffers from the processing queue and to wrap them in *DirectByteBuffer* objects, which can be read from Java without copying memory. To write trace files, we use the Java NIO subsystem [JavaNio15], which can use the DirectByteBuffer objects directly. We also support fast compression of the trace data by using a Java implementation of the high-performance LZ77-type compression algorithm *Snappy* [Snappy15, Sundstrom15]. Our online analysis mode currently generates a text file with statistics, but it could be extended to provide an interface for the Java Management Extensions (JMX, [JMX15]) to configure analysis parameters and to access the produced statistics.

4.2.2 Tracing Intrinsic Locks

Locking causes a major performance impact when threads contend for a lock. Threads that fail to acquire a lock are suspended and cannot make progress, and a thread that releases a contended lock must also do extra work to resume a blocked thread as its successor. However, locking itself is not expensive in the HotSpot VM. An available intrinsic lock can be acquired with a single compare-and-set instruction in many cases. When a thread holds a lock only briefly, another thread that requests that lock can often still acquire it through spinning without suspending itself. Therefore, we chose to record only *lock contention* with our approach instead of recording *all* lock operations.

Conceptually, each Java object has an intrinsic lock associated with it. This lock stores its current owner thread, the threads that are blocked trying to acquire it, and the threads that are waiting for notifications on it. However, in a typical application, most Java objects are never used for locking. Therefore, the HotSpot VM assigns a lock to an object only when threads start using that object for locking. Even then, *biased locking* can avoid allocating an actual lock as long as the object's lock is never used by more than one thread [Russell06].

The act of assigning a lock to an object is called *lock inflation.* Intrinsic locks in HotSpot are data structures in native memory which are never moved by the garbage collector and can therefore be uniquely identified by their address. Whenever lock inflation happens, we record an *inflation event* with the lock's address, the object that the lock is being assigned to, and that object's class. In the trace events that follow, we record only the lock's address. When analyzing the trace, we are thus still able to infer the lock's associated object and its class from the inflation event.

When a thread is blocked from entering a synchronized block or method because it cannot acquire a lock, we record a *contended enter event* with the lock's address, a timestamp, and the call chain of the thread. Recording the call chain is expensive, but the impact on performance is moderate because the thread is unable to make progress anyway.

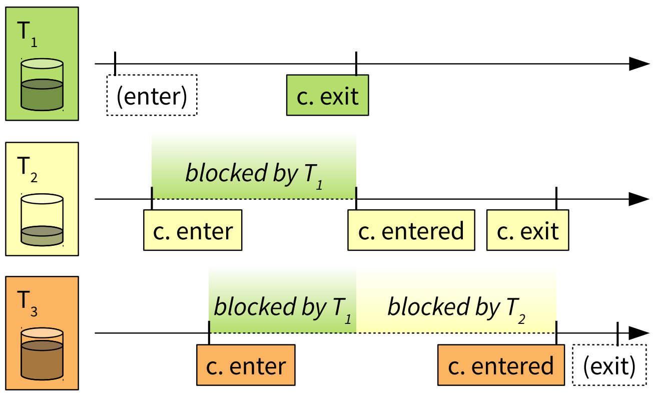


Figure 4.4: Events in three contending threads

When the thread later acquires the lock, we record a *contended entered event* with only a timestamp.

With those two events, we can determine which threads were blocked when trying to acquire a lock, how long they were blocked, and what call chains they were executing. However, this information reveals only the symptoms of locking bottlenecks and not their causes. To determine the causes of contention, we record events not only in those threads that are blocked, but also in those threads which block other threads by holding a contended lock. We modified all code paths in the VM that release a lock when exiting a synchronized block or method so that they check whether any threads are currently blocked on that lock. If so, we write a *contended exit event* with a timestamp and the call chain. We delay recording the call chain and writing the event until after the lock has been released to avoid causing additional contention.

Figure 4.4 shows an example of events that we trace for three threads T_1, T_2 and T_3 that are executing in parallel and are contending for a single intrinsic lock. First, T_1 acquires the lock without contention, so we do not record an event. Next, T_2 tries to acquire the lock, but the lock is held by T_1, so T_2 writes a *contended enter event* in its trace buffer and suspends itself. T_3 then also fails to acquire the lock and also records a *contended enter event*. When T_1 finally releases the lock, it sees T_2 and T_3 on the lock's queue of blocked threads, so it resumes thread T_2 and writes a *contended exit event*. T_2 acquires the lock and writes a *contended entered event*. When T_2 later releases the lock, it sees T_3 on the lock's queue, resumes T_3 and writes a *contended exit event*. T_3 then acquires the lock and writes a *contended entered event*. When T_3 releases the lock, no threads are queued, and T_3 continues without writing an event. When the trace analysis later examines the events of all threads, it can infer from the *contended exit events* that T_2 and T_3 were being blocked by T_1 holding the lock, and that T_3 was subsequently being blocked by T_2. It can further compute the duration of those periods from the timestamps in the events.

Event Ordering

The trace analysis needs to arrange all events for a specific lock in their correct order to analyze them. For this purpose, we introduced a counter in HotSpot's intrinsic lock structure. When we write an event for a lock, we atomically increment the lock's counter and record its new value in the event as its *per-lock sequence number.* Unlike timestamps (which we also record), these sequence numbers have no gaps between them, which enables the analysis to determine whether it can already analyze a sequence of parsed events, or whether there are still events in a trace buffer from another thread that it has not parsed yet. This considerably simplifies and speeds up the analysis. However, when a thread records an event and then does not submit its trace buffer for a long time, it still delays the analysis of subsequent events. For this reason, we also reclaim the trace buffers of all threads during garbage collections and add them to the processing queue. The threads subsequently allocate new buffers to write further events.

Sequence numbers are also more reliable than timestamps for correctly ordering events. Although we retrieve the timestamps from a monotonic system clock, this clock typically uses a timer of the CPU or of the CPU core on which the thread is currently executing. When the timers of different CPUs or CPU cores are not perfectly synchronized, the recorded timestamps are not accurate enough to establish a happened-before relationship, while our atomically incremented sequence numbers always guarantee a correct order.

Conditional Waiting

As we described in Section 4.1.1, the *wait* method temporarily releases a lock and suspends the calling thread until another thread resumes it. When the released lock was contended, we also need to write a *contended exit event* with a call chain. However, *wait* may be called in a different method than the one that acquired the lock, as is the case with *awaitNotEmpty* in Figure 4.1. In this case, recording the current call chain would misrepresent the source of contention. Instead, we generate a *forward reference* to the call chain of the method that acquired the lock and record this reference in the *contended exit event*. We remember the reference so that when the thread later releases the lock in the method which acquired it, we record an additional event that resolves the reference to that method's call chain.

When a waiting thread is woken up using *notify*, the thread tries to reacquire the lock. When it has to wait for the lock, we also record a *contended enter event.* However, contention after conditional waiting is not necessarily problematic. Often, multiple threads are woken up at once and only the first of them can make progress. The other

```
class LockSupport {
  static void park(Object blocker);
  static void unpark(Thread thread);
  static void parkNanos(Object blocker, long timeout);
  static void parkUntil(Object blocker, long deadline);
  // ... variants of park without blocker argument ...
}
```

Figure 4.5: Methods of the LockSupport class

threads will find that the condition that they have been waiting for is again not met, and therefore call *wait* again. For that reason, we use an extra flag in the event to indicate when contention was preceded by conditional waiting. This allows us to classify this type of contention differently in the analysis.

4.2.3 Tracing Park/Unpark Synchronization

Most synchronization mechanisms in java.util.concurrent are implemented entirely in Java. To trace them, it would be possible to instrument each class individually and to generate custom trace events that are specific to the semantics of the class in question. However, what these classes have in common is that they rely on the *park* and *unpark* methods that the VM provides through the *LockSupport* class.

Figure 4.5 shows an outline of the *LockSupport* class. The *park* method parks (suspends) the calling thread. A parked thread remains suspended until another thread calls *unpark* on it. The methods *parkNanos* and *parkUntil* suspend the calling thread until the thread is unparked, or until a timeout has elapsed or a deadline is passed, whichever comes first. The callers of the park methods pass a *blocker object* which represents the entity that caused the thread to park. The blocker object is typically the synchronization object itself (such as a lock) or an object associated with it. For example, *ReentrantLock* passes an instance of its inner class *NonfairSync* (or *FairSync*). Although passing a blocker object is optional, implementers are strongly encouraged to do so for diagnostic purposes.

When a thread cannot acquire a lock, it calls *park* to suspend itself and passes a blocker object that represents the lock. When a thread releases a contended lock, it calls *unpark* to resume a parked thread that has requested the lock. Therefore, we decided to trace the individual *park* and *unpark* calls in all threads. The class of the blocker object reveals the used type of lock (or other synchronization mechanism) and enables us to infer the exact semantic meaning of the park and unpark calls, so we can correlate them with each other and determine which threads blocked which other threads.

We also need to arrange the events for the park and unpark calls in their correct order to analyze them. Ideally, we would also generate separate sequence numbers for each lock so that the events from different locks can be ordered and analyzed independently. However, the lock is represented by the blocker object, which is passed only to *park,* not to *unpark,* and is therefore unknown when writing an unpark event. Therefore, we assign a *global sequence number* to each event, which establishes a definitive happened-before relationship between all park and unpark calls in all threads. We further use the global sequence number of an event to refer to that event from other events. We use a single global counter that we atomically increment to generate the global sequence numbers.

When a thread calls *park,* we record a *park begin event* with a sequence number, a timestamp, the identity of the blocker object, the object's class, and the call chain. Moreover, we include whether a timeout or deadline was specified upon which the thread would resume even without being unparked.

When a thread calls *unpark* to resume a parked thread, we record an *unpark event* with a sequence number, a timestamp, the identifier of the unparked thread, and the call chain. Moreover, we store the unpark event's sequence number to a thread-local structure of the unparked thread.

As soon as the unparked thread resumes its execution, we write a *park end event* with a sequence number and a timestamp. We retrieve the sequence number of the corresponding *unpark event* from the thread-local structure and also include it in this event, so the analysis can easily match the two events. Because a call to *park* can also end due to a timeout, we also record in the event whether this was the case.

Figure 4.6 shows an example of events that we record in four contending threads that use a non-fair *ReentrantLock.* Initially, T_1 is able to acquire the lock without contention. Next, T_2 tries to acquire the lock and fails, so it enters the queue of the lock and parks, and thus, we record a *park begin event.* T_4 then also fails to acquire the lock, enters the queue and parks, so we record another *park begin event.* When T_1 releases the lock, it unparks T_2 as its successor and we write an *unpark event.* However, T_3 is able to acquire the lock before T_2 resumes its execution. T_2 writes a *park end event,* but finds that the lock is still unavailable, so T_2 parks again, and we write another *park begin event.* Finally, T_3 releases the lock and unparks T_2 as its successor. When T_2 resumes its execution, we record a *park end event,* and T_2 is finally able to acquire the lock. T_4 remains parked.

During the analysis of the trace, we examine the first *park end event* of T_2, which leads us to the *unpark event* of T_1. Because a blocker object of class *ReentrantLock.NonfairSync* was recorded, we can infer that the unpark call was the consequence of an *unlock* operation, and that T_1 held the lock before the *unpark event.* The same applies to the the second

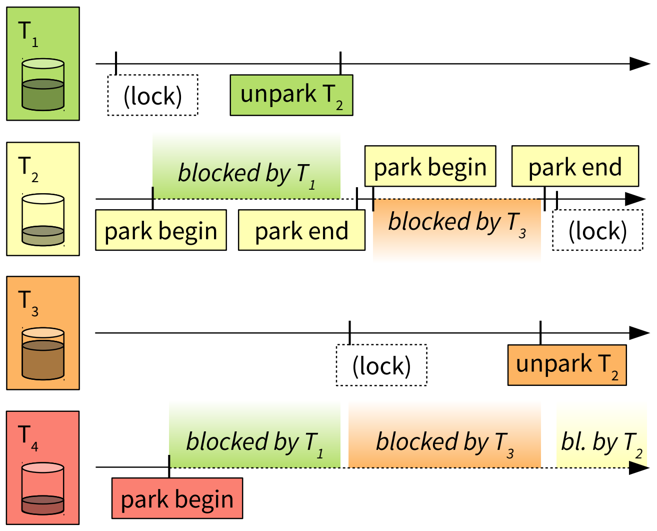


Figure 4.6: Park/unpark events in contending threads

unpark call, where T_3 held the lock. We can then account for the contentions in T_2 as being caused by T_1 and T_3 and their recorded call chains. Thread T_4 was also blocked by T_1 and T_3, although this is not obvious because unpark was called only on T_2. However, because T_4 specified the same blocker object as T_2 when parking, we can infer that it was blocked by the same lock owners and also account for its contention as being caused by T_1 and T_3. Because the time between an *unpark event* and a *park end event* cannot be precisely attributed to the previous or to the next lock owner, we simply attribute such typically very short time periods to an unknown lock owner.

4.2.4 Metadata

Our traces contain a significant amount of repetitive data, such as the identities of threads, classes and objects, as well as call chains. Therefore, we want to collect and encode such data as efficiently as possible. However, for the data to be valuable for a user, we need to provide a meaningful representation, such as the name of a thread instead of just its numeric identifier. We decided to address this issue with *metadata events*. When we encounter an entity (such as a thread or a class) for the first time, we record a metadata event with a unique identifier for the entity and include information that is meaningful to a user. In the events that follow, we refer to that entity with only its identifier.

When the application launches a new thread, we record a *thread start event* with the thread's name and the numeric identifier that the Java runtime assigned to the thread. In future events, we refer to the thread only with that identifier. When the name of a thread is changed later, we record a *thread name change event* with the new name.

When we encounter a specific Java class for the first time, we write a *class metadata event* with the fully-qualified name of the class. HotSpot stores the metadata of a class in a data structure with a constant address, so we use that address as the unique identifier of the class. We introduced an additional field in the class metadata structures that indicates whether the class has already been defined in the trace. Because two threads might race to write a class metadata event, we atomically update that field before writing an event, and the thread that succeeds in updating the field then writes the event.

Finding a unique identifier for Java objects is difficult. Because objects are moved by the garbage collector, their address is not suitable as an identifier. Instead, we refer to an object by recording its *identity hash code* and its class in our events. In HotSpot, the identity hash code of an object is a 31-bit integer that is randomly generated and stored with the object. In rare cases, two different objects of the same class can be assigned the same identity hash code, so that the two objects would be indistinguishable during analysis. We consider this to be an acceptable tradeoff compared to a more complex approach that involves tracking objects.

When recording call chains, we also refer to individual Java methods. Like for classes, HotSpot stores the metadata of Java methods in data structures with constant addresses, which we use as unique method identifiers. When we encounter a method for the first time, we write a *method metadata event* with its identifier, the identifier of the method's class, the method's name, and the method's signature. We also use a newly introduced field to mark methods that we have already defined in the trace.

Although the thread which first encounters an entity records a metadata event for it, some other thread may submit its trace buffer before that thread does. The trace analysis must be able to handle situations where events refer to an entity whose metadata event has not been processed yet, and must be able to resolve such references later.

Call Chains

We consider call chains to be vital for understanding locking bottlenecks, but walking the stack and storing them is expensive. Therefore, we have devised several techniques to record call chains more efficiently.

In a typical application, the number of call chains which use locks is limited, and many of the events that we record share identical call chains. To reduce the amount of data, we maintain a hash set of encountered call chains. When we record a call chain for an event, we look it up in that set. If it does not exist in the set, we assign a unique identifier to it, insert it into the set, and write a *call chain metadata event* with its identifier and its methods. If the call chain already exists in the set, we can just record its identifier in our event. We compute the hash code of a call chain from the instruction pointers of its methods.

Because multiple threads can access the set of call chains concurrently when recording events, those accesses require synchronization. We minimized the risk that the hash set becomes a synchronization bottleneck by implementing its operations in a *lock-free* way: when we record a call chain, we first walk the collision chain for its hash code without using any synchronization. If the call chain is found, we simply use its identifier. Otherwise, we generate a unique identifier for the call chain, and attempt to insert the call chain into the collision chain using an atomic compare-and-set operation. If that operation succeeds, the insertion was successful and we record a call chain metadata event. If the compare-and-set fails, some other thread has inserted a call trace in the same collision chain, so we start over and check if the collision chain now contains the call chain in question. Therefore, we keep the overhead of insertion to a minimum, and looking up an existing call chain incurs no synchronization overhead at all.

We also optimized the stack walk itself. A JIT-compiled Java method usually has several of its callees inlined into its compiled code, and analyzing the stack frame of such a method typically entails resolving which methods are inlined at the current execution position. In HotSpot, this requires decoding compressed debugging information from the compiler. In order to avoid that, we perform light-weight stack walks which do not resolve the inlined methods, and also store call chains without the inlined methods in our hash set. We resolve the inlined methods only for the call chain metadata event that we write when we encounter a new call chain.

Finally, we devised a technique to reuse parts of a call chain that were recorded earlier in the same thread. We derived this technique and its implementation from our work on Incremental Stack Tracing, which we described in Section 3.3.3. When we walk the stack of a thread to construct its call chain, we cache the resulting call chain in a thread-local structure. We also mark the stack frame that is below the top frame by replacing its return address with the address of a code snippet, and retain the original return address in a thread-local structure. When the marked frame returns, the code snippet is executed and simply jumps to the original return address, and the marking is no longer present. However, as long as the marked frame does not return, we can be certain that the frames below it have not changed. When we walk the stack again later and encounter a marked frame, we can stop the stack walk and complete the call chain using the frames of the cached call chain. This technique is intended to reduce the overhead of stack walks when recording multiple events in the same method, such as a contended enter event and a contended exit event, or multiple park begin events.

Unloading of Classes and Compiled Methods

We refer to classes and methods by using the constant addresses of their metadata structures as identifiers. However, when HotSpot's garbage collector detects that a class loader has become unreachable, it unloads all classes loaded by that class loader and reclaims the memory occupied by their metadata. When other metadata is loaded into the same memory, the addresses that we used as identifiers in earlier events can become ambiguous during the analysis of the trace.

Therefore, we need to record when identifiers become invalid. We extended the class metadata event to include the class loader, which we also identify by the address of its metadata structure. When a class loader is unloaded during garbage collection, the application is at a *safepoint,* so all application threads are suspended and cannot write trace events. At this point, we first reclaim the trace buffers of all threads, which can still contain references to classes that are about to be unloaded, and add them to the processing queue to ensure that they are processed first. Then we acquire a new buffer, write a single *class loader unload event* with the identifier of the class loader, and immediately submit the buffer to the processing queue. When the trace analysis processes this event, it discards all class, method and call chain metadata that refers to the unloaded classes. Finally, we let HotSpot unload the classes.

Compiled methods are also frequently unloaded, for example when assumptions that were made during their compilation turned out to be wrong. Other code can then be loaded into the same memory. Because we store addresses of compiled methods in our call chains, these addresses can then become ambiguous, so we purge call chains that refer to unloaded methods from our set of encountered call chains.

4.3 Trace Analysis

In order to identify synchronization bottlenecks effectively, we need to compute meaningful statistics from the recorded events. We accomplish this in two phases: first, we correlate events from different threads with each other to identify individual lock contentions. In the second phase, we aggregate these contentions by user-defined criteria.

4.3.1 Correlation and Contention Analysis

Figure 4.7 shows the process of extracting contentions from a trace. The *event parser* processes one trace buffer at a time. It parses the events in the buffer and forwards each event to the metadata resolver. The *metadata resolver* extracts all metadata from metadata events and keeps them in data structures. It replaces the metadata identifiers in all types of events with references to those data structures so that later phases can access those data structures directly. The metadata resolver then passes the events to the *event rearranger,* which reorders them according to their sequence numbers. The event rearranger maintains one queue per intrinsic lock (IL_i) and passes each intrinsic lock's events in their correct order to the *intrinsic lock dispatcher.* For park/unpark events with a global sequence number, the event rearranger uses a single queue (G) and passes the events to the *park thread dispatcher.*

The *intrinsic lock dispatcher* creates a *lock analyzer* for each intrinsic lock that it encounters in the events, and passes the events of that lock to its analyzer. The analyzer replays the events and keeps track of which threads were blocked and which thread held the lock, and finally generates *contention objects (C).* These contention objects store the duration of the contention, the thread that was blocked, the thread's call chain, the lock's associated object, and that object's class. Most importantly, the contention objects also store the cause of the contention, that is, the thread that held the lock and that thread's call chain. These contention objects are then submitted to aggregators that compute statistics, which we describe in Section 4.3.2.

For the park/unpark mechanism, the analysis is more complex. The *park thread dispatcher* creates a *thread analyzer* for each thread that parked or was unparked, and forwards the events of that thread to its analyzer. The thread analyzer replays the events and creates bundles of related *park begin, unpark* and *park end* events. It submits those bundles to the *park blocker dispatcher,* which creates a *blocker analyzer* for each blocker object that occurs in those bundles. We implemented different types of blocker analyzers to handle the different synchronization semantics of java.util.concurrent classes. The park blocker dispatcher chooses which type of blocker analyzer to create based on the blocker object's class. The blocker analyzer examines the event bundles that are passed to it, tracks the state of the blocker, and creates contention objects that it submits to the aggregator hierarchy. We have implemented blocker analyzers for *ReentrantLock* and for *ReentrantReadWriteLock.* ReentrantLock is very similar to intrinsic locks, and its analyzer processes events as described in the discussion of Figure 4.6. With a ReentrantReadWriteLock, multiple readers can share the lock at the same time without calling park or unpark, so we do not record events for those readers. Only the last reader that releases the lock calls unpark on a blocked writer and records an event. Therefore, our analyzer for ReentrantReadWriteLock cannot determine all readers which blocked a writer, but it still accurately determines which writers blocked which readers, and which writers blocked each other. We have also implemented a generic analyzer for otherwise unsupported blocker objects or when no blocker object is provided. This analyzer keeps track of the time that was spent parking, but cannot determine the owner of the blocker.

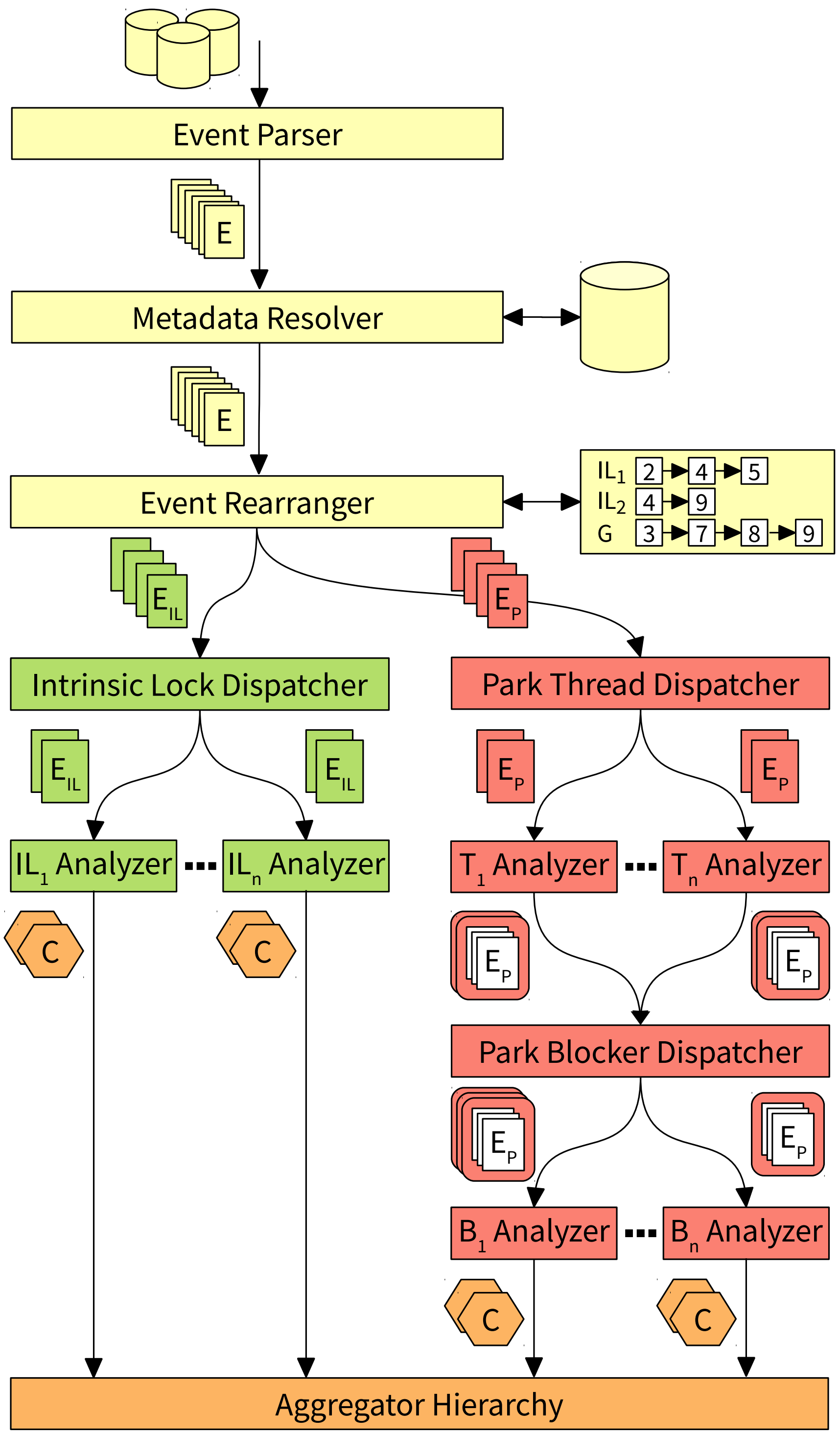


Figure 4.7: Processing events to identify contentions

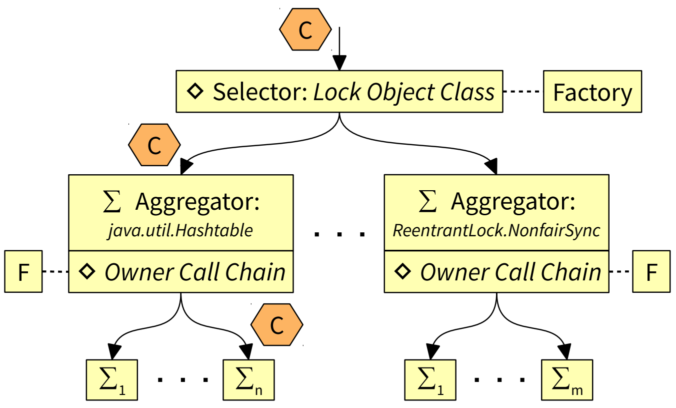

C
◇ Selector: Lock Object Class
Factory
C
Σ Aggregator: java.util.Hashtable
◇ Owner Call Chain
F
Σ Aggregator: ReentrantLock.NonfairSync
◇ Owner Call Chain
F
C
Σ_1
Σ_n
Σ_1
Σ_m

Figure 4.8: Aggregating events

4.3.2 Aggregation of Contentions

To enable users to find and examine locking bottlenecks in an effective way, we devised a versatile method to aggregate the individual contentions by different aspects. These aspects of contentions are the contending thread, the contending call chain (or method), the lock owner thread, the lock owner's call chain (or method), the lock object (an intrinsic lock's associated object, or a park blocker object), and that object's class. As another aspect for aggregation, we categorize contentions into groups for intrinsic locks and for park/unpark synchronization. The user selects one or more of these aspects in a specific order. We then build a hierarchy of selectors and aggregators that break down all contentions by those aspects. An *aggregator* computes the total duration of all contentions that it processes. A *selector* distinguishes contentions by a specific aspect and forwards them to a specific aggregator according to their values.

Figure 4.8 shows an example in which contentions are first distinguished and aggregated by the *lock object's class*, and then by the *lock owner's call chain*. When the trace analysis produces a contention object, it submits it to the hierarchy's *root selector*, which distinguishes contentions by the lock object's class. The selector has a factory that it uses to create an aggregator for each distinct lock object class that it encounters. Assuming that the submitted contention has *java.util.Hashtable* as its lock object class, the selector forwards the contention to the aggregator for that class. The aggregator then adds the contention's duration to its total duration. Because we aggregate by the lock owner's call chain next, each aggregator on this hierarchy level is coupled with a selector that distinguishes a contention by the lock owner's call chain after the aggregator has processed it. The selector also has a factory (denoted by F), which it uses to create aggregators for each encountered call chain and then forwards contentions to them. Assuming that the contention matches the call chain of aggregator Σ_n, the selector forwards the contention to that aggregator, which then adds the contention's duration to its total duration. Because there are no more aspects to aggregate by, the aggregators on that level are not coupled with selectors. The final result of aggregating multiple contentions is the tree of aggregators and the total contention times that they computed. In this example, a user might recognize from that tree and its contention times that there is significant contention with *Hashtable* object locks, and that there are two call chains that cause most of these contentions. The user could then choose to optimize the code that these two call chains execute, or also select a different data structure or implementation, such as ConcurrentHashMap.

4.3.3 Interactive Visualization

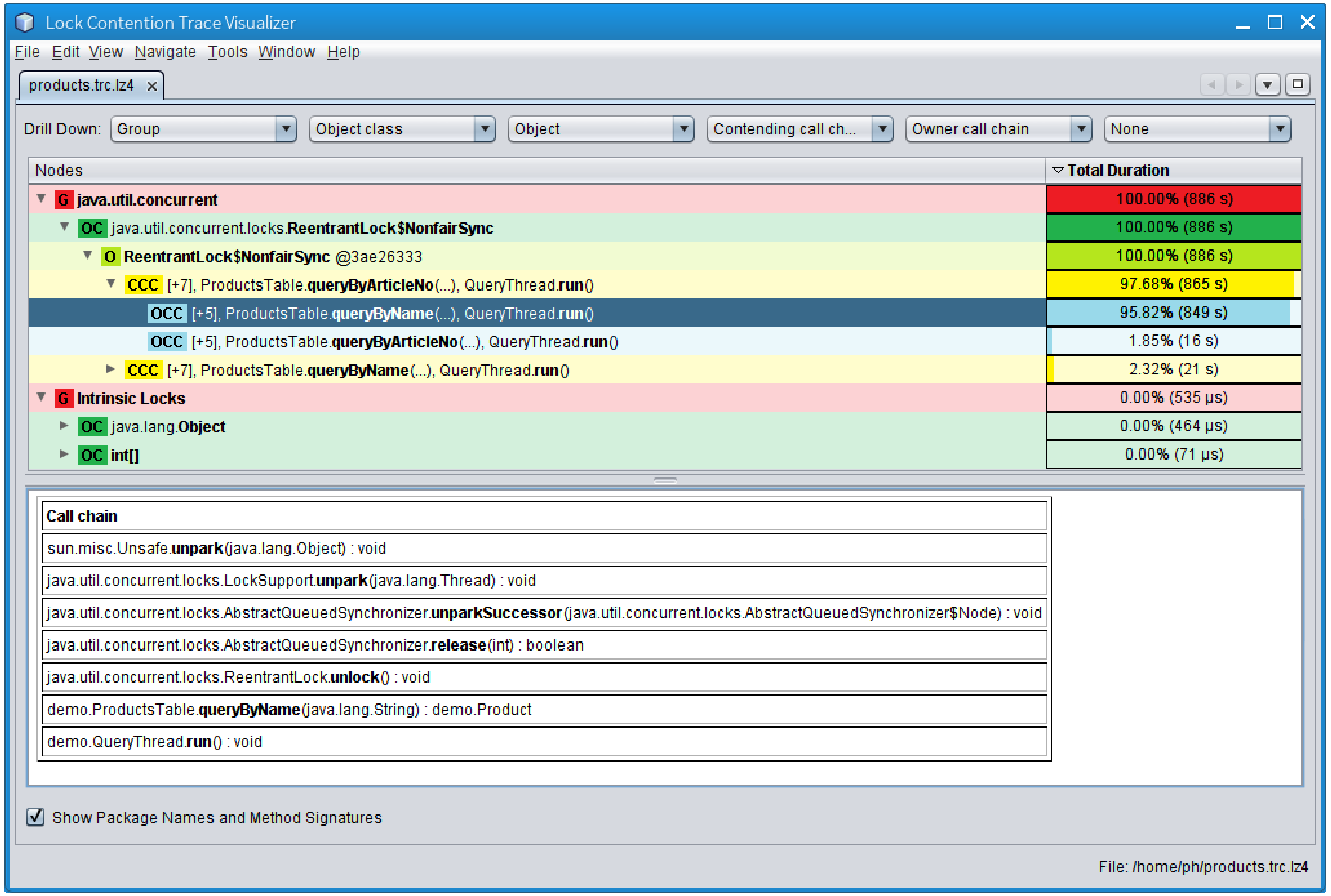
In order to enable users to perform a comprehensive offline analysis of the generated traces, we built an interactive visualization tool. We demonstrate this tool with a simple application that maintains an in-memory hash table of products. Multiple threads continuously query that table for individual products, for which the threads hold a lock to guard against potential concurrent changes to the table. The vast majority of queries use a product's number to retrieve its details, which requires only an inexpensive hash lookup. However, 1% of the queries search for a product by its name, which entails iterating over all products until a match is found, also while holding the lock.

Figure 4.9 shows a screen capture of our visualization tool displaying a trace file from this application. The main window is divided into three parts: a drill-down selection panel at the top, an aggregation tree in the center, and a detail view for the selected entry in the aggregation tree at the bottom. In this example, the drill-down panel is configured to aggregate contentions first by group, then by the lock object's class, next by individual lock objects, then by the contending thread's call chain, and finally by the lock owner's call chain. Therefore, the root level of the tree displays the different groups that we categorize events in. The first entry represents contentions from java.util.concurrent synchronization, and the *Total Duration* column displays that it makes up 100% of all contentions, an absolute value of 886 seconds. The second entry on the root level represents contentions from intrinsic locks, which is negligible, so we do not describe it in greater detail.

The entries on the next two tree levels break down the contended time by lock object classes and further by individual objects. They show that all of the java.util.concurrent contention comes from locking *ReentrantLock$NonfairSync* objects, specifically from a single instance with identity hash code *3ae26333*. The tree level below that displays the call chains where the threads spent time waiting for the lock and reveals that 97.68% of the contended time was spent in method *queryByArticleNo*, while only 2.32% of the time was spent waiting in *queryByName*. Such information is also provided by commonly available tools, and a developer might conclude that because contention primarily affected *queryByArticleNo*, optimizing it could reduce lock contention. However, the lock owner call chains that are displayed on the next tree level reveal that when threads waited for the lock in *queryByArticleNo*, the contention was almost exclusively caused by *queryByName* which was holding the lock. Therefore, optimizing *queryByName* is likely to be more effective for reducing lock contention. This could be accomplished with a fast index data structure for product names, and also by using a read-write lock that allows multiple concurrent queries. Note that because call chains are typically too long to fit into a single line, our tool shows *[+n]* to denote that *n* calls have been omitted. However, a user can select a specific entry to view the entire call chain in the detail view below the tree. In this example, this reveals implementation details of ReentrantLock using *AbstractQueuedSynchronizer*, which our tool always omits in the tree for brevity.

As an example of using our visualization tool on a more sophisticated application, Figure 4.10 shows a screen capture of our tool displaying a trace file from the DaCapo suite's *avrora* benchmark, which simulates a network of microcontrollers. In this example, the drill-down panel is configured to aggregate contentions by group, then by the lock object's class, next by individual lock objects, then by the lock owner's method, and finally by the lock owner's call chain. The aggregation tree shows that all of the contentions involve intrinsic locks, with a total duration of 37 seconds. The entries on the second and third tree level indicate that 99.78% of the contention comes from locking a single *Class* instance. The two tree levels below that display the owner methods and the owner call chains. With 99.32%, *SimPrinter.printBuffer* is almost always the owner of the lock when a contention occurs. The multiple owner call chains show that this method is called from more than one location, and that the amount of caused contention varies significantly by call chain.

We examined the source code of avrora and found that *SimPrinter.printBuffer* is used to log simulation events and that it calls certain static methods of class *Terminal.* To avoid that the output from different threads is interleaved, printBuffer acquires the intrinsic lock of the *Class* object of Terminal, which poses a locking bottleneck. In Figure 4.10, we see two call chains that cause 72.75% and 11.41% of all contention, and in both of them, printBuffer is called by *fireAfterReceiveEnd*. This method logs when a simulation node receives a network packet, which is the most frequently logged event, and its output is very large because it includes a hexadecimal representation of the packet's data. To mitigate this bottleneck, contention could be reduced by logging fewer details or fewer events, or also by queueing log events and asynchronously writing them to a file in a background thread.


Lock Contention Trace Visualizer
File Edit View Navigate Tools Window Help
products.trc.lz4

Drill Down: Group | Object class | Object | Contending call ch... | Owner call chain | None

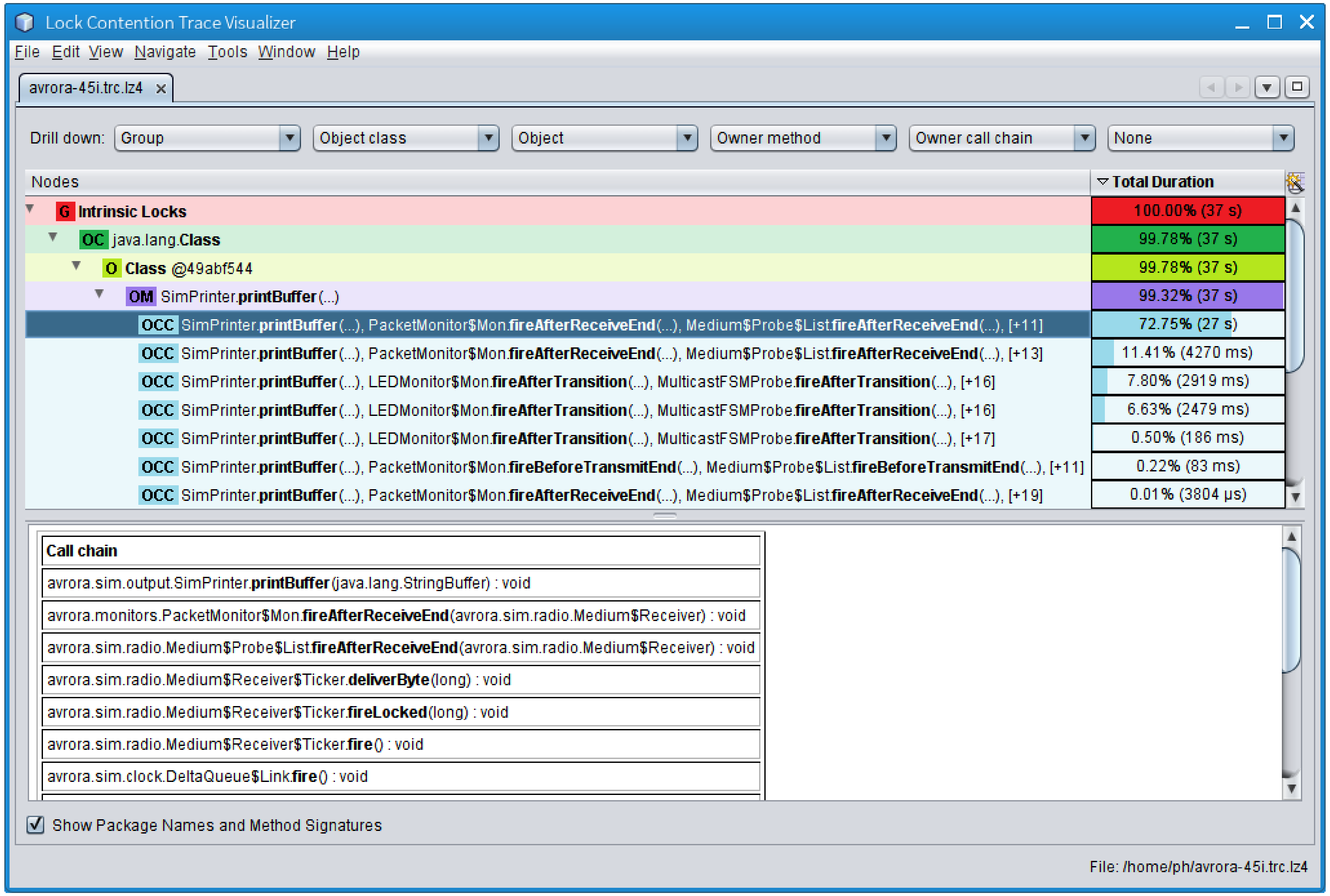
Nodes	Total Duration
G **java.util.concurrent**	100.00% (886 s)
OC java.util.concurrent.locks.**ReentrantLock$NonfairSync**	100.00% (886 s)
O **ReentrantLock$NonfairSync** @3ae26333	100.00% (886 s)
CCC [+7], ProductsTable.**queryByArticleNo**(...), QueryThread.**run**()	97.68% (865 s)
OCC [+5], ProductsTable.**queryByName**(...), QueryThread.**run**()	95.82% (849 s)
OCC [+5], ProductsTable.**queryByArticleNo**(...), QueryThread.**run**()	1.85% (16 s)
CCC [+7], ProductsTable.**queryByName**(...), QueryThread.**run**()	2.32% (21 s)
G **Intrinsic Locks**	0.00% (535 µs)
OC java.lang.**Object**	0.00% (464 µs)
OC **int[]**	0.00% (71 µs)

Call chain
sun.misc.Unsafe.**unpark**(java.lang.Object) : void
java.util.concurrent.locks.LockSupport.**unpark**(java.lang.Thread) : void
java.util.concurrent.locks.AbstractQueuedSynchronizer.**unparkSuccessor**(java.util.concurrent.locks.AbstractQueuedSynchronizer$Node) : void
java.util.concurrent.locks.AbstractQueuedSynchronizer.**release**(int) : boolean
java.util.concurrent.locks.ReentrantLock.**unlock**() : void
demo.ProductsTable.**queryByName**(java.lang.String) : demo.Product
demo.QueryThread.**run**() : void

[✓] Show Package Names and Method Signatures

File: /home/ph/products.trc.lz4

Figure 4.9: Our interactive lock contention trace visualization tool


Lock Contention Trace Visualizer
File Edit View Navigate Tools Window Help
avrora-45i.trc.lz4
Drill down:
Group
Object class
Object
Owner method
Owner call chain
None
Show Package Names and Method Signatures

Nodes	Total Duration
G **Intrinsic Locks**	100.00% (37 s)
OC java.lang.**Class**	99.78% (37 s)
O **Class** @49abf544	99.78% (37 s)
OM SimPrinter.**printBuffer**(...)	99.32% (37 s)
OCC SimPrinter.**printBuffer**(...), PacketMonitor$Mon.**fireAfterReceiveEnd**(...), Medium$Probe$List.**fireAfterReceiveEnd**(...), [+11]	72.75% (27 s)
OCC SimPrinter.**printBuffer**(...), PacketMonitor$Mon.**fireAfterReceiveEnd**(...), Medium$Probe$List.**fireAfterReceiveEnd**(...), [+13]	11.41% (4270 ms)
OCC SimPrinter.**printBuffer**(...), LEDMonitor$Mon.**fireAfterTransition**(...), MulticastFSMProbe.**fireAfterTransition**(...), [+16]	7.80% (2919 ms)
OCC SimPrinter.**printBuffer**(...), LEDMonitor$Mon.**fireAfterTransition**(...), MulticastFSMProbe.**fireAfterTransition**(...), [+16]	6.63% (2479 ms)
OCC SimPrinter.**printBuffer**(...), LEDMonitor$Mon.**fireAfterTransition**(...), MulticastFSMProbe.**fireAfterTransition**(...), [+17]	0.50% (186 ms)
OCC SimPrinter.**printBuffer**(...), PacketMonitor$Mon.**fireBeforeTransmitEnd**(...), Medium$Probe$List.**fireBeforeTransmitEnd**(...), [+11]	0.22% (83 ms)
OCC SimPrinter.**printBuffer**(...), PacketMonitor$Mon.**fireAfterReceiveEnd**(...), Medium$Probe$List.**fireAfterReceiveEnd**(...), [+19]	0.01% (3804 µs)

Call chain
avrora.sim.output.SimPrinter.**printBuffer**(java.lang.StringBuffer) : void
avrora.monitors.PacketMonitor$Mon.**fireAfterReceiveEnd**(avrora.sim.radio.Medium$Receiver) : void
avrora.sim.radio.Medium$Probe$List.**fireAfterReceiveEnd**(avrora.sim.radio.Medium$Receiver) : void
avrora.sim.radio.Medium$Receiver$Ticker.**deliverByte**(long) : void
avrora.sim.radio.Medium$Receiver$Ticker.**fireLocked**(long) : void
avrora.sim.radio.Medium$Receiver$Ticker.**fire**() : void
avrora.sim.clock.DeltaQueue$Link.**fire**() : void

File: /home/ph/avrora-45i.trc.lz4

Figure 4.10: Visualization of a bottleneck in the *avrora* benchmark

Figures 4.9 and 4.10 demonstrate that our tool enables users to see the methods and call chains that caused contentions, unlike common approaches that show only which methods and call chains were blocked. However, using the drill-down panel, users can select other aspects and also choose any desired order of aspects for aggregation. This allows users to extract a wealth of additional information, such as whether some threads caused or suffered more contention than other threads, which call chains held the lock when a specific call chain was blocked and for how long, or which contended locks were used by a specific thread, method, or call chain.

4.4 Evaluation

We implemented our approach for OpenJDK 8u45 and evaluated it with synthetic workloads from a purpose-built test suite, as well as with real-world benchmarks.

4.4.1 Synthetic Workloads and Correctness

In order to verify that our approach accurately depicts the locking behavior of an application, we devised a test suite that generates predictable synthetic locking workloads. We built this test suite using the Java Microbenchmark Harness [JMH15]. In our tests, we vary the number of threads and the number of call chains. Most importantly, we vary how long the different threads or call chains hold a lock, which changes how much contention each of them causes. We implemented those tests for intrinsic locks and for java.util.concurrent locks and found that our generated traces match the expected amounts of contention for each test. Moreover, we verified that the recorded call chains are correct when the code throws exceptions, when the compiler inlines code, and when the code uses *wait* and *notify*.

4.4.2 Benchmarking

We evaluated the runtime overhead, the amount of trace data and other aspects of our approach with the DaCapo 9.12 benchmark suite [Blackburn06] and with the benchmarks of the Scala Benchmarking Project 0.1.0 [Sewe11] according to the same methodology which we first described in Section 3.4.1.

We performed all tests on a server-class system with two Intel Xeon E5-2670v2 processors with ten cores each and with hyperthreading, and thus, 40 hardware threads. The system has a total of 32 GB of memory and runs Oracle Linux 7. To get more reproducible results, we disabled dynamic frequency scaling and turbo boost, and we used a fixed Java heap size of 16 GB. With the exception of system services, no other processes were running during our measurements.

We executed 45 successive *iterations* of each benchmark in a single HotSpot instance, and discarded the data from the first 35 iterations to compensate for the duration of HotSpot's startup phase that we observed on this hardware. We reinitialized event tracing at the start of each iteration. We further executed more than 10 *rounds* of each benchmark (with 45 iterations each) to ensure that the results are not biased by factors that are determined during HotSpot's startup phase.

4.4.3 Runtime Overhead

We measured the benchmark execution times with tracing when writing an uncompressed output file, when writing a compressed output file, and when analyzing the events online. The online analysis executes in parallel to the benchmark and aggregates the contentions by lock object class, then by contending call chain, and then by the lock owner's call chain. We compare these execution times to those of an unmodified OpenJDK 8u45 without tracing.

Figure 4.11 shows the median execution times of each benchmark, normalized to the median execution times without tracing. The error bars indicate the first and third quartiles. We categorized the benchmarks into multi-threaded and single-threaded benchmarks. The *G.Mean* bars show the geometric means of each category, and their error bars indicate a 50% confidence interval. For the multi-threaded benchmarks, the mean overhead of generating a trace file is 7.8%, both with and without compression. This shows that the overhead of the compression is negligible. With online analysis, the mean overhead is 9.4%. For the single-threaded benchmarks, the mean overhead of generating a compressed trace file is 0.8%, and without compression and with online analysis, it is 1.2%. The overhead for single-threaded benchmarks is caused in part by the trace buffer management, and in part because the JDK itself uses multi-threading and synchronization, which we trace as well.

The overheads of the individual benchmarks correlate directly with the amount of contention that they exhibit. The benchmarks with the highest overhead are actors and xalan. *actors* is a concurrency benchmark with many fine-grained interactions between threads, and tracing them results in an overhead of 22%. Online analysis increases it to 42%, which is caused by the benchmark's extensive use of java.util.concurrent synchronization, for which the analysis is more complex and thus slower. The *xalan* benchmark transforms XML documents. It distributes work to as many threads as there

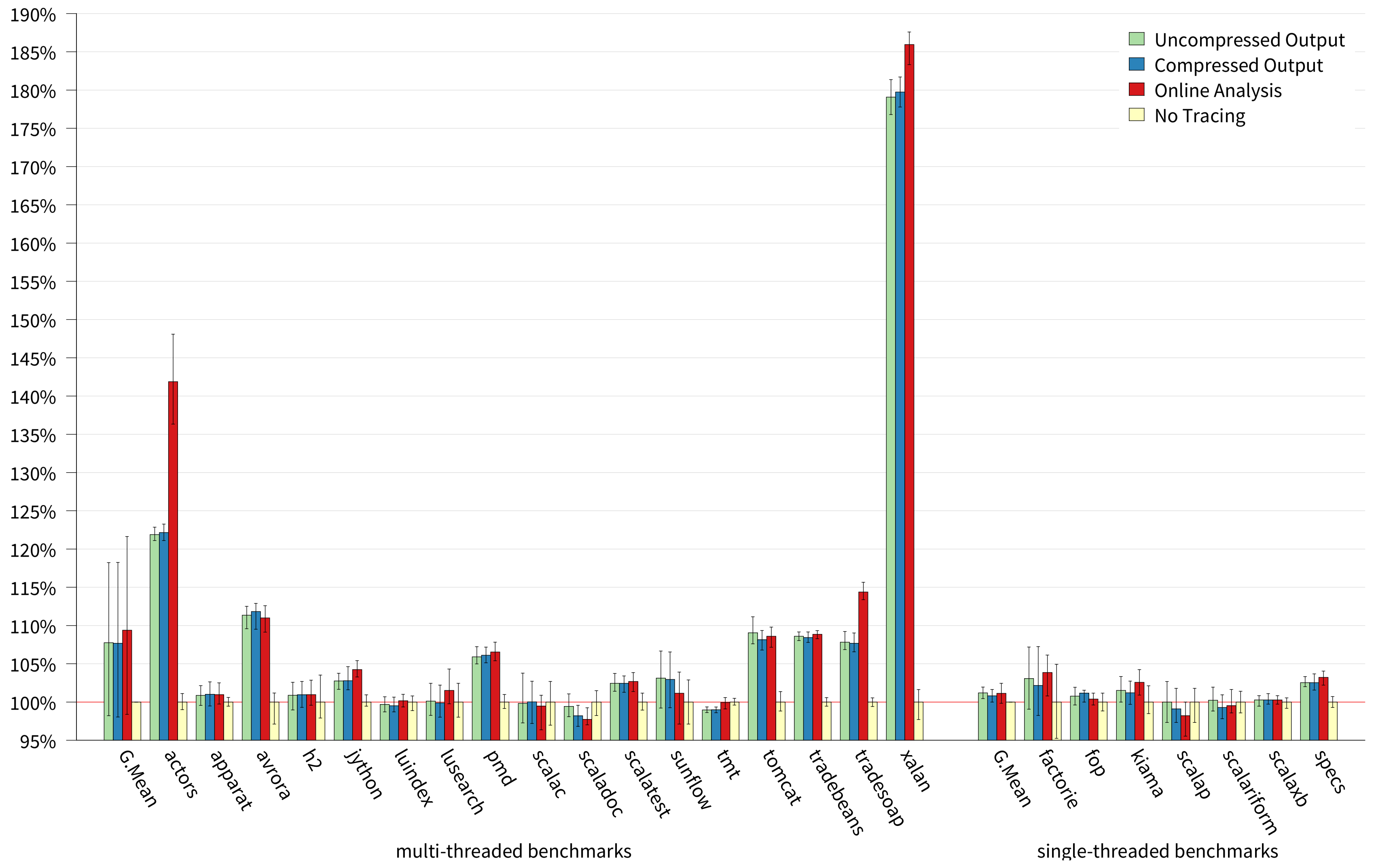


Figure 4.11: Overhead of tracing with uncompressed output, with compressed output, and with online analysis, relative to no tracing

are hardware threads, but all threads access a single Hashtable instance. This causes a substantial number of short contentions on our machine with 40 hardware threads, and tracing them incurs an overhead of around 80%.

For some benchmarks such as *scaladoc* or *tmt,* tracing even slightly improves the benchmark's performance. We attribute this to the delays that are introduced by recording events and call chains. David et al. found that HotSpot's locks saturate the memory bus, and that delays in lock acquisition reduce memory bus contention, which can increase performance [David14]. Also, after we write a contended enter event, HotSpot's implementation of intrinsic locks retries to spin-acquire the lock while it adds the thread to the queue of blocked threads. The delay of writing the event before spinning can increase its chances of being successful instead of suspending the thread. Additionally, the activity of the background thread in which we write the trace file or analyze the trace data influences the behavior of the garbage collector and of the thread scheduler, which can also have minor effects on performance.

The runtime overhead of our tracing is well below 10% for all but three benchmarks. We consider this to be feasible for monitoring a production system. On a quad-core workstation, we measured even lower mean overheads below 3%.

4.4.4 Generated Amount of Data

The amount of generated trace data is also an important factor for production use. Figure 4.12 shows the mean amount of generated trace data per second for the multi-threaded benchmarks, with and without compression. Tracing *xalan, actors* and *avrora* generates the most uncompressed trace data at 26.3 MB/s, 14.7 MB/s, and 8.6 MB/s, respectively. For the other benchmarks, our approach generates less than 6 MB of uncompressed data per second. For those benchmarks that do not exhibit significant contention, we record less than 50 KB per second. Our on-the-fly compression typically reduces the amount of data by between 60% and 70%, and decreases the data rate of *xalan* to around 10 MB/s. Therefore, 60 minutes of trace data from a *xalan*-type application require less than 40 GB of disk space and should be more than sufficient to analyze performance problems. Nevertheless, the compression adds only negligible runtime overhead.

We also inspected the memory footprint of the trace buffers, which have a mean capacity of 16 KB. We found that we always use fewer than 500 buffers at any time, and hence occupy less than 8 MB of memory with trace buffers.

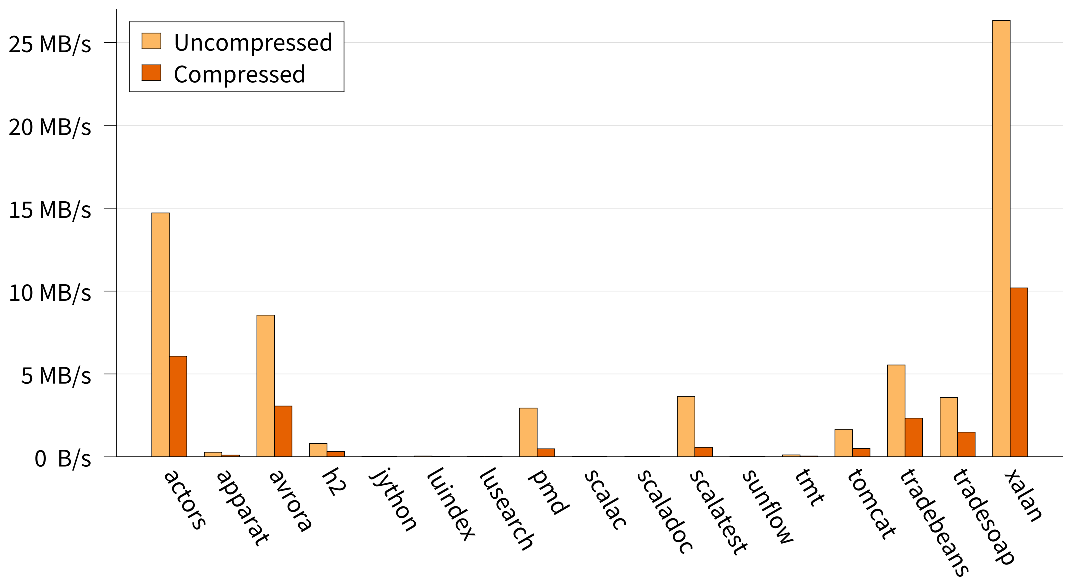


Figure 4.12: Trace data generated per second

4.4.5 Trace Composition

We further examined the composition of the generated traces. Figure 4.13 shows the relative frequencies of individual events for the multi-threaded benchmarks. We grouped all types of metadata events for brevity. The *actors* and *apparat* benchmarks are the only ones that predominantly rely on java.util.concurrent synchronization, with *tmt, tomcat, tradebeans* and *tradesoap* using it to some extent. As would be expected, there are typically equal numbers of park begin events, park end events, and unpark events. Surprisingly, we record only unpark events for some benchmarks, such as *jython.* This is because these benchmarks call the *Thread.interrupt* method, which always implicitly performs an unpark operation, regardless of whether the interrupted thread is currently parked.

With intrinsic locks, we record an equal number of contended enter and contended entered events, but always a significantly higher number of contended exit events. The difference is particularly large with benchmarks that acquire locks only very briefly, such as *luindex.* The reason for that is non-fair locking, with which a thread can instantly acquire an available lock even when there are queued threads, but when that thread releases the lock, it must write a contended exit event. When one of the queued threads has already been resumed and cannot acquire the lock, we do not write any additional events in that thread. With park-based synchronization, we would record another park begin event and park end event in that case, which is why the number of the three types of park events is more balanced.

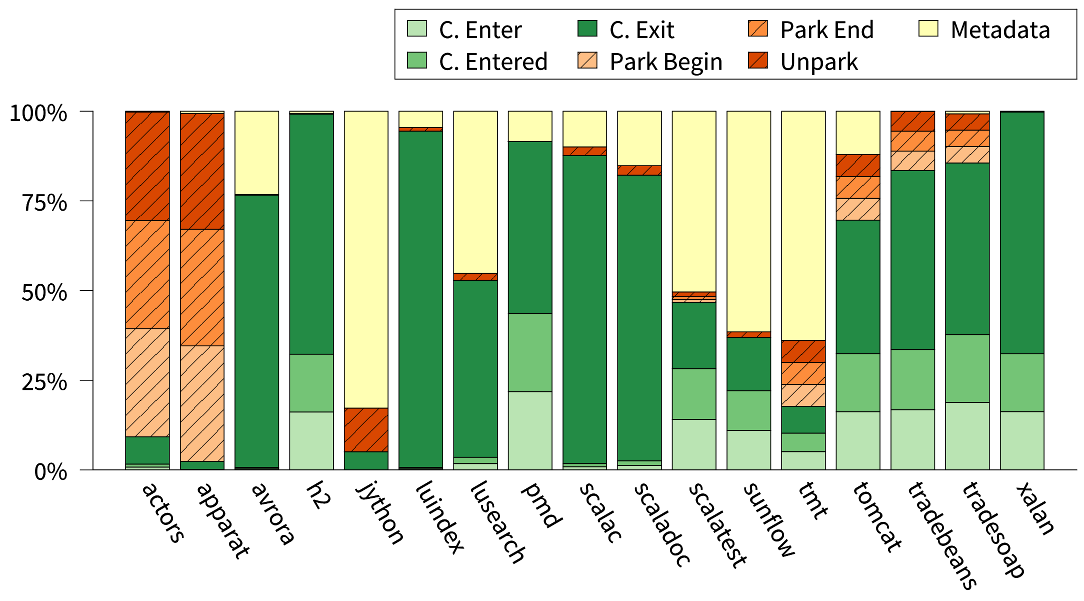


Figure 4.13: Frequency of trace events

For some benchmarks, metadata constitutes a relatively large portion of the trace data. Most of these benchmarks exhibit low contention, so that the amount of metadata that we record for that contention becomes relevant. In contrast to those benchmarks, *scalatest* and *tomcat* exhibit significant contention, but they generate many Java classes at runtime which use locks, so we record a large number of different call chains. *avrora* heavily uses conditional waiting with intrinsic locks, and to collect correct call chains, we use forward references to call chains and must record an additional metadata event to resolve these forward references when a lock is finally released.

4.4.6 Call Chain Optimizations

In order to assess how much of our overhead comes from recording call chains, we also measured the tracing overhead when not recording call chains. Figure 4.14 compares the overheads of tracing with and without recording call chains for the multi-threaded benchmarks when writing an uncompressed trace file. When not recording call chains, the mean overhead decreases from 7.8% to 6.4%. Hence, our method of recording call chains typically constitutes less than 20% of the tracing overhead. The overhead for tracing *xalan* (which is cut off in the chart) is reduced from 79% to 73%.

We further examined the effectiveness of reusing call chains. In all benchmarks for which we recorded more than 10,000 call chains per second, we were able to reuse more than 99.5% of all call chains from the set of encountered call chains (*pmd* is the sole exception

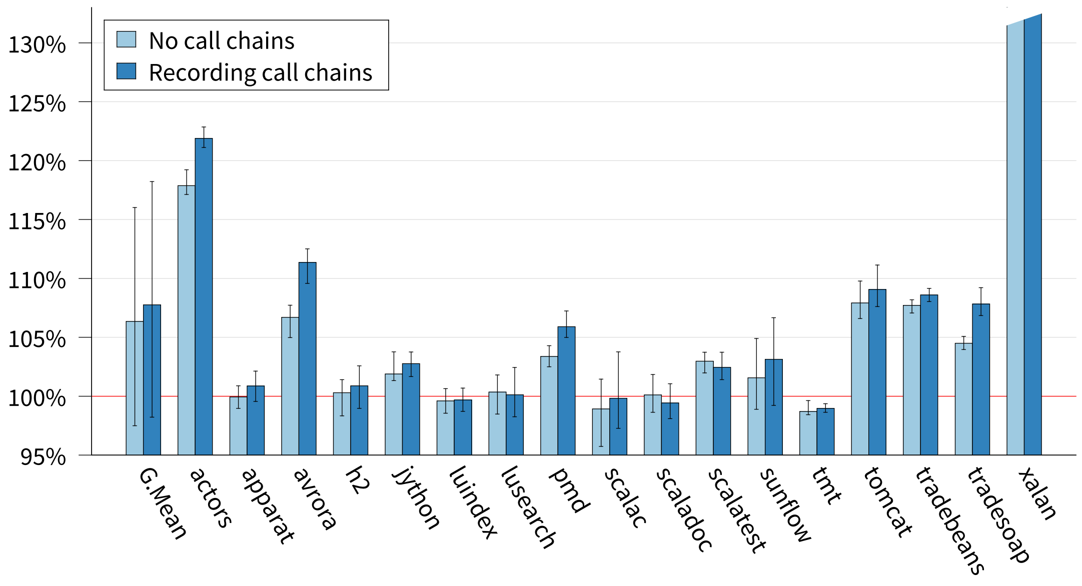


Figure 4.14: Overhead of recording call chains

at 89%). This reduces the amount of metadata in the trace and requires only light-weight stack walks for the lookups in the set. With our technique of marking stack frames, we further save examining between 10% and 50% of all stack frames for those same benchmarks.

4.5 Related Work

Tallent et al. describe a sampling profiler which, like our approach, identifies which threads and call chains block other threads and call chains by holding a contended lock [Tallent10]. The profiler associates a counter with each lock and periodically takes samples. When it samples a thread that is blocked on a lock, it increases that lock's counter. When a thread releases a lock, the thread inspects the lock's counter, and if it is non-zero, the thread "accepts the blame" and records its own call chain. The profiler was implemented for C programs and is reported to have an overhead of 5%, but determines *only* which threads and call chains blocked other threads. Our approach also records which threads and call chains were blocked by a specific thread or call chain, which we consider to have diagnostic value when reasoning about performance problems that *occur* in a specific part of an application.

David et al. propose a profiler that observes *critical section pressure (CSP)*, a per-lock metric which correlates the time that threads spend waiting to acquire the lock to the time they spend executing and making progress [David14]. Phases when a lock exhibits a high CSP indicate that the lock is a bottleneck. Therefore, the profiler continuously computes the CSP of each lock over one-second periods and when a lock's CSP exceeds a threshold, the profiler records the identity of the lock and a call chain from one thread that was blocked. The profiler was implemented in HotSpot and is reported to have a worst-case overhead of 6%. We consider this approach complementary to ours because the CSP of locks can be determined from the events that we record.

Patros et al. describe a VM-internal profiler for java.util.concurrent contention [Patros15]. The profiler assigns a data structure to each park blocker object to store various metrics, such as how often threads were parked on that object. When the metrics of a blocker object reach a threshold, the profiler records the object's identity and the name and the call chain of the next thread that parks. The profiler then begins capturing additional metrics for that blocker object, including the average and total times that threads spend parking on that object. The profiler was implemented in the IBM JVM and is claimed to have negligible overhead, but captures only one thread and call chain per blocker object, and also does not determine which threads or call chains cause contention.

Inoue and Nakatani describe a sampling profiler in a Java VM which injects special instructions at lock acquisition sites [Inoue09]. It then uses hardware performance counters to capture where the application acquires locks and where it blocks. The profiler constructs call chains with a probabilistic method that uses the stack depth. It was implemented in the IBM JVM and is claimed to have an overhead of less than 2.2%, but it does not determine which threads and call chains block other threads and call chains.

Java Flight Recorder (JFR, [Hirt10]) is a commercial feature of the Oracle JDK that efficiently records performance-related events in an application. It collects information on blocked threads, their call chains, and the classes of lock objects. To keep the overhead low, JFR records only long contentions (more than 10ms) by default. JFR also records which thread most recently owned a lock before it could be acquired after a contention. However, although more than one thread can own a lock over that time, it considers the entire time that is spent blocking to be caused by that thread. Unlike our approach, JFR does not record call chains for threads that block other threads. It also only provides contention statistics for intrinsic locks and not for java.util.concurrent locks.

JVMTI also provides functionality to observe contention from intrinsic locks. Profilers can register callbacks for contention events and can then examine the thread, the lock's associated object, and the call trace of the blocked thread. However, those events cannot be used to determine the thread which holds the lock and therefore caused the contention, or that thread's call chain. JVMTI also does not provide events to observe contention from java.util.concurrent locks.

Stolte et al. describe *Polaris*, a system for analyzing and visualizing data in multidimensional databases [Stolte02]. Polaris provides a visual query language for generating a range of graphical presentations, which enables users to rapidly explore the data. We believe that such a system would complement our visualization tool, and we consider implementing an export feature for our trace analyis results to a format that can be used with such systems.

4.6 Summary and Future Work

In this chapter, we presented a novel approach for analyzing locking bottlenecks in Java applications by efficiently tracing lock contention in the Java Virtual Machine. We trace contention from both Java's intrinsic locks and from java.util.concurrent locks. For the analysis of the traces, we devised a versatile approach to aggregate and visualize the recorded contentions. Unlike other methods, our approach shows not only where contention occurs, but also where contention is caused. Nevertheless, our implementation in HotSpot incurs a low mean overhead of 7.8% on a server-class system with 40 hardware threads, so we consider it feasible to use our approach for monitoring production systems.

Future work could focus on situations where our approach has higher overhead, in particular when tracing a substantial number of short contentions. Such situations could be addressed by supporting enabling and disabling tracing at runtime, which would allow users to analyze lock contention on a production system on demand, while causing little to no overhead when tracing is not active. Another possible direction for future work is to focus further on the activities of lock owner threads while their locks are contended, for example by taking periodic samples of their call chains. This could provide users with more detailed information on where to apply optimizations to hold locks for less time and reduce lock contention. Further research could also attempt to identify connections between lock contentions, for example when a thread holds a contended lock and is then blocked when trying to acquire another lock. Finally, future work could focus on analyzing conditional waiting with both intrinsic locking and java.util.concurrent mechanisms. This could reveal problems such as when threads wait for each other to finish related tasks, but some tasks take significantly longer than others, leaving some threads idle.

Chapter 5

Hardware Virtualization Steal Time Accounting

This chapter characterizes the concept of steal time in hardware virtualization and describes our approach for attributing such time to individual application threads without modifications to the hypervisor or operating system. We evaluate the accuracy and overhead of our approach and compare it to related work.

The previous chapters presented approaches for revealing performance problems that originate in the application's implementation. However, performance problems can also be caused by the environment in which an application is executed. Applications are commonly deployed in virtualized environments, which enable the sharing of hardware resources between applications. When the physical resources are insufficient for the combined workload of all virtualized environments, some virtual processors must be temporarily suspended in favor of others. This results in lost processing time that is incorrectly accounted for and misrepresents the resource usage of an application.

At the outset of this thesis, we identified the correct accounting of virtualization effects as a major challenge of APM software. In this chapter, we present an efficient and accurate approach for how a performance analysis tool running in a virtualized environment can estimate to what degree suspension affects individual application threads. Our approach relies on the virtualization software to provide basic information on suspension. We periodically sample this information as well as information on the resource usage of threads and then account for the suspension. Our results show that we are able to reliably separate true CPU usage and steal time at negligible performance overhead. We published our approach in [Hofer15c]. Part of this work has been done by master's student Florian Hörschläger.

5.1 Background

Hardware virtualization is commonly used to efficiently utilize and share hardware resources. The resources of a physical machine (the *host system*) are shared between several virtual machines (the *guest systems*). The creation and execution of guest systems is managed by a *hypervisor.* Each guest system has its own operating system that runs in an isolated execution domain, which provides reliability and security. Guest systems can also be deployed and moved between sites with less effort than physical hardware.

Guest systems are assigned one or more virtual CPUs to execute on. The number of virtual CPUs of all guest systems can exceed the number of physical CPUs of the host system. In that case, the hypervisor cannot schedule all virtual CPUs on the available physical CPUs and must temporarily suspend some of them. Time periods when a virtual CPU is ready to execute, but is suspended, are referred to as *steal time.*

The amount of steal time is an important indicator for whether the physical resources of the host system are insufficient for the virtual resources that are assigned to the guest systems. However, the operating system in a guest system is often not aware of steal time or does not incorporate it into resource usage accounting because the concept of steal time does not exist with physical hardware. When a virtual CPU is suspended, the steal time is considered active CPU time and counted toward the resource usage of the currently executing thread. Therefore, it is difficult for a performance engineer to spot whether a performance problem is caused by an actual bottleneck in the application or by virtualization.

Hypervisors commonly provide an interface that allows the operating system or other software in a guest system to detect that they are running under virtualization and to interact with the hypervisor. Using such an interface can benefit the performance of the guest system. A hypervisor interface typically also exposes steal time information. However, since the hypervisor has no knowledge of the processes and threads running within the guest system, it attributes steal time only to virtual CPUs. On this level, steal time is only useful to performance analysis tools as an indicator for how the entire guest system is affected by the suspension of virtual CPUs, but not how individual threads are affected.

5.2 Steal Time Accounting

We present a sampling-based approach for how a performance analysis tool can estimate to what degree steal time affects individual application threads in a guest system. Our approach works from within the guest system and breaks down the steal time that the hypervisor reports for the virtual CPUs to the monitored application threads. We base our approach on the fact that those threads which use the most CPU time (or, to which the most CPU time is attributed) are also the ones that are most affected by the suspension of virtual CPUs. Hence, we divide the steal time among the threads in proportion to the CPU time they have consumed.

When a guest system has multiple virtual CPUs, we would ideally determine which threads were executed on each virtual CPU and divide that CPU's steal time among these threads. Unfortunately, common operating systems do not make such scheduling information available. However, bare-metal hypervisors often employ a form of co-scheduling in which all virtual CPUs of a guest system are scheduled to run on physical CPUs at the same time, which avoids problems such as when one virtual CPU waits for some other virtual CPU that is currently suspended by the hypervisor [VMWare13]. With such scheduling, all virtual CPUs of a guest system are similarly affected by hypervisor steal time. Other hypervisors simply rely on the host operating system's scheduler, and a fair scheduler should ensure that one ready virtual CPU of a guest system is not suspended significantly longer than others. Therefore, we sum up the steal time from all virtual CPUs of the guest system and divide it among all threads which consumed CPU time.

The extent of hypervisor suspension in a guest system also depends on the load in the other guest systems as well as on the host system itself and therefore varies over time. For this reason, we use a sampling approach and periodically read or compute the following values to account for the steal time since the last sample:

$\Delta t_{steal,total}$: The total steal time of all virtual CPUs since the previous sample.

$\Delta t_{cpu,total}$: The total apparent CPU time (including steal time) that was consumed by all virtual CPUs of the guest system since the previous sample.

$\Delta t_{cpu}(T)$: The apparent CPU time (including steal time) that was consumed by thread T since the last sample.

The hypervisor and the guest operating system typically make steal times and CPU times available as total times since the start of the guest system or since the start of the thread. Computing the deltas between samples thus requires storing the values that were read in the previous sample. We then use the deltas between samples to divide the total steal time among the monitored threads using the following equation for each thread T:

$$\Delta t_{steal}(T) = \Delta t_{steal,total} \frac{\Delta t_{cpu}(T)}{\Delta t_{cpu,total}}$$

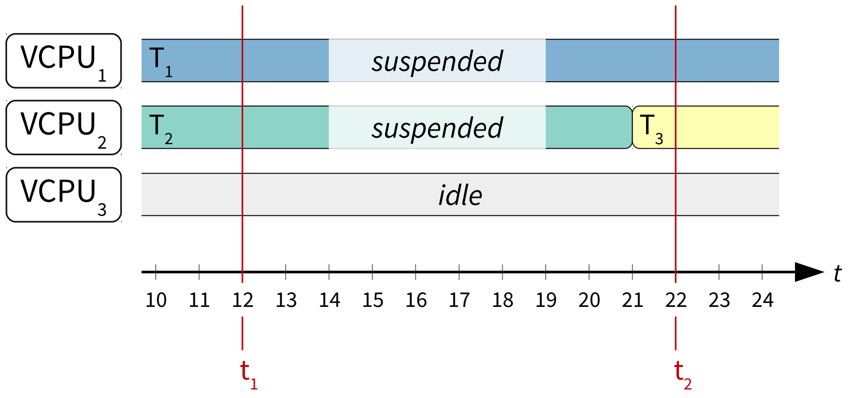


Figure 5.1: Example of a schedule with three threads on three virtual CPUs

The steal time of a process can be computed in the same way if the operating system provides a per-process CPU usage that includes all threads. Alternatively, the steal times of all threads of the process can be computed individually and added up, but this can lead to inaccuracies when threads exit between samples.

Figure 5.1 shows an example of how our approach works in a schedule with three threads, T_1, T_2, and T_3, that are executing on three virtual processors, $VCPU_1$, $VCPU_2$ and $VCPU_3$. Initially, $VCPU_1$ is executing thread T_1, $VCPU_2$ is executing T_2, and $VCPU_3$ is idle. At $t_1 = 12$, we take a first sample. Because we do not have a previous sample, we cannot compute deltas at this point and do not attribute steal time to the threads. At $t = 14$, the hypervisor suspends both $VCPU_1$ and $VCPU_2$ until it resumes them at $t = 19$. At $t = 21$, the operating system schedules out T_2 on $VCPU_2$ and executes T_3. Finally, at $t_2 = 22$, we take a second sample and compute the deltas to the first sample as follows:

$\Delta t_{cpu}(T_1) = 10$ $\Delta t_{cpu}(T_2) = 9$ $\Delta t_{cpu}(T_3) = 1$

$$\Delta t_{cpu,total} = 10 + 9 + 1 = 20$$

$$\Delta t_{steal,total} = 5 + 5 = 10$$

The time during which the virtual CPUs were suspended is attributed as CPU time to the scheduled threads because the operating system does not consider steal time when it does CPU time accounting. Note that $VCPU_3$ was idle, so although it was technically also suspended, the hypervisor did not assign any steal time to it. Our approach now computes the following steal times for the individual threads:

$$\Delta t_{steal}(T_1) = 10\tfrac{10}{20} = 5$$

$$\Delta t_{steal}(T_2) = 10\tfrac{9}{20} = 4.5$$

$$\Delta t_{steal}(T_3) = 10\tfrac{1}{20} = 0.5$$

Therefore, T_1 is assigned the correct amount of steal time. Although T_3 was not affected by suspension, it is assigned a small portion of the steal time because it was scheduled briefly at the end of the sampling period. For the same reason, T_2 is assigned slightly less than the actual amount of suffered steal time. However, we expect that such deviations become insignificant with a sufficient number of samples.

5.3 Implementation

We implemented our approach in a JVMTI agent that computes the steal time of all threads of a Java application. The agent is designed to run in a Linux guest system with the Kernel-based Virtual Machine (KVM, [Kivity07]) or with Xen [Barham03] as a hypervisor. Linux has built-in support for detecting when it is running under KVM or Xen virtualization and for using their hypervisor interfaces.

Linux exposes information about the resource usage of the system, its processes, and its threads via the *procfs* pseudo-filesystem, which is typically mapped to */proc*. The */proc/stat* file provides the resource usage of the entire system, which is broken down into CPU time spent executing application code, CPU time spent in the operating system (mostly on behalf of an application), steal time, and idle time. These times are given in clock ticks since the start of the operating system. Unlike for threads, the system-wide CPU times exclude the steal time. We therefore add the provided steal time to the system-wide CPU times to get $t_{cpu,total}$, and use the steal time as $t_{steal,total}$. We then use the stored values from the last sample to compute $\Delta t_{cpu,total}$ and $\Delta t_{steal,total}$.

The CPU time of an individual process is available in its */proc/[pid]/stat* file, where *[pid]* is the numeric process identifier. The CPU time of a specific thread of a process is available in */proc/[pid]/task/[tid]*, where *[tid]* is the thread identifier. The times are given in clock ticks since the start of the thread and are split up into "regular" user-space ticks and system ticks. System ticks denote time spent in the operating system on behalf of the application. In our implementation, we do not distinguish between user-space ticks and system ticks. Whenever we take a sample, we take their sum and compare it to the stored sum from the last sample to get $\Delta t_{cpu,total}(x)$.

Our JVMTI agent is loaded during the startup of the Java VM. During initialization, we create data structures to store the tick counts from */proc* that are required to compute the deltas, and to store the steal time that is attributed to each thread. Our agent registers for JVMTI events so that it is notified when application threads start and end. This enables us to create a record for a thread when it starts and to stop monitoring a thread (but keep its record) when it ends.

When the application is launched, our agent starts a separate thread with a sampling loop, in which we periodically retrieve the system-wide tick counts as well as the tick counts for all existing application threads. We then compute the deltas using the stored values from our records, attribute the new steal time to the threads, and update our records. At the end of an iteration, the sampling loop pauses for the sampling interval before it takes another sample and attributes the new steal time.

5.4 Evaluation

In order to back our claims that our approach is efficient and accurate, we evaluated our agent's accuracy and its overhead. We determined our agent's accuracy by subtracting the steal time that was attributed to a thread from the thread's reported CPU time, and by comparing this corrected CPU time to the thread's true CPU time. This comparison requires knowing the true CPU time that a thread consumes, which is ideally achieved by using a workload in which each thread consumes a predictable amount of CPU time. However, the benchmarks of the DaCapo and Scala suites that we used earlier in this thesis generally do not fit this criterion well. The benchmarks commonly use thread pools to divide work between threads, so the threads consume varying amounts of CPU time in different iterations. Moreover, the same threads are often not consistently named over multiple iterations and are therefore difficult to match to each other. Because of these reasons, we devised our own benchmark with a synthetic, predictable workload.

Our benchmark consists of six threads with different execution patterns. Figure 5.2 illustrates the patterns that each thread T_i executes, with dashed lines denoting periods of waiting, and with solid bars indicating periods of CPU usage. T_1 to T_4 each alternate between periods of CPU usage and periods of waiting. T_5 continuously consumes CPU time. T_6 alternates between waiting and consuming CPU time for random durations up to 100ms. T_7 to T_{36} are 30 additional threads that are always idle, but must still be considered in steal time accounting. T_{all} shows the CPU usage of all threads if they executed exactly in parallel, with darker segments indicating periods of higher CPU usage. It demonstrates that while each thread's CPU usage follows a pattern, the entire benchmark's CPU usage changes rather irregularly. We designed our benchmark so that each iteration takes 10 seconds on our hardware (when it is otherwise idle), so that all threads run in parallel during that time, and so that each thread consumes nearly the same fixed amount of CPU time per iteration.

For our evaluation, we set up KVM virtualization under openSUSE Linux 13.2 on a computer with a quad-core Intel Core i7-4790K processor with 16 GB of memory. To get more stable results, we disabled the hyperthreading, turbo boost and dynamic frequency

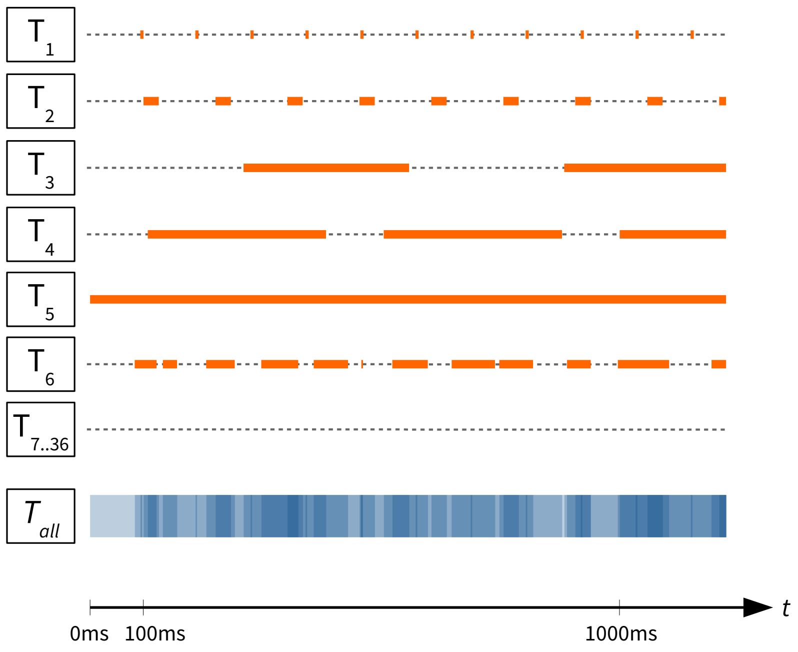


Figure 5.2: Patterns executed in the threads of our synthetic benchmark

scaling features. With the exception of essential system services, no other processes were running during our measurements. On this host system, we created two identical guest systems, each running our synthetic benchmark. Each guest system has four virtual CPUs, 1 GB of memory and openSUSE Linux 13.1 and Oracle JDK 8u20 installed on it.

We performed the measurements that we present in the following sections according to the methodology that we first described in Section 3.4.1. For each measurement, we executed 10 *rounds* of our benchmark. Each round consists of 30 successive *iterations* of our benchmark in a single Java VM instance, of which we discarded the data from the first 20 iterations to compensate for the VM's startup phase. For each iteration, our agent recorded the steal time that it attributed to each thread, and the CPU time that the operating system reported for each thread.

5.4.1 Accuracy

In order to determine our agent's accuracy, we first needed to determine the true CPU usage of each thread of our benchmark. We executed our benchmark in the first guest system while the second guest system remained shut down. In this otherwise idle environment, the first guest system was not affected by steal time. For each iteration of our benchmark, we recorded the CPU times that were consumed by each thread.

We then measured the impact of steal time on the CPU time reported by the operating system, and how accurately our agent attributes this steal time to the threads. We executed our benchmarks again in the first guest system, with our agent taking samples and accounting for the steal time. For these measurements, we started our benchmark also in the second guest system. Therefore, both guest systems competed for the physical CPUs of the host system, and the hypervisor was forced to suspend virtual CPUs. Because the CPU usage of our benchmark in both guest systems continuously changes, their different threads are subjected to changing amounts of steal time. We performed measurements using sampling intervals of 1 second, 100 milliseconds and 10 milliseconds for our agent.

For each of the threads T_1 to T_6, Figure 5.3 shows the thread's median CPU time as reported by the operating system, the thread's median corrected CPU time after subtracting the steal time that our agent attributed to it, and the thread's median expected CPU time that we determined in an idle environment. The CPU times are normalized to each thread's expected CPU time. The error bars indicate the first and third quartiles. The idle threads T_7 to T_{36} are omitted in the diagram, and, as expected, were never attributed any CPU time or steal time. The error bars of the expected CPU times are insignificant and demonstrate that the CPU time of each thread is very stable and predictable across iterations. Overall, the CPU times reported by the operating system exceed the expected CPU times by typically more than 60% and up to 100%. In contrast to that, the corrected steal times are always within 20% of the expected CPU time, and frequently within 10% or even 5% of it. With T_1, T_2, and T_6, which consume CPU time in shorter bursts, our agent slightly underestimates the impact of steal time on them. For T_3, T_4, and T_5, which consume more CPU time at once, our agent tends to overestimate the amount of steal time. Using shorter sampling intervals does not increase the accuracy of the results in general. In fact, shorter sampling periods lead to unanticipated increases or decreases of the CPU time that the guest operating system reports for each thread. We consider this to be an effect of our agent's sampling thread on the guest operating system's scheduler, and possibly even on the hypervisor's virtual CPU scheduler.

5.4.2 Runtime Overhead

In order to determine the overhead of our agent, we compare our benchmark's wall-clock execution time when sampling at different intervals to its wall-clock execution time without our agent. We do so both with and without a second guest system that also executes our benchmark and therefore causes steal time by competing for physical CPU time. In either scenario, our agent must always retrieve the CPU times of all benchmark threads from the guest operating system and keep them in its data structures.

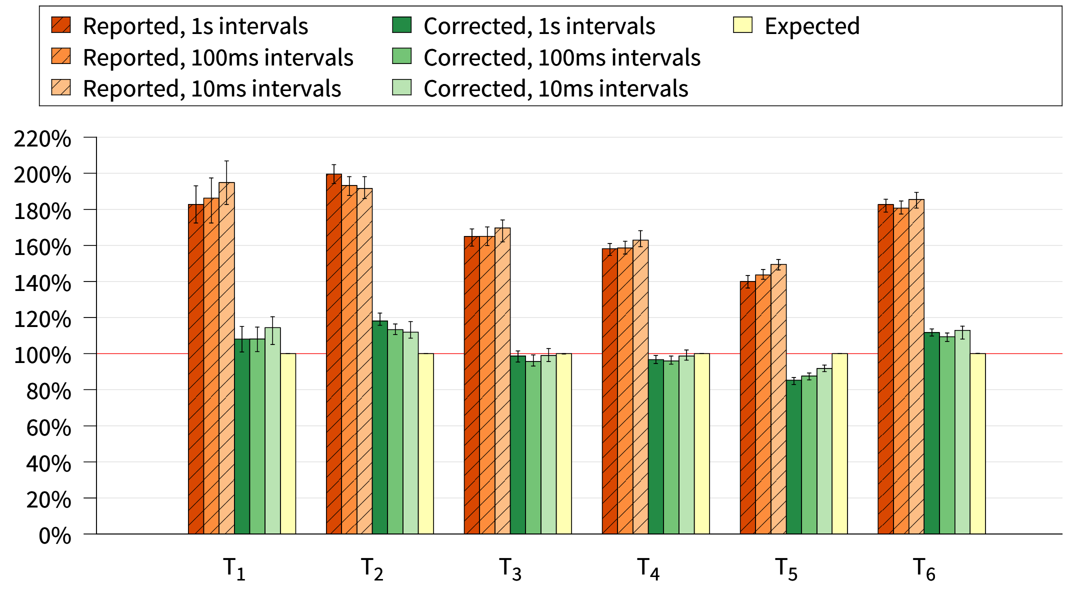


Figure 5.3: CPU times of threads with and without accounting for steal time at different sampling intervals, normalized to each thread's expected CPU time

Figure 5.4 shows the median wall-clock execution times for both scenarios, normalized to the median wall-clock execution time without an agent in each scenario. The error bars indicate the first and third quartiles. When generating load in the second guest system and with 1-second intervals, we measured an unanticipated speedup by around 4%, which we suspect to be the result of interference of our agent with scheduling. With 100 ms and 10 ms intervals, we measured execution times that were 2% and 8% longer (respectively) than without an agent. However, in an otherwise idle environment, our agent only has 1% overhead with 10 millisecond sampling intervals, and negligible overheads at longer sampling intervals. Therefore, we consider the overhead of our agent to be low in general, but to be dependent on the workload and on the sampling interval.

5.5 Related Work

Timing issues in guest systems have been investigated before. Lampe et al. measured the impact of inaccurate timing in publicly available cloud computing services [Lampe12]. Johnson et al. describe extensions to the Performance API (PAPI) project which provide more accurate timers and shared resource usage statistics under virtualization [Johnson12]. However, these extensions only provide system-wide steal

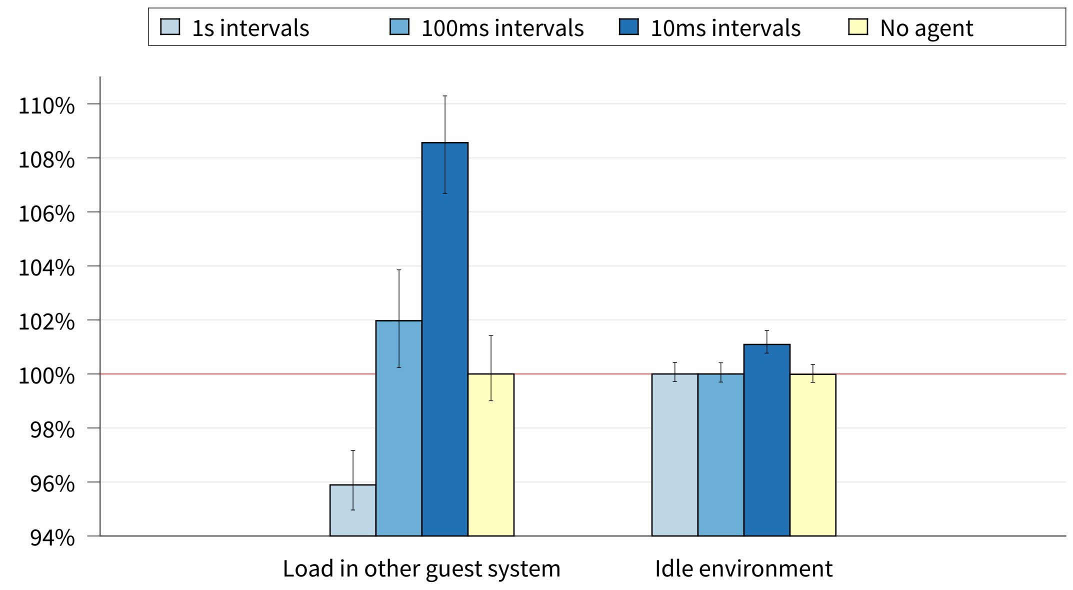


Figure 5.4: Overhead of our steal time accounting agent at different sampling intervals

time information, and Johnson et al. emphasize the need for per-process steal time accounting.

Chen et al. describe a modification to the Xen hypervisor that delivers hypervisor scheduling events to a modified guest operating system which then uses them to correctly attribute steal time to processes and threads [Chen10]. Similarly, Yamamoto and Nakashima propose a host-level sampling profiler that provides steal time samples to a guest-level profiler, which can then attribute those samples to processes, threads and functions [Yamamoto16]. Holzheu proposed a patch for the Linux kernel that attributes per-CPU steal time them to the scheduled processes and threads [Holzheu10]. All three approaches require modifications to the operating system or to the hypervisor, while our approach only requires access to system-wide steal time information.

Binder et al. point out that CPU time in seconds is difficult to compare between systems and that it is not an appropriate metric in distributed, heterogeneous environments [Binder06b]. They propose an approach for Java which uses instrumentation to count the executed bytecode instructions as a portable CPU consumption metric. Further work by Hulaas and Binder focused on optimizations that reduce the runtime overhead of this approach [Hulaas08]. The approach is unaffected by hypervisor suspension, but with 30-50% runtime overhead, it has a higher impact on performance than our approach.

5.6 Summary and Future Work

In this chapter, we described a novel approach for determining to what degree application threads in a guest system are affected when the hypervisor suspends virtual CPUs. Our approach does not require any modifications to the hypervisor or to the operating system. We implemented our approach for Java applications in a Linux guest system with KVM or Xen as the hypervisor. An evaluation with a synthetic workload demonstrated that our approach accurately attributes the system-wide steal time to the application threads at low overhead, and that the attributed steal time can be used to correctly estimate a thread's true execution time on a physical CPU.

Future work could focus on attributing steal time not only to threads, but also to *transactions* that execute within a thread, such as individual web requests. Measuring the duration and resource usage of transactions is a central feature of APM software, and suspension can considerably distort these measurements because of the typically short execution time of a transaction. Further research could also investigate the use of scheduling traces from the Linux perf subsystem to try to attribute steal time more accurately, and to compare the results with those from our approach.

Chapter 6

Summary

This chapter summarizes and concludes this thesis.

In this thesis, we claimed that profiling mechanisms that are integrated into a Java VM can collect performance data more efficiently and more accurately than external profiling mechanisms. We proposed multiple approaches for profiling within a Java VM and implemented those approaches in the production-quality, high-performance HotSpot VM to evaluate them.

First, we proposed novel approaches for **sampling-based method profiling.** We described *Stack Fragment Sampling*, an approach which integrates with the operating system to sample threads only while they are running and to interrupt them only briefly to copy stack fragments. Our approach asynchronously decodes these fragments to stack traces and uses heuristics for stacks with native code and VM-internal code. In our evaluation, we demonstrated that the performance impact of Stack Fragment Sampling is significantly lower than that of comparable techniques, with a mean overhead of 2.1% at 1 ms sampling intervals, and that it consistently generates an accurate picture of the application's behavior.

We then described *Partial Safepoints, Self-Sampling and Incremental Stack Tracing.* Partial safepoints only pause a subset of the application threads for sampling and can be used to target those threads which are currently executing. With self-sampling, threads take their own stack traces in parallel. Incremental stack tracing constructs stack traces lazily instead of walking the entire stack for each sample. These three techniques form an alternative to Stack Fragment Sampling that is independent of the operating system and hardware. Our evaluation demonstrated that the approach significantly reduces the sampling overhead and pause times compared to profiling with JVMTI, while providing similar accuracy. Incremental stack tracing with partial safepoints and self-sampling incurs a mean overhead of 6.7% at 1 ms sampling intervals, and 2.1% at 10 ms sampling intervals.

Second, we proposed a novel approach for **profiling lock contention.** Our approach efficiently records events inside the Java VM and traces lock contention from both Java's intrinsic locks and from java.util.concurrent locks. For the analysis of the traces, we devised a versatile technique to aggregate and visualize the recorded contentions. Unlike other methods, our approach shows not only where contention occurs, but also where contention is caused. In our evaluation, we measured overheads of less than 10% for almost all benchmarks, which we consider adequate for continuously monitoring production systems.

Our evaluations show that our approaches for sampling-based method profiling and for profiling lock contention incur low overheads while collecting accurate data. This supports our claims that integrating profiling mechanisms into a Java VM is an effective strategy to achieve both efficiency and accuracy. Although we implemented our approaches in HotSpot, we consider them to be general enough to be adapted for other virtual machines, runtime environments and languages.

Finally, we proposed a new approach for **hardware virtualization steal time accounting.** Our sampling-based approach only uses commonly available performance counters. It can be implemented outside of the Java virtual machine and does not require any modifications to the hypervisor or to the operating system. Our evaluation demonstrated that the approach accurately attributes steal time to threads at negligible overhead. We believe that the general approach is also applicable to other hypervisors and guest operating systems.

List of Figures

Bibliography

[Adamoli10] Andrea Adamoli and Matthias Hauswirth. *Trevis: A Context Tree Visualization & Analysis Framework and Its Use for Classifying Performance Failure Reports*. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 73–82. ACM, 2010.

[Ammons97] Glenn Ammons, Thomas Ball, and James R. Larus. *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling*. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96. ACM, 1997.

[Angerer15] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. *Configuration-Aware Change Impact Analysis (T)*. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 385–395. IEEE, 2015.

[Ansaloni13] Danilo Ansaloni, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, and Petr Tůma. *Enabling Modularity and Re-use in Dynamic Program Analysis Tools for the Java Virtual Machine*. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP '13, pages 352–377. Springer, 2013.

[Arnold05] Matthew Arnold and David Grove. *Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines*. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 51–62. IEEE, 2005.

[Bacon98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. *Thin Locks: Featherweight Synchronization for Java*. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268. ACM, 1998.

[Barham03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. *Xen and* *the Art of Virtualization*. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177. ACM, 2003.

[Binder06a] Walter Binder. *Portable and Accurate Sampling Profiling for Java*. *Software: Practice and Experience*, 36(6):615–650, 2006.

[Binder06b] Walter Binder and Jarle Hulaas. *Using Bytecode Instruction Counting As Portable CPU Consumption Metric*. *Electronic Notes in Theoretical Computer Science*, 153(2):57–77, 2006.

[Blackburn06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190. ACM, 2006.

[Bond07] Michael D. Bond and Kathryn S. McKinley. *Probabilistic Calling Context*. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 97–112. ACM, 2007.

[Chen10] Huacai Chen, Hai Jin, and Kan Hu. *XenHVMAcct: Accurate CPU Time Accounting for Hardware-Assisted Virtual Machine*. In *Proceedings of the 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '10, pages 191–198. IEEE, 2010.

[Citron06] Daniel Citron, Adham Hurani, and Alaa Gnadrey. *The Harmonic or Geometric Mean: Does It Really Matter?* *ACM SIGARCH Computer Architecture News*, 34(4):18–25, 2006.

[David14] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. *Continuously Measuring Critical Section Pressure with the Free-lunch Profiler*. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 291–307. ACM, 2014.

[D'Elia11] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. *Mining Hot Calling Contexts in Small Space*. In *Proceedings of the 32nd ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '11, pages 516–527. ACM, 2011.

[Detlefs04] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. *Garbage-first Garbage Collection*. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48. ACM, 2004.

[Feller98] Peter T Feller. *Value Profiling for Instructions and Memory Locations*. Master's thesis, University of California, San Diego, 1998.

[Fleming86] Philip J Fleming and John J Wallace. *How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results*. *Communications of the ACM*, 29(3):218–221, 1986.

[Gnedt14] David Gnedt. *Fast Profiling in the HotSpot Java VM with Incremental Stack Tracing and Partial Safepoints*. Bachelor's thesis, Johannes Kepler University Linz, 2014.

[Gnedt16] David Gnedt. *Runtime Analysis of High-Level Java Synchronization Mechanisms on the Virtual Machine Level*. Master's thesis, Johannes Kepler University Linz, 2016.

[Goetz06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Pearson Education, 2006.

[Gosling15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java®Language Specification: Java SE 8 Edition*. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`, 2015.

[Graal15] Oracle. *Graal Project*. `http://openjdk.java.net/projects/graal/`, 2015.

[Griesemer99] Robert Griesemer. *Generation of Virtual Machine Code at Startup*. In *OOPSLA '99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design*. 1999.

[Grimmer15] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. *High-Performance Cross-Language Interoperability in a Multi-Language Runtime*. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS '15, pages 78–90. ACM, 2015.

[Han12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. *Performance Debugging in the Large via Mining Millions of Stack Traces*. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155. IEEE, 2012.

[Hirt10] Marcus Hirt and Marcus Lagergren. *Oracle JRockit: The Definitive Guide*. Packt Publishing Ltd, 2010.

[Hirt13] Marcus Hirt. *Low Overhead Method Profiling with Java Mission Control*. `http://hirt.se/blog/?p=364`, 2013. In response to a comment, Hirt states: *The method profiling in Java Flight Recorder is based upon AsyncGetCallTrace, but improved to meet the needs for low overhead profiling in production. This work has been of benefit for the old unsupported AsyncGetCallTrace too, as some of the work to harden the Flight Recorder profiler has gone back into the old AsyncGetCallTrace.*

[Hofer14a] Peter Hofer and Hanspeter Mössenböck. *Efficient and Accurate Stack Trace Sampling in the Java Hotspot Virtual Machine*. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 277–280. ACM, 2014.

[Hofer14b] Peter Hofer and Hanspeter Mössenböck. *Fast Java Profiling with Scheduling-aware Stack Fragment Sampling and Asynchronous Analysis*. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 145–156. ACM, 2014.

[Hofer15a] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. *Efficient Dynamic Analysis of the Synchronization Performance of Java Applications*. In *Proceedings of the 13th International Workshop on Dynamic Analysis*, WODA '15, pages 14–18. ACM, 2015.

[Hofer15b] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. *Lightweight Java Profiling with Partial Safepoints and Incremental Stack Tracing*. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 75–86. ACM, 2015.

[Hofer15c] Peter Hofer, Florian Hörschläger, and Hanspeter Mössenböck. *Sampling-based Steal Time Accounting Under Hardware Virtualization*. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 87–90. ACM, 2015.

[Hofer16] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. *Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level*. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 263–274. ACM, 2016.

[Holzheu10] Michael Holzheu. *[RFC][PATCH v2 4/7] taskstats: Add per task steal time accounting*. `https://lkml.org/lkml/2010/11/11/271`, 2010.

[HotSpot15] Oracle. *Java HotSpot Virtual Machine*. `http://openjdk.java.net/groups/hotspot/`, 2015.

[Huang13] Jipeng Huang and Michael D. Bond. *Efficient Context Sensitivity for Dynamic Analyses via Calling Context Uptrees and Customized Memory Management*. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 53–72. ACM, 2013.

[Hulaas08] Jarle Hulaas and Walter Binder. *Program Transformations for Light-weight CPU Accounting and Control in the Java Virtual Machine*. *Higher-Order and Symbolic Computation*, 21(1-2):119–146, June 2008.

[Inoue09] Hiroshi Inoue and Toshio Nakatani. *How a Java VM Can Get More from a Hardware Performance Monitor*. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 137–154. ACM, 2009.

[JavaNio15] Oracle. *java.nio package in Java SE 8*. `https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html`, 2015.

[JMH15] Oracle. *OpenJDK Code Tools: Java Microbench Harness*. `http://openjdk.java.net/projects/code-tools/jmh/`, 2015.

[JMX15] Oracle. *The Java^{TM} Management Extensions*. `http://openjdk.java.net/groups/jmx/`, 2015.

[JNI15] Oracle. *Java Native Interface 6.0 Specification*. `http://docs.oracle.com/javase/8/docs/technotes/guides/jni/`, 2015.

[Johnson12] Matthew Johnson, Heike McCraw, Shirley Moore, Phil Mucci, John Nelson, Dan Terpstra, Vince Weaver, and Tushar Mohan. *PAPI-V: Performance Monitoring for Virtual Machines*. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 194–199. IEEE, 2012.

[Juneau10] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto Muoz, Victor Ng, Alex Ng, and Donna L Baker. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, 2010.

[JVMTI13] Oracle. *JVM^{TM}Tool Interface Version 1.2.3*. `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`, 2013.

[Kivity07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. *KVM: the Linux Virtual Machine Monitor*. In *Proceedings of the Linux Symposium 2007*, volume 1, pages 225–230. 2007.

[Kotzmann05] Thomas Kotzmann and Hanspeter Mössenböck. *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 111–120. ACM, 2005.

[Kotzmann08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. *Design of the Java HotSpot^{TM} Client Compiler for Java 6*. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, 2008.

[Lampe12] Ulrich Lampe, André Miede, Nils Richerzhagen, Dieter Schuller, and Ralf Steinmetz. *The Virtual Margin of Error: On the Limits of Virtual Machines in Scientific Research*. In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*, CLOSER '12. 2012.

[Lea05] Doug Lea. *The java.util.concurrent Synchronizer Framework*. *Science of Computer Programming*, 58(3):293–309, 2005.

[Lengauer14] Philipp Lengauer and Hanspeter Mössenböck. *The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors*. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 111–122. ACM, 2014.

[Lengauer15] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. *Accurate and Efficient Object Tracing for Java Applications*. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 51–62. ACM, 2015.

[Lengauer16] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. *Efficient Memory Traces with Full Pointer Information*. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16. ACM, 2016.

[Lettner15] Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. *Feature Modeling of Two Large-Scale Industrial Software Systems: Experiences and Lessons Learned*. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, MODELS '15, pages 386–395. IEEE, 2015.

[Lindholm15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java®Virtual Machine Specification: Java SE 8 Edition*. `https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf`, 2015.

[Linux15] Linus Torvalds et al. *Linux*. `https://kernel.org/`, 2015.

[Maplesden15] David Maplesden, Ewan Tempero, John Hosking, and John C. Grundy. *Subsuming Methods: Finding New Optimisation Opportunities in Object-Oriented Software*. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 175–186. ACM, 2015.

[Marek12] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. *DiSL: A Domain-specific Language for Bytecode Instrumentation*. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250. ACM, 2012.

[Marek13] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. *ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform*. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 105–114. ACM, 2013.

[Moret09] Philippe Moret, Walter Binder, and Alex Villazon. *CCCP: Complete Calling Context Profiling in Virtual Execution Environments*. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 151–160. ACM, 2009.

[Moret10] Philippe Moret, Walter Binder, Alex Villazón, Danilo Ansaloni, and Abbas Heydarnoori. *Visualizing and Exploring Profiles with Calling Context Ring Charts*. *Software: Practice and Experience*, 40(9):825–847, 2010.

[Mössenböck00] Hanspeter Mössenböck. *Adding Static Single Assignment Form and a Graph Coloring Register Allocator to the Java HotSpot^{TM}Client Compiler*. Technical report, Institute for Practical Computer Science, Johannes Kepler University Linz, 2000.

[Mössenböck02] Hanspeter Mössenböck and Michael Pfeiffer. *Linear Scan Register Allocation in the Context of SSA Form and Register Constraints*. In *Compiler Construction*, pages 229–246. Springer, 2002.

[Mytkowicz10] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. *Evaluating the Accuracy of Java Profilers*. In *Proceedings of* *the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 187–197. ACM, 2010.

[NetBeans14] Oracle. *NetBeans Profiler*. `http://profiler.netbeans.org/`, 2014.

[Odersky04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification*, 2004.

[Paleczny01] Michael Paleczny, Christopher Vick, and Cliff Click. *The Java Hotspot Server Compiler*. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, JVM '01, pages 1–12. USENIX, 2001.

[Patros15] Panagiotis Patros, Eric Aubanel, David Bremner, and Michael Dawson. *A Java Util Concurrent Park Contention Tool*. In *Proceedings of the 6th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 106–111. ACM, 2015.

[perf15] Linux Kernel Organization. *perf: Linux profiling with performance counters*. `https://perf.wiki.kernel.org/`, 2015.

[Pool14] Tõnis Pool. *Lock Optimizations on the HotSpot VM*. Technical report, University of Tartu, 2014.

[POSIX13] Andrew Josey, Donald Cragun, Nicholas Stoughton, Mark Brown, Cathy Fox, et al. *The Open Group Base Specifications Issue 7 / IEEE Std 1003.1 2013 Edition*. The IEEE and The Open Group, 2013.

[Pozo04] Roldan Pozo and Bruce Miller. *SciMark 2.0*. `http://math.nist.gov/scimark2/`, 2004.

[Rose08] John Rose, Danny Coward, Ola Bini, Remi Forax, William Cook, Samuele Pedroni, and Jochen Theodorou. *JSR 292: Supporting Dynamically Typed Languages on the Java^{TM}Platform*. `https://jcp.org/en/jsr/detail?id=292`, 2008.

[Russell06] Kenneth Russell and David Detlefs. *Eliminating Synchronization-related Atomic Operations with Biased Locking and Bulk Rebiasing*. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 263–272. ACM, 2006.

[Serrano09] Mauricio Serrano and Xiaotong Zhuang. *Building Approximate Calling Context from Partial Call Traces*. In *Proceedings of the 7th IEEE/ACM*

International Symposium on Code Generation and Optimization, CGO '09, pages 221–230. IEEE, 2009.

[Sewe11] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. *Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine*. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 657–676. ACM, 2011.

[Snappy15] Google. *Snappy: a fast compressor/decompressor*. `http://google.github.io/snappy/`, 2015.

[Stadler09] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. *Lazy Continuations for Java Virtual Machines*. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 143–152. ACM, 2009.

[Stolte02] Chris Stolte, Diane Tang, and Pat Hanrahan. *Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases*. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[Sumner10] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. *Precise Calling Context Encoding*. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 525–534. ACM, 2010.

[Sun15] Haiyang Sun, Yudi Zheng, Lubomír Bulej, Alex Villazón, Zhengwei Qi, Petr Tůma, and Walter Binder. *A Programming Model and Framework for Comprehensive Dynamic Analysis on Android*. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY '15, pages 133–145. ACM, 2015.

[Sundstrom15] Dain Sundstrom. *Snappy in Java*. `https://github.com/dain/snappy`, 2015.

[Tallent10] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. *Analyzing Lock Contention in Multithreaded Applications*. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280. ACM, 2010.

[Vierhauser15] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, and Benedikt Aumayr. *A Requirements Monitoring Model for Systems of Systems*. In *2015 IEEE 23rd International Requirements Engineering Conference*, RE '15, pages 96–105. IEEE, 2015.

[Vierhauser16] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel. *ReMinds: A Flexible Runtime Monitoring Framework for Systems of Systems*. *Journal of Systems and Software*, 112:123 – 136, 2016.

[VMWare13] *The CPU Scheduler in VMware vSphere 5.1*. VMware Inc., 2013.

[Weaver13] Vincent M Weaver. *Linux perf_event Features and Overhead*. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems*, FastPath '13. 2013.

[Weaver15] Vincent M Weaver. *The Unofficial Linux Perf Events Web-Page*. `http://web.eece.maine.edu/~vweaver/projects/perf_events/`, 2015.

[Whaley00] John Whaley. *A Portable Sampling-based Profiler for Java Virtual Machines*. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 78–87. ACM, 2000.

[Wimmer05] Christian Wimmer and Hanspeter Mössenböck. *Optimized Interval Splitting in a Linear Scan Register Allocator*. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 132–141. ACM, 2005.

[Würthinger09] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. *Array Bounds Check Elimination in the Context of Deoptimization*. *Science of Computer Programming*, 74(5):279–295, 2009.

[Würthinger13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. *One VM to Rule Them All*. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 187–204. ACM, 2013.

[Yamamoto16] Masao Yamamoto and Kohta Kohta Nakashima. *Execution Time Compensation for Cloud Applications by Subtracting Steal Time Based on Host-Level Sampling*. In *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*, ICPE '16 Companion, pages 69–73. ACM, 2016.

[Zheng15] Yudi Zheng, Lubomír Bulej, and Walter Binder. *Accurate Profiling in the Presence of Dynamic Compilation*. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, pages 433–450. ACM, 2015.

[Zhuang06] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. *Accurate, Efficient, and Adaptive Calling Context Profiling*. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271. ACM, 2006.