Universidade Federal da Bahia
Instituto de Computação

Bacharelado em Ciência da Computação

JVMTIPROF: AN EXTENSION TO THE JAVA VIRTUAL MACHINE TOOL INFRASTRUCTURE FOR BUILDING PROFILING AGENTS

Denilson das Mercês Amorim

TRABALHO DE GRADUAÇÃO

Salvador
25 de Novembro de 2022

DENILSON DAS MERCÊS AMORIM

JVMTIPROF: AN EXTENSION TO THE JAVA VIRTUAL MACHINE TOOL INFRASTRUCTURE FOR BUILDING PROFILING AGENTS

Este Trabalho de Graduação foi apresentada ao Bacharelado em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Tiago de Oliveira Januario

Salvador
25 de Novembro de 2022

Sistema de Bibliotecas - UFBA

Amorim, Denilson das Mercês.

JVMTIPROF: An extension to the Java Virtual Machine Tool Infrastructure for building profiling agents / Denilson das Mercês Amorim – Salvador, 2022.

47p.: il.

Orientador: Prof. Dr. Tiago de Oliveira Januario.

Monografia (Graduação) – Universidade Federal da Bahia, Instituto de Computação, 2022.

1. Java Virtual Machine. 2. Instrumentação de bytecode. 3. Interceptação de função. 4. Perfilamento de software. 5. Safe points. I. Januario, Tiago de Oliveira. II. Universidade Federal da Bahia. Instituto de Computação. III Título.

CDD – XXX.XX

CDU – XXX.XX.XXX

TERMO DE APROVAÇÃO

DENILSON DAS MERCÊS AMORIM

JVMTIPROF: AN EXTENSION TO THE JAVA VIRTUAL MACHINE TOOL INFRASTRUCTURE FOR BUILDING PROFILING AGENTS

Este Trabalho de Graduação foi julgada adequada à obtenção do título de Bacharel em Ciência da Computação e aprovada em sua forma final pelo Bacharelado em Ciência da Computação da Universidade Federal da Bahia.

Salvador, 25 de Novembro de 2022

Prof. Dr. Tiago de Oliveira Januario
Universidade Federal da Bahia

Prof. Dr. Vinicius Tavares Petrucci
Micron Technology, Inc.

Prof. Dr. Ramon Pereira Lopes
Universidade Federal do Recôncavo da Bahia

ACKNOWLEDGEMENTS

This thesis is dedicated to the memory of my dear father, Edilson Amorim. He conceded to me the privilege of having a personal computer from a tender age, greatly enhancing my access to information and related opportunities. I'm also grateful to my caring mother, Jucilene das Mercês, who, despite the social and financial difficulties, was able to raise a decent and literate man.

During my academic years, I have met many individuals. Some have become close friends, and some have parted away, but they all contributed a little to my formation. I'm grateful for that. Many thanks to Dr. Vinicius Petrucci for showing me the way of science. The exposure to a research environment was vital to sharpening my critical thinking skills. Thanks to Dr. Maurício Segundo and Dr. Rubisley Lemes for leading UFBA's competitive programming group (GRUPRO). It improved my problem-solving skills and gave me access to opportunities I could only dream of.

Results presented in this work were obtained using the Chameleon testbed supported by the National Science Foundation of the United States of America.

The journey of a thousand miles begins with one step.

—LAO TZU (Tao Te Ching)

RESUMO

Construir agentes de perfilamento eficientes para a máquina virtual Java é uma tarefa desafiadora. O trabalho é dificultado pelo fato da maioria dos frameworks de instrumentação de alto nível serem escritos e precisarem ser usados dentro das fronteiras da linguagem de programação Java. Este trabalho apresenta a JVMTIPROF, uma extensão para a JVM Tool Infrastructure provendo funcionalidades de alto nível para serem utilizadas por agentes de instrumentação e perfilamento. A JVMTIPROF pode ser utilizada para interceptar chamadas a métodos e obter amostras precisas da pilha de chamadas de uma aplicação, tudo em código nativo. Nós avaliamos nossa solução substituindo a JVMTI em um escalonador de frequência guiada por perfil, e não encontramos nenhuma penalidade de desempenho significativa.

Palavras-chave: Java Virtual Machine. Instrumentação de bytecode. Interceptação de função. Perfilamento de software. Safe points.

ABSTRACT

Building efficient profiling agents for the Java Virtual Machine is a challenging task. The job is hardened by most high-level instrumentation frameworks being written in and for use within the boundaries of the Java programming language. This work presents JVMTIPROF, an extension to the JVM Tool Infrastructure providing high-level functionality for instrumentation and profiling agents. JVMTIPROF can be used to intercept method calls and accurately sample the call stack of an application, all in native code. We evaluated our solution by replacing JVMTI in a profile-guided frequency scaling scheme and found no significant performance penalties.

Keywords: Java Virtual Machine. Bytecode instrumentation. Function interception. Software profiling. Safe points.

CONTENTS

LIST OF FIGURES

LISTINGS

Chapter 1

INTRODUCTION

Profiling is a commonly used technique in the software industry to identify application bottlenecks. These bottlenecks may be of different natures, such as performance, storage, or energy consumption.

Profilers often use instrumentation to perform program transformation. For example, to accurately measure the time spent in each function, its entry and exit points may be modified to collect timings.

Profiling suffers from a compromise between precision and performance (PONDER; FATEMAN, 1988). Too much interference to the application may cause its performance characteristics to change, producing unreliable results. On the other hand, little interference may not yield sufficient information for an accurate profile.

1.1 MOTIVATION

Profilers are typically standalone programs, reporting metrics and events about a target application. These profilers are tailored to a specific purpose and do not allow customization points.

Tool-building systems are frequently used to construct program analyzers. These systems permit the installation of events on the target program, which can be acted upon by client-driven code. These systems can be used for several tasks, such as building profilers, schedulers, and patch snippets of existing programs.

The Java Virtual Machine Tool Infrastructure (JVMTI) is an example of a tool-building system used by most existing Java profilers. However, this infrastructure provides only bare-bones features, leaving it to the client to build high-level functionality. For example, to instrument the entry point of a selected method, the client needs to intercept the loading of compiled code and rebuild it by transforming the related data structures.

1.2 CONTRIBUTION

This work presents JVMTIPROF, a programming interface extending the JVMTI. We use the same patterns, idioms, and types, extending its interface to provide more functionalities. This way, developers of profiling and instrumentation agents can focus their effort on methods instead of on the supporting infrastructure. We also demonstrate how the JVMTI can be extended without modifying its source code.

1.3 THESIS STRUCTURE

Chapter 2 discusses this thesis's background and related work. Chapter 3 presents the design and implementation of JVMTIPROF. Chapter 4 demonstrates its use in the instrumentation of a search engine. Chapter 5 wraps up the work and points to future directions.

Chapter 2

BACKGROUND

This chapter presents related work and background information necessary to understand this thesis. We first introduce concepts of profiling and instrumentation, then dive into some aspects of the Java Virtual Machine (JVM for short). We follow by presenting a search engine that runs on top of the JVM and concepts related to Linux power governors. Finally, we discuss related work.

2.1 INSTRUMENTATION

Software instrumentation is a technique that adds auxiliary code to an existing computer program. Instrumentation can be performed either statically (i.e., at compile-time) or dynamically (i.e., at runtime). Static instrumentation is accomplished by modifying the source code or rewriting the application target binary, and dynamic instrumentation is performed after the binary is in-memory (KEMPF et al., 2008).

Function hooking is an instrumentation technique by which function calls are intercepted to perform additional operations before or after the function is invoked (LOPEZ et al., 2017).

Instrumentation has many applications, such as in profiling, debugging, program analysis, code coverage, and just-in-time compilation (GRAHAM et al., 1982; ZHAO et al., 2008; SEWARD; NETHERCOTE, 2005; IVANKOVIć et al., 2019; Oracle, 2011a). For example, a profiler might hook every function of an application to measure and report its execution time.

2.2 PROFILING

Software profiling is a form of dynamic program analysis used to measure the performance characteristics of a computer program. A profiler can estimate a range of metrics, such as execution time, memory usage, and energy consumption (GRAHAM et al., 1982; REISS, 2009; PATHAK et al., 2012).

Profilers are typically used to identify code blocks where most of the application's time is spent. Once identified, these hot paths can be improved, either manually (i.e., by

a programmer) or automatically (i.e., through an algorithm). An optimizing compiler, for example, might use a profile to make more informed decisions or optimize certain functions more aggressively (PETTIS; HANSEN, 1990).

CPU profilers can be classified as sampling profilers, in which threads are interrupted once in a while to take samples of its execution, and instrumenting profilers, where the program's code is modified at critical locations to include the reporting of statistics. Sampling profilers typically produce much lower runtime overhead than the instrumented counterpart (MOSELEY et al., 2007).

2.2.1 Sampling in Unix-like Systems

Sampling profilers need to sample the call stack of a thread every few milliseconds. A common way to achieve this on Unix-like systems is to have an application-wide timer that notifies its expiration through a signal. The `SIGPROF` signal and `ITIMER_PROF` timer are typically used for this purpose. Once the signal is received, the call stack is sampled by walking on the stack of the running thread.

The code executed in the signal handler must be carefully crafted to be *async-signal-safe* (KERRISK, 2016). Async-signal-safe operations are guaranteed not to interfere with operations in the interrupted thread. For example, signals may produce deadlocks if they try to hold the same locks as the interrupted thread. As such, async-signal-safe code is limited to primitive memory operations and a subset of system calls. Given this limitation, profiling signal handlers typically publish the acquired samples to a separate worker thread through an async-signal-safe data structure.

2.3 JAVA VIRTUAL MACHINE

The *Java Virtual Machine* is an abstract computing machine. Its primary purpose is to execute code independently of the host hardware and operating system (LINDHOLM et al., 2022). This machine has an instruction set and a set of memory areas. There are several implementations of the JVM. Some are in hardware (O'CONNOR; TREMBLAY, 1997), but most are software-based (Oracle, 2022c; Eclipse Foundation, 2022; Azul Systems, 2022). Oracle's HotSpot is the most popular of these implementations.

Java is an object-oriented programming language that compiles to the JVM instruction set. Java found wide popularity in the software industry due to the characteristics of the Java Virtual Machine (KUMAR; DAHIYA, 2017). Several other programming languages can target the JVM, such as Scala, Clojure, JRuby, and Jython (LI et al., 2013).

2.3.1 Execution Pipeline

A program targeting the JVM compiles into a set of class files. A *class file* is a binary format containing JVM instructions (or *bytecode*), symbol tables, and other ancillary information. Class files are loaded, linked, and initialized by the JVM's *class loader*.

The JVM executes classes once they are loaded. Most software implementations employ a just-in-time compiler (KOTZMANN et al., 2008). Bytecode compiles into the

host machine instruction set as it gets interpreted. The compiled code can execute with performance characteristics comparable to programming languages natively compiled to machine code (GHERARDI et al., 2012; TABOADA et al., 2013).

Just-in-time compilers may employ profile-guided optimizations. Oracle's HotSpot, for example, performs tiered compilation (Oracle, 2011b). It has four levels of compilation which optimizes and deoptimizes functions between levels as the application profile evolves. HotSpot's JIT constructs profiles through instrumentation of method invocations, and loop backedges (Oracle, 2011a).

The Java Virtual Machine features automatic memory management through garbage collection (PUFEK et al., 2019). An unreachable instance of an object is automatically classified as garbage and freed from memory.

2.3.2 Java Native Interface

The *Java Native Interface (JNI)* is an application programming interface (API) that allows JVM bytecode to interoperate with native machine code (Oracle, 2022a). This library can invoke functions written in natively compiled programming languages such as C++ from within a Java program.

Native methods are declared in the Java program, and during class loading, the JVM binds that method to a symbol of a dynamic link library. Listings 2.1 and 2.2 present an example of this interaction.

The JVM has a per-thread *JNI Environment* that native functions can use to interact with the JVM. This environment is passed as an argument to every native method invocation. The environment consists of a pointer to a function table and internal data opaque to the native code. The function table contains function pointers that can be used to interface with the JVM. Figure 2.1 illustrates the approach.

The JNI interface provides functionality to invoke Java methods, access object fields, create objects, create references to objects, define classes, throw exceptions, and more. All data type interaction must occur through JNI's data types which map to Java types, e.g., JNI's `jobject` maps to Java's `Object`.

2.3.3 JVM Tool Interface

The *JVM Tool Interface (JVMTI)* is a programming interface that native agents can use to build development and monitoring tools for the JVM (Oracle, 2022b). It provides ways to inspect states and react to events of the VM. Although a JVM implementation is not required to support the JVMTI, most software implementations do.

JVMTI follows most of the approaches in JNI, such as using the same data types and exposing its API through a function table. JVMTI environments can be created by calling JNI's `GetEnv` function. A native agent is compiled into a dynamic link library and loaded into the VM through command line options. The agent's dynamic library should include the native symbols `Agent_OnLoad` and `Agent_OnUnload`. The VM invokes these functions during initialization and termination, respectively, to offer the agent an opportunity for setup and tear-down. An example agent is shown in Listing 2.3.

Capabilities JVMTI provides several functions that can be invoked from most contexts, e.g., `GetStackTrace` and several events that can be used to react to the VM, e.g., `MethodEntry`. These functions and events may come at a cost in execution speed, start-up time, and memory footprint. For this reason, JVMTI does not provide any capability by default. During `Agent_OnLoad`, the agent must add (through JVMTI's `AddCapabilities`) the capabilities it needs (e.g. `can_generate_method_entry_events`).

Events In addition to capability setting, events of interest can be enabled (and disabled) by calling JVMTI's `SetEventNotificationMode`. The event's callback functions can be set with `SetEventCallbacks`. Event callbacks receive their JNI and JVMTI environments as arguments, plus additional information related to the event.

Environment Disposal A JVMTI environment can be disposed of explicitly by calling `DisposeEnvironment` or implicitly during agent unloading. During disposal, the environment capabilities are automatically relinquished and its events disabled. Memory and any other additional resources the agent allocates must be freed manually.

2.3.4 Safepoints

A *safepoint* is a point of execution in which all root object references are known, and heap object contents are consistent (Sun Microsystems, 2006b). The most popular JVM implementations use safepoints to assist in implementing operations such as garbage collection, thread dumps, and compiler deoptimization (LIN et al., 2015).

A thread is in a safepoint while performing I/O, in between two interpreted bytecode instructions, when running native code, when contended or waiting in a lock, and at specific points of JIT-compiled code (WAKART, 2015).

Safepoint Polls JIT-compiled code can only know the root of object references through compiler-generated structures known as garbage collection maps. These maps contain which machine registers and locations of the stack frame the objects are stored. Generating these maps for every instruction would require large amounts of memory and therefore is infeasible. Moreover, object references may be inconsistent across particular instructions. As an alternative, these maps are created only for specific locations known as *safepoint polls*. Safepoint polls are generated at non-inlined method entry/exits, and non-counted loop backedges (HOHENSEE, 2006).

Whenever the JVM needs to perform an operation that requires a safepoint (e.g., garbage collection), it must bring all threads to a safepoint. It does so by posting a safepoint request. Each executing thread checks the request at the next safepoint poll and pauses its execution. The time needed to bring a set of threads to a safepoint is known as *time to safepoint*. Once all threads are blocked, the VM performs its operation and resumes the interrupted threads when finished.

Safepoint Bias JVMTI provides `GetStackTrace`, `GetThreadListStackTrace` and `GetAllStackTraces` functions. These functions can be used to get information on the call stack of threads. These operations require threads to be in a safepoint. Several Java profilers are built around these functions, resulting in profiles with *safepoint bias* (MYTKOWICZ et al., 2010). These profilers are biased towards the location of the next safepoint poll, resulting in, for example, profiles not showing functions that do not have polls, despite their high CPU usage (WAKART, 2016). These profilers also introduce additional overhead associated with time to safepoint.

Open-source profilers that mitigate the safepoint bias exists (NISBET et al., 2019). `perf-map-agent` (RUDOLPH, 2012) is capable of profiling kernel, native, and JIT-generated code by producing JIT stack maps for Linux's `perf` tool. However, it cannot profile interpreted code. `honest-profiler` (WARBURTON, 2014) uses the `AsyncGetCallTrace` undocumented API to circumvent JVMTI's safepoint strategy. It can trace interpreted code but not kernel and native code. `async-profiler` (PANGIN, 2016) is a hybrid of both approaches and can sample native, kernel, interpreted, and JIT-compiled code.

2.4 ELASTICSEARCH

Elasticsearch is an open-source search engine built atop Apache Lucene (BIAŁECKI et al., 2012), a Java library that executes indexing and querying/scoring functions. Like Lucene, Elasticsearch is written in Java and is mainly responsible for serving the search results through an HTTP interface while allowing Lucene to scale among clusters.

As a distributed application, multiple nodes or machines comprise an Elastichsearch cluster. This cluster handles all the non-Lucene tasks, such as creating and replicating shards, query distribution, and maintenance. Each node contains one or more shards, where each holds a search index.

2.5 FREQUENCY SCALING

Dynamic Frequency Scaling (DFS) is a technique for improving power efficiency on modern multi-core systems. The operating frequency of a particular core increases or decreases according to the application's demand. Given that power consumption is linearly proportional to core frequency, a lower frequency results in less power consumption.

Linux implements DFS in its Ondemand governor (PALLIPADI; STARIKOVSKIY, 2006), which attempts to minimize energy consumption by changing the CPU speed according to processor utilization thresholds. The Ondemand governor has been Linux's default since kernel 3.4.

The kernel also provides a userspace governor, in which the user chooses a fixed frequency.

2.6 RELATED WORK

2.6.1 Machine-Code Instrumentation

Static instrumentation was pioneered by ATOM (SRIVASTAVA; EUSTACE, 1994). It applies an instrumentation routine to compiled code, wherein tool developers can insert calls into their analysis functions. Following works, such as EEL (LARUS; SCHNARR, 1995), and Pebil (LAURENZANO et al., 2010) are conceptually similar but improve on usability, platform support, and functionality.

Pin (LUK et al., 2005), Valgrind (NETHERCOTE; SEWARD, 2007), and DynamoRIO (BRUENING; AMARASINGHE, 2004) are popular trace-based dynamic instrumentation alternatives. They operate by recompiling code with the extra instrumentation as the program executes. The approach can perform worse than static methods but has better ergonomics, including the significant advantage of manipulating every reachable code path, including dynamically linked and dynamically-generated code.

2.6.2 Bytecode Instrumentation

The mentioned systems were built with native applications in mind and cannot manipulate bytecode. Although native code's dynamic instrumentation can influence the interpreter and JIT-compiled code, it cannot expose the specifics of the virtual machine, such as which Java method or variable is accessed.

BCEL (DAHM, 1999), ASM (BRUNETON et al., 2002), Javassist (CHIBA; NISHIZAWA, 2003) and SOOT (VALLéE-RAI et al., 2010) are popular static instrumentation libraries for JVM bytecode. These tools and their augmentation routines are written in pure Java and can be used during dynamic instrumentation through events exposed by either JVMTI's agents or Java agents. The former uses native code and executes before the JVM bootstraps itself, while the latter is written in Java and, therefore, cannot instrument core runtime classes.

FERRARI (BINDER et al., 2007) is an instrumentation framework built as a hybrid of static and dynamic techniques. It statically instruments core classes while using a Java agent for the other ones. DiSL (MAREK et al., 2012) achieves full dynamic instrumentation by using a separate JVM process for manipulating bytecode. It uses a JVMTI agent to intercept class loading and forward it to another bootstrapped VM.

JNIF (MASTRANGELO; HAUSWIRTH, 2014) was created to foster native alternatives. It is a static instrumentation library written in and for use by native code. JVMTIPROF uses JNIF and is, to our knowledge, the first fully native dynamic instrumentation framework for Java.

2.6.3 Probe-Based Instrumentation

JVMTIPROF's instrumentation interface is probe-based, notifying clients of specific events instead of providing the power of arbitrary augmentation. Sofya (KINNEER et al., 2007), RoadRunner (FLANAGAN; FREUND, 2010), and Btrace (SUNDARARAJAN; BALASUBRAMANIAN, 2007) are examples of existing probe-based instrumenta-

tion systems for Java. Byteman (DINN, 2011) provides an efficient Java-based domain-specific language for probing methods. Linux exposes Byteman's probes in SystemTap's (PRASAD et al., 2005) scripting language that compiles into kernel modules.

```
package com.waldo;

class JNIExample
{
  native int foo(int bar);

  static {
    System.
      loadLibrary("qux");
  }
}
```

Listing 2.1: Example of a Java program containing the declaration of a native method `foo`. The program also loads the dynamic library `qux` during class loading.

```
#include <jni.h>

extern "C" JNIEXPORT
jint JNICALL
Java_com_waldo_JNIExample_foo(
    JNIEnv* jni_env,
    jobject JNIExample_obj,
    jint bar)
{
  return bar * 2;
}
```

Listing 2.2: Example of a C++ program exporting a symbol that can be bound to a Java native method. The exported symbol contains the `Java` prefix, the package, class, and the method names defined in the Java program. The native method receives the JNI environment, the Java object instance used in the method call, and the `bar` parameter defined in the Java declaration. This program can be compiled into a dynamic link library, e.g., `libqux.so` and loaded into the Java program.

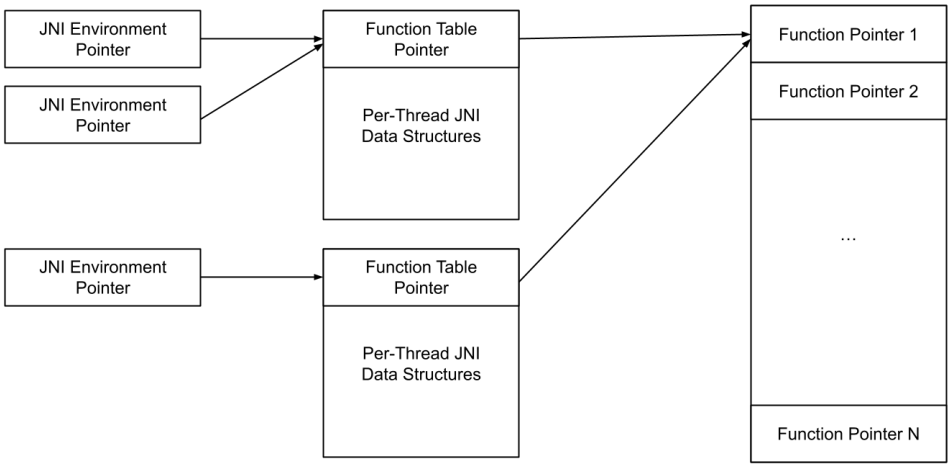

Figure 2.1: The function table approach. Agents have access to an `JNIEnv*`, which points to a per-thread JNI environment. This per-thread environment contains a pointer to a function table shared between the environments. The function table holds the JNI interface that clients can use to interact with the JVM.

Listing 2.3: Example of a JVMTI agent that intercepts every Java method entry. Agent setup creates a JVMTI environment and requests the necessary capabilities from it. Then, the method entry event is enabled, and its callbacks are set. Error handling is omitted for brevity.

```
#include <jvmti.h>

JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm,
             char *options,
             void *reserved)
{
    jvmtiEnv *jvmti;

    vm->GetEnv((void **)&jvmti, JVMTI_VERSION);

    jvmtiCapabilities caps;
    memset(&caps, 0, sizeof(caps));
    caps.can_generate_method_entry_events = true;
    jvmti->AddCapabilities(&caps);

    jvmtiEventCallbacks callbacks;
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.MethodEntry = OnMethodEntry;

    jvmti->SetEventNotificationMode(
            JVMTI_ENABLE, JVMTI_EVENT_METHOD_ENTRY, NULL);
    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));

    return 0;
}

void JNICALL
OnMethodEntry(jvmtiEnv *jvmti_env,
              JNIEnv *jni_env,
              jthread thread,
              jmethodID method)
{
    printf("method␣%p␣called␣by␣thread␣%p\n",
            (void*) method, (void*) thread);
}
```

Chapter 3

METHODOLOGY

In this chapter, we present the design (Section 3.1) and implementation (Section 3.2) of JVMTIPROF.

3.1 DESIGN

JVMTIPROF follows a similar design to the JVMTI. Native agents create environments, and those environments have capabilities, events, and other functionalities. Once the environment is disposed of, all the associated capabilities are relinquished and events disabled.

3.1.1 Startup & Shutdown

JVMTIPROF must be used in conjunction with a JVMTI environment. During agent loading, the `jvmtiProf_Create` function can be invoked, which modifies the existing JVMTI environment and returns an accompanying JVMTIPROF environment. JVMTI functionality can continue to be accessed as usual.

During agent shutdown, the JVMTIPROF environment must be disposed of through the `DisposeEnvironment` function. Unlike JVMTI, the disposal must be done explicitly since the JVM does not know about JVMTIPROF. However, if the JVMTI environment is disposed of explicitly (through its `DisposeEnvironment`), the associated JVMTIPROF environment is automatically destroyed.

3.1.2 Functionality

JVMTIPROF provides functionality to intercept individual methods, sample the application execution, and obtain accurate call stack traces.

Similarly to the JVMTI, events can be set through the `SetEventNotificationMode` and `SetEventCallbacks` functions. Capabilities necessary for the event to work properly must be added through the `AddCapabilities` function.

Appendix A presents details on the programming interface. An example agent that samples execution and prints call stack traces is shown in listing 3.1.

Execution Sampling To sample the application, an agent must add the `can_generate_sample_execution_events` capability and set an event callback for `SampleExecution`. The `SetExecutionSamplingInterval` function can adjust the sampling interval.

Call stack traces can be obtained by using `GetStackTraceAsync`. The `can_get_stack_trace_asynchronously` capability must be set. Unlike JVMTI's `GetStackTrace`, this function is async-signals-safe and does not require a safepoint.

Listing 3.1: Example agent that uses JVMTIPROF to sample the application and print its call trace. Error handling is omitted for brevity.

```
jvmtiProfEnv *jvmtiprof;

JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm,
             char *options,
             void *reserved)
{
    jvmtiEnv *jvmti;

    vm->GetEnv((void **)&jvmti, JVMTI_VERSION);

    jvmtiProf_Create(vm, jvmti, &jvmtiprof, JVMTIPROF_VERSION);

    jvmtiProfCapabilities caps;
    memset(&caps, 0, sizeof(caps));
    caps.can_generate_sample_execution_events = true;
    caps.can_get_stack_trace_asynchronously = true;
    jvmtiprof->AddCapabilities(&caps);

    jvmtiProfEventCallbacks callbacks;
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.SampleExecution = OnSampleExecution;
    jvmtiprof->SetEventCallbacks(&callbacks, sizeof(callbacks));

    jvmtiprof->SetEventNotificationMode(
            JVMTI_ENABLE, JVMTIPROF_EVENT_SAMPLE_EXECUTION,
            NULL);

    const jlong ms_to_ns = 1000000;
    jvmtiprof->SetExecutionSamplingInterval(
            1000 * ms_to_ns);

    return 0;
}
```

```
JNIEXPORT void JNICALL
Agent_OnUnload(JavaVM *vm)
{
    jvmtiprof->DisposeEnvironment();
}

void JNICALL
OnSampleExecution(jvmtiProfEnv *jvmtiprof,
                  jvmtiEnv *jvmti,
                  JNIEnv *jni,
                  jthread thread)
{
    char printbuf[256];
    jvmtiProfError err;

    const jint depth = 1;
    const jint max_frame_count = 1;
    jvmtiProfFrameInfo frames[max_frame_count];
    jint frame_count;

    // fprintf isn't async-signal-safe, thus we use write.
    snprintf(printbuf, sizeof(printbuf), "sampling...\n");
    write(STDERR_FILENO, printbuf, strlen(printbuf));

    err = jvmtiprof->GetStackTraceAsync(
            depth, max_frame_count, frames, &frame_count);
    if(err != JVMTIPROF_ERROR_NONE)
        return;

    for(int i = 0; i < frame_count; ++i)
    {
        snprintf(printbuf, sizeof(printbuf),
                 "\tframe␣%d;␣bci␣offset␣%d;␣method␣%p\n",
                 i, frames[i].offset,
                 (void *)frames[i].method);
        write(STDERR_FILENO, printbuf, strlen(printbuf));
    }
}
```

.

Method Interception JVMTI provides the `MethodEntry` event to intercept *every* Java method call, significantly degrading application performance. JVMTIPROF introduces

a low-overhead alternative capable of intercepting individual methods.

Clients can hook methods of interest by passing their signature to `SetMethodEventFlag`. Entry and exit events can be controlled independently of each other. The associated event notification, callbacks (i.e. `SpecificMethodEntry`, `SpecificMethodExit`) and capabilities (i.e. `can_intercept_methods`) must also be set.

3.2 IMPLEMENTATION

3.2.1 JVMTI Injection

JVMTIPROF uses events and capabilities from JVMTI to implement some of its functionalities. Method interception, for instance, is performed through JVMTI's `ClassFileLoadHook` event. This poses a challenge because an API client would not be able to use that event for other purposes. Therefore JVMTI needs to be modified for its functionality and JVMTIPROF to co-exist.

The changes are achieved by redirecting function pointers in the JVMTI function table to JVMTIPROF-managed functions. The `DisposeEnvironment`, `SetEventCallbacks`, `SetEventNotificationMode`, `RetransformClasses`, `RedefineClasses`, `GetCapabilities`, `AddCapabilities` and `RelinquishCapabilities` functions are hooked. Patching occurs during `jvmtiProf_Create`.

Environment Disposal JVMTI's `DisposeEnvironment` is hooked such that it also disposes of the associated JVMTIPROF environment.

Event Management The `SetEventCallbacks` function is patched such that the client does not replace event callbacks used by JVMTIPROF. Instead, the pointer to the client event handler is stored, and whenever JVMTIPROF's callback is called, the client handler is also invoked.

The `SetEventNotificationMode` hook works similarly. It avoids replacing notification modes in use by JVMTIPROF and instead stores them internally. When an event used by both occurs, the modes set by the client are inspected to decide whether the callback stored in `SetEventCallbacks` should be called.

Capability Management The capabilities functions are modified to avoid exposing capabilities set by JVMTIPROF to the client. JVMTI's `GetCapabilities` should return an empty set of capabilities even though JVMTIPROF has set some of them (e.g. `can_retransform_classes`). `RelinquishCapabilities` must not relinquish capabilities possessed by JVMTIPROF. The state of relinquished capabilities is maintained internally by JVMTIPROF such that `GetCapabilities` can return a view according to the client's expectations. `AddCapabilities` is also modified for this purpose.

Class Redefinition `RetransformClasses` and `RedefineClasses` may need to be replaced to force allocation of method identifiers after class redefinition and retransformation. This is explained in detail in Section 3.2.4.

3.2.2 Method Interception

JVMTIPROF can notify clients about method calls of interest. This is achieved by instrumenting the method's bytecode to include calls to a JVMTIPROF-managed native function at its prologue and epilogue. The method call event is sent upstream when the native function is called.

Agents set the target methods through `SetMethodEventFlag`. When the bytecode of the class associated with the method is loaded, it is instrumented to include JVMTIPROF's internal calls. Class loading is intercepted through JVMTI's `ClassFileLoadHook` event. If the class is already loaded, the instrumentation is forced by calling JVMTI's `RetransformClasses` on the said class.

```
public class JVMTIPROF {
     public static native void onMethodEntry(long hookPtr);
     public static native void onMethodExit(long hookPtr);
}

public class Example {

     static final long sumHookPtr = /* determined at runtime */;

     public int sum(int a, int b) {
          JVMTIPROF.onMethodEntry(sumHookPtr);
          try {
               return a + b;
          } finally {
               JVMTIPROF.onMethodExit(sumHookPtr);
          }
     }
}
```

Listing 3.2: Example instrumentation applied by method interception. Instrumented code is in gray. The `sum` method is modified such that JVMTIPROF is notified about entries and exits on it.

An illustration of the instrumentation performed is given in Listing 3.2. JVMTIPROF defines a new class with native methods to communicate back with C++. The method exit notification is enclosed in a `try-finally` block, so exceptions do not cause the event to be missed. An internal identifier of the hooked method is passed as an argument to JVMTIPROF, which is sent upstream, enabling clients to identify which method of interest has been invoked.

3.2.3 Execution Sampling

JVMTIPROF provides an event to simplify the setup of sampling profilers. It is implemented using a high-precision CPU timer (`CLOCK_PROCESS_CPUTIME_ID`) and a no-

tification signal (`SIGPROF`). The timer is set to expire at certain intervals, as set by `SetExecutionSamplingInterval`, and once expired, the application receives a signal at the thread that caused the timer to expire.

The signal handler then sends the event upstream. The event handler must be async-signal-safe, restricting the client to primitive operations and system calls. It must also be careful to consume little CPU time since it blocks the calling thread, safepoint polling, and receiving other signals. Usually, the handler must push the information of interest (e.g., stack trace) into a queue, which can be consumed, e.g., by a worker thread in a non-async-signal context.

3.2.4 Call Stack Trace

JVMTI provides the `GetStackTrace` function to obtain call stack traces of the application. However, it cannot be used during sampling because of its non-async-signal-safe nature.

JVMTIPROF introduces `GetStackTraceAsync`, a function capable of taking stack traces in async-signal contexts. This function can take traces at non-safepoints, such as in the sampling event handler.

AsyncGetCallTrace Oracle's HotSpot JVM provides an unsupported internal API capable of async-signal-safe stack tracing. It is famously known as `AsyncGetCallTrace` (AGCT for short). JVMTIPROF uses such API to implement its `GetStackTraceAsync`. As a downside, this functionality is not supported in other JVM implementations.

AGCT requires all methods identifiers to be pre-allocated because it cannot do so lazily in an async-signal-safe manner. JVMTI's `ClassLoad` and `ClassPrepare` events are used for this purpose. Additionally, `RetransformClasses` and `RedefineClasses` are hooked in the JVMTI function table to reallocate identifiers after class redefinition.

Debugging Non-Safepoints For AGCT to work, the JIT must generate metadata such as GC maps for every compiled instruction, which is achieved by setting the `-XX:+DebugNonSafepoints` command-line option. JVMTIPROF sets this flag by enabling JVMTI's `CompileMethodLoad` event. This event has the side-effect of enabling debugging information at non-safepoint locations.

Chapter 4

EVALUATION

In this chapter, we evaluate JVMTIPROF by using it to instrument Elasticsearch. A profile-guided frequency scaling solution (MEDEIROS et al., 2021) is adapted to use JVMTIPROF instead of JVMTI. We then measure the overhead and identify whether JVMTIPROF can produce the same gains as the baseline despite its abstractions.

4.1 EXPERIMENTAL SETUP

We conduct experiments on a bare-metal server instance provided by the Chameleon Cloud service (KEAHEY et al., 2020). The server consists of an Intel Xeon Gold 6126 (Skylake architecture) with 196GB of DRAM and a 220GB SSD disk. We run Ubuntu 20.04 (Linux kernel 5.4) and Elasticsearch version 6.5.4 under OpenJDK 8u342. For the search index database, we use the Wikipedia dump version `enwiki-latest-pages-articles` downloaded in September 2022.

The CPU has 24 physical cores equally divided between two sockets. Intel's Hyper-threading technology is turned off, so we only consider the physical core count. To isolate the network effects in the shared experimental platform, we configure socket 0 to run the server-side applications (Elasticsearch) and socket 1 to execute the load generator (FABAN).

Energy measurement is done through Intel's Running Average Power Limit (RAPL) interface. We measure the energy consumption counter at the beginning and end of the experiment, and the difference between those two values is considered the energy consumed for that particular experiment.

The client, FABAN (Sun Microsystems, 2006a), is responsible for generating query requests with randomized keywords, ranging from 1 to 18 keywords, under a Zipfian distribution.

The load generation process consisted of 20-minute runs of continuous search queries. The FABAN workload generator is used to create four clients, with each client issuing, on average, one request per second. The goal is to analyze the behavior of each solution,

considering a single request at a time. An additional configuration of three times the requests-per-seconds is used to explore loads similar to that of production environments.

We perform six runs for each combination of load and solution and report the mean and standard deviation of the 99-percentile latency and energy consumption.

4.2 BASELINE

We use the Linux Ondemand power governor and Hurryup (MEDEIROS et al., 2021) as baselines for our experiments. Hurryup is a profile-guided frequency scaling solution for Elasticsearch that outperforms Ondemand on energy consumption while attaining acceptable levels of tail latency constraints.

Hurryup works by instrumenting Elasticsearch's hot functions to produce enter and exit events. A runtime scheme uses these events to adapt the individual processor core frequencies. In our experiments, we observe Hurryup saving up to 25% in energy compared to Linux's Ondemand governor, as seen in Figure 4.1.

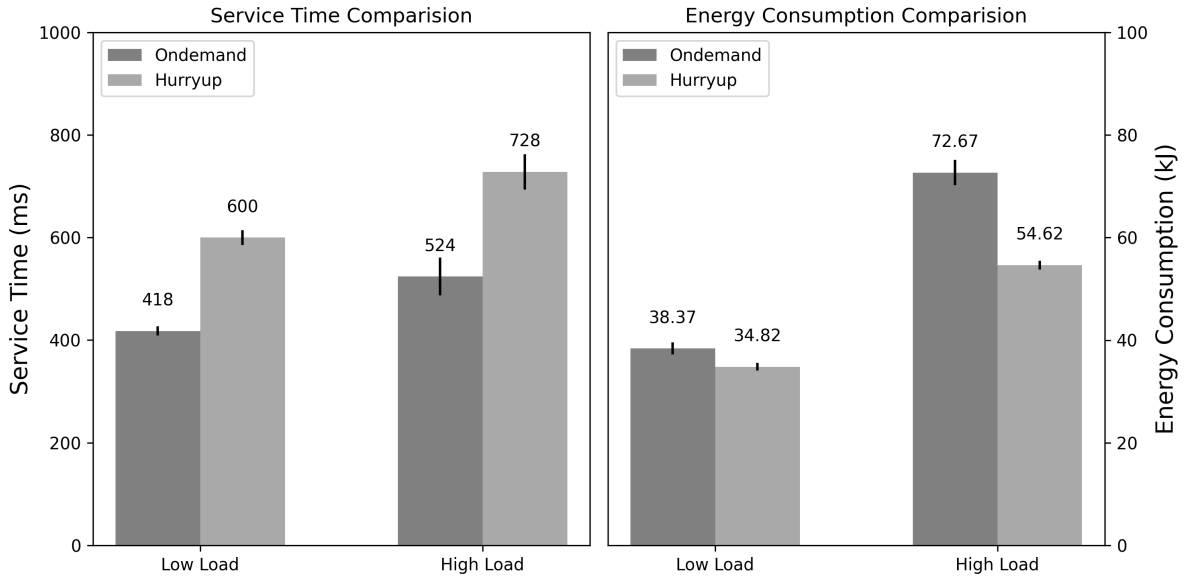

Figure 4.1: An experiment comparing Linux's Ondemand governor against Hurryup's frequency scaling scheme. The left plot presents Elasticsearch's service time, and the right plot the energy consumption of the CPU while executing search requests. Results are shown for both a low and high server load.

4.3 RESULTS

We modify Hurryup to use JVMTIPROF's instrumentation methods and re-execute the experiments as reported in Figure 4.2. We notice that JVMTIPROF can be similar to or slightly worse than JVMTI in the context of Hurryup but can still achieve energy gains compared to Ondemand.

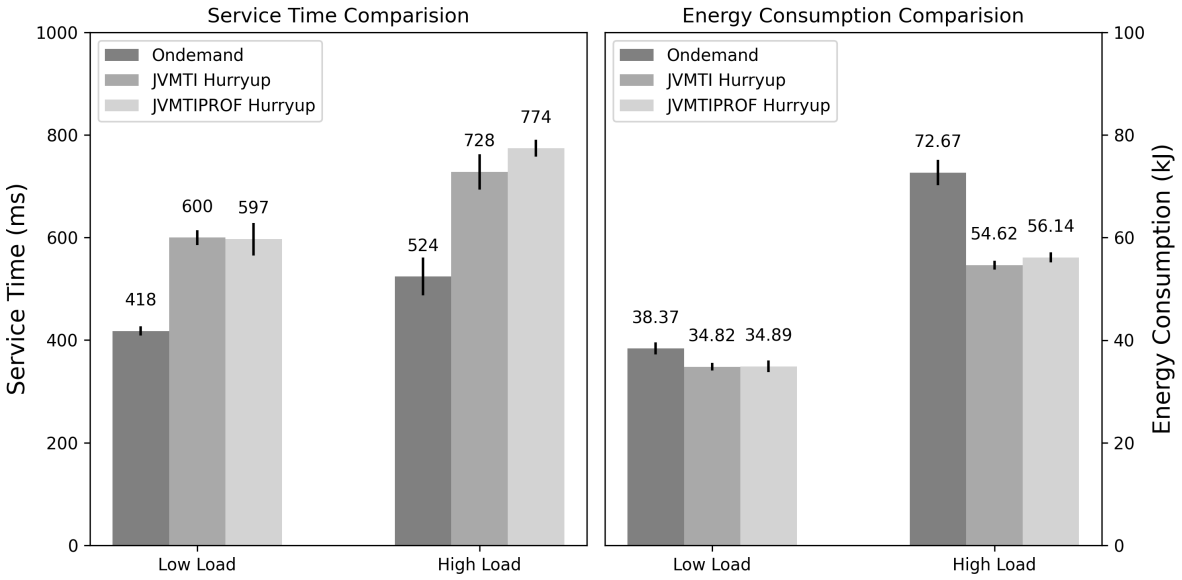

Figure 4.2: An experiment comparing Hurryup's performance when implemented with JVMTIPROF versus the baseline. The left plot presents Elasticsearch's service time, and the right plot the energy consumption of the CPU while executing search requests. Results are shown for both a low and high server load.

At a lower load, the JVMTI and JVMTIPROF implementations do not present a statistically significant difference in service time and energy consumption. When tripling the load, the JVMTIPROF version can be 6% worse in service time and 2.78% worse in energy consumption.

We also compare the overhead of instrumenting Elasticsearch's hot functions, excluding Hurryup-specific logic. Results are presented in Figure 4.3. There is no statistically significant difference between instrumenting or not instrumenting said functions with either JVMTI or JVMTIPROF.

These results also show that the context in which the instrumentation is used may cause performance differences. When instrumenting to run Hurryup at a higher load, JVMTIPROF performed worse than JVMTI, but no difference is seen at lower loads or with scheduling logic removed.

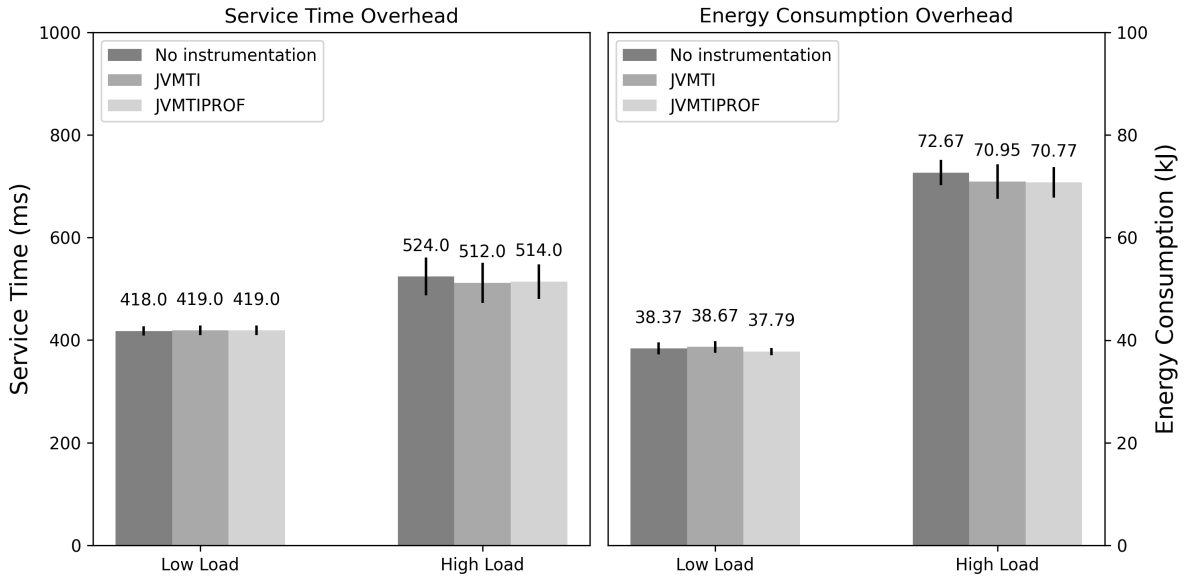

Figure 4.3: An experiment comparing the performance of Elasticsearch when their hot functions are kept as-is or hooked with JVMTI and JVMTIPROF. The left plot presents Elasticsearch's service time, and the right plot the energy consumption of the CPU while executing search requests. Results are shown for both a low and high server load.

Chapter 5

CONCLUDING REMARKS

This work presented the design and implementation of JVMTIPROF, an extension to JVMTI with higher-level functionalities. We evaluated the framework in an existing system and found minimal penalties compared to pure JVMTI.

We have also shown how JVMTI can be extended by manipulating its function table. In our experience, the technique is overly complex and hard to maintain, suggesting alternative solutions should be favored, such as the creation of a JVMTI environment private to the extended interface.

In the future, cross-platform support and additional functionalities can be explored. The present components have room for improvement as well. Call stack tracing does not expose kernel, and native functions, which can be solved by the hybrid approach of Pangin (2016). Method interception may be more performant if we generate specialized code for each hook, avoiding the intermediate lookup by an integer identifier. Finally, execution sampling may benefit from a per-thread timer instead of an application-wide one.

BIBLIOGRAPHY

Azul Systems. *Azul Platform Prime*. 2022. <https://www.azul.com/products/zing>. [Online; accessed 19-Oct-2022].

BIAŁECKI, A. et al. Apache lucene 4. In: *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*. [S.l.: s.n.], 2012. p. 17–24.

BINDER, W.; HULAAS, J.; MORET, P. Advanced java bytecode instrumentation. In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. New York, NY, USA: Association for Computing Machinery, 2007. (PPPJ '07), p. 135–144. ISBN 9781595936721. Disponível em: <https://doi.org/10.1145/1294325.1294344>.

BRUENING, D.; AMARASINGHE, S. *Efficient, transparent, and comprehensive runtime code manipulation*. Tese (Doutorado) — Massachusetts Institute of Technology, Department of Electrical Engineering, 2004.

BRUNETON, E.; LENGLET, R.; COUPAYE, T. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, v. 30, n. 19, 2002.

CHIBA, S.; NISHIZAWA, M. An easy-to-use toolkit for efficient java bytecode translators. In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*. Berlin, Heidelberg: Springer-Verlag, 2003. (GPCE '03), p. 364–376. ISBN 3540201025.

DAHM, M. Byte code engineering. In: *JIT'99*. [S.l.]: Springer, 1999. p. 267–277.

DINN, A. E. Flexible, dynamic injection of structured advice using byteman. In: *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development Companion*. New York, NY, USA: Association for Computing Machinery, 2011. (AOSD '11), p. 41–50. ISBN 9781450306065. Disponível em: <https://doi.org/10.1145/1960314.1960325>.

Eclipse Foundation. *Eclipse OpenJ9*. 2022. <https://projects.eclipse.org/projects/technology.openj9>. [Online; accessed 19-Oct-2022].

FLANAGAN, C.; FREUND, S. N. The roadrunner dynamic analysis framework for concurrent programs. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA: Association for Computing Machinery, 2010. (PASTE '10), p. 1–8. ISBN 9781450300827. Disponível em: <https://doi.org/10.1145/1806672.1806674>.

GHERARDI, L.; BRUGALI, D.; COMOTTI, D. A java vs. c++ performance evaluation: A 3d modeling benchmark. In: *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Berlin, Heidelberg: Springer-Verlag, 2012. (SIMPAR'12), p. 161–172. ISBN 9783642343261. Disponível em: <https://doi.org/10.1007/978-3-642-34327-8_17>.

GRAHAM, S. L.; KESSLER, P. B.; MCKUSICK, M. K. Gprof: A call graph execution profiler. Association for Computing Machinery, New York, NY, USA, p. 120–126, 1982. Disponível em: <https://doi.org/10.1145/800230.806987>.

HOHENSEE, P. *The Hotspot Java Virtual Machine*. 2006. <https://www.cs.princeton.edu/picasso/mats/HotspotOverview.pdf>. [Online; accessed 18-Oct-2022].

IVANKOVIć, M. et al. Code coverage at google. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 955–963. ISBN 9781450355728. Disponível em: <https://doi.org/10.1145/3338906.3340459>.

KEAHEY, K. et al. Lessons learned from the chameleon testbed. In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. [S.l.]: USENIX Association, 2020.

KELL, S. et al. The jvm is not observable enough (and what to do about it). In: *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*. New York, NY, USA: Association for Computing Machinery, 2012. (VMIL '12), p. 33–38. ISBN 9781450316330. Disponível em: <https://doi.org/10.1145/2414740.2414747>.

KEMPF, T.; KARURI, K.; GAO, L. Software instrumentation. John Wiley & Sons, Ltd, p. 1–11, 2008. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse386>.

KERRISK, M. *signal-safety(7) Linux manual page*. [S.l.], 2016. Disponível em: <https://man7.org/linux/man-pages/man7/signal-safety.7.html>.

KINNEER, A.; DWYER, M. B.; ROTHERMEL, G. Sofya: Supporting rapid development of dynamic program analyses for java. In: *Companion to the Proceedings of the 29th International Conference on Software Engineering*. USA: IEEE Computer Society, 2007. (ICSE COMPANION '07), p. 51–52. ISBN 0769528929. Disponível em: <https://doi.org/10.1109/ICSECOMPANION.2007.68>.

KOTZMANN, T. et al. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, Association for Computing Machinery, New York, NY, USA, v. 5, n. 1, may 2008. ISSN 1544-3566. Disponível em: <https://doi.org/10.1145/1369396.1370017>.

KUMAR, K.; DAHIYA, S. Programming languages: A survey. *International Journal on Recent and Innovation Trends in Computing and Communication*, v. 5, n. 5, p. 307–313, 2017.

LARUS, J. R.; SCHNARR, E. Eel: Machine-independent executable editing. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1995. (PLDI '95), p. 291–300. ISBN 0897916972. Disponível em: <https://doi.org/10.1145/207110.207163>.

LAURENZANO, M. A. et al. Pebil: Efficient static binary instrumentation for linux. In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. [S.l.: s.n.], 2010. p. 175–183.

LI, W. H.; WHITE, D. R.; SINGER, J. Jvm-hosted languages: They talk the talk, but do they walk the walk? In: *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: Association for Computing Machinery, 2013. (PPPJ '13), p. 101–112. ISBN 9781450321112. Disponível em: <https://doi.org/10.1145/2500828.2500838>.

LIN, Y. et al. Stop and go: Understanding yieldpoint behavior. In: *Proceedings of the 2015 International Symposium on Memory Management*. New York, NY, USA: Association for Computing Machinery, 2015. (ISMM '15), p. 70–80. ISBN 9781450335898. Disponível em: <https://doi.org/10.1145/2754169.2754187>.

LINDHOLM, T. et al. *The Java Virtual Machine Specification, Java SE 19 Edition*. [s.n.], 2022. Disponível em: <https://docs.oracle.com/javase/specs/jvms/se19/jvms19.pdf>.

LOPEZ, J. et al. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security*, v. 1, 06 2017.

LUK, C.-K. et al. Pin: Building customized program analysis tools with dynamic instrumentation. Association for Computing Machinery, New York, NY, USA, p. 190–200, 2005. Disponível em: <https://doi.org/10.1145/1065010.1065034>.

MAREK, L. et al. Disl: A domain-specific language for bytecode instrumentation. In: *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*. New York, NY, USA: Association for Computing Machinery, 2012. (AOSD '12), p. 239–250. ISBN 9781450310925. Disponível em: <https://doi.org/10.1145/2162049.2162077>.

MASTRANGELO, L.; HAUSWIRTH, M. Jnif: Java native instrumentation framework. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. New York, NY, USA: Association for Computing Machinery, 2014. (PPPJ '14), p. 194–199. ISBN 9781450329262. Disponível em: <https://doi.org/10.1145/2647508.2647516>.

MEDEIROS, D. A. de; AMORIM, D. das M.; PETRUCCI, V. Profile-guided frequency scaling for latency-critical search workloads. In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. [S.l.: s.n.], 2021. p. 304–313.

MOSELEY, T. et al. Shadow profiling: Hiding instrumentation costs with parallelism. In: *Proceedings of the International Symposium on Code Generation and Optimization*. USA: IEEE Computer Society, 2007. (CGO '07), p. 198–208. ISBN 0769527647. Disponível em: <https://doi.org/10.1109/CGO.2007.35>.

MYTKOWICZ, T. et al. Evaluating the accuracy of java profilers. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2010. (PLDI '10), p. 187–197. ISBN 9781450300193. Disponível em: <https://doi.org/10.1145/1806596.1806618>.

NETHERCOTE, N.; SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007. (PLDI '07), p. 89–100. ISBN 9781595936332. Disponível em: <https://doi.org/10.1145/1250734.1250746>.

NISBET, A. et al. Profiling and tracing support for java applications. In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (ICPE '19), p. 119–126. ISBN 9781450362399. Disponível em: <https://doi.org/10.1145/3297663.3309677>.

O'CONNOR, J. M.; TREMBLAY, M. Picojava-i: The java virtual machine in hardware. *IEEE Micro*, IEEE Computer Society Press, Washington, DC, USA, v. 17, n. 2, p. 45–53, mar 1997. ISSN 0272-1732. Disponível em: <https://doi.org/10.1109/40.592314>.

Oracle. *AdvancedThresholdPolicy source code from OpenJDK 7*. 2011. <http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/5d8f5a6dced7/src/share/vm/runtime/advancedThresholdPolicy.hpp>. [Online; accessed 19-Oct-2022].

Oracle. *Java HotSpot™ Virtual Machine Performance Enhancements*. 2011. <https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html>. [Online; accessed 19-Oct-2022].

Oracle. *Java Native Interface Specification*. 2022. <https://docs.oracle.com/en/java/javase/19/docs/specs/jni/index.html>. [Online; accessed 19-Oct-2022].

Oracle. *JVM™ Tool Interface*. 2022. <https://docs.oracle.com/en/java/javase/19/docs/specs/jvmti.html>. [Online; accessed 19-Oct-2022].

Oracle. *OpenJDK JDK 19 General-Availability Release*. 2022. <https://jdk.java.net/19>. [Online; accessed 19-Oct-2022].

PALECZNY, M.; VICK, C.; CLICK, C. The java hotspot™ server compiler. In: *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*. USA: USENIX Association, 2001. (JVM'01), p. 1.

PALLIPADI, V.; STARIKOVSKIY, A. The ondemand governor. In: *Proceedings of the Linux Symposium*. [S.l.: s.n.], 2006. v. 2, n. 00216, p. 215–230.

PANGIN, A. *async-profiler*. 2016. <https://github.com/jvm-profiling-tools/async-profiler>. [Online; accessed 18-Oct-2022].

PATHAK, A.; HU, Y. C.; ZHANG, M. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2012. (EuroSys '12), p. 29–42. ISBN 9781450312233. Disponível em: <https://doi.org/10.1145/2168836.2168841>.

PETTIS, K.; HANSEN, R. C. Profile guided code positioning. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1990. (PLDI '90), p. 16–27. ISBN 0897913647. Disponível em: <https://doi.org/10.1145/93542.93550>.

PONDER, C.; FATEMAN, R. J. Inaccuracies in program profilers. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., USA, v. 18, n. 5, p. 459–467, may 1988. ISSN 0038-0644. Disponível em: <https://doi.org/10.1002/spe.4380180506>.

PRASAD, V. et al. Locating system problems using dynamic instrumentation. In: *2005 Ottawa Linux Symposium*. [S.l.: s.n.], 2005. p. 49–64.

PUFEK, P.; GRGIC, H.; MIHALJEVIC, B. Analysis of garbage collection algorithms and memory management in java. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. [S.l.: s.n.], 2019. p. 1677–1682.

REISS, S. P. Visualizing the java heap to detect memory problems. In: *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. [S.l.: s.n.], 2009. p. 73–80.

RUDOLPH, J. *perf-map-agent*. 2012. <https://github.com/jvm-profiling-tools/perf-map-agent>. [Online; accessed 18-Oct-2022].

SEWARD, J.; NETHERCOTE, N. Using valgrind to detect undefined value errors with bit-precision. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USA: USENIX Association, 2005. (ATEC '05), p. 2.

SRIVASTAVA, A.; EUSTACE, A. Atom: A system for building customized program analysis tools. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing

Machinery, 1994. (PLDI '94), p. 196–205. ISBN 089791662X. Disponível em: <https://doi.org/10.1145/178243.178260>.

Sun Microsystems. *Faban Harness and Benchmark Framework*. 2006. <http://faban.org>. [Online; accessed 04-Oct-2022].

Sun Microsystems. *HotSpot Glossary of Terms*. 2006. <https://openjdk.org/groups/hotspot/docs/HotSpotGlossary.html>. [Online; accessed 18-Oct-2022].

SUNDARARAJAN, A.; BALASUBRAMANIAN, K. *btrace*. 2007. <https://github.com/btraceio/btrace>. [Online; accessed 01-Nov-2022].

TABOADA, G. L. et al. Java in the high performance computing arena: Research, practice and experience. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., USA, v. 78, n. 5, p. 425–444, may 2013. ISSN 0167-6423. Disponível em: <https://doi.org/10.1016/j.scico.2011.06.002>.

VALLéE-RAI, R. et al. Soot: A java bytecode optimization framework. In: *CASCON First Decade High Impact Papers*. USA: IBM Corp., 2010, (CASCON '10). p. 214–224. Disponível em: <https://doi.org/10.1145/1925805.1925818>.

VENNERS, B. *Inside the Java Virtual Machine*. 2. ed. McGraw-Hill, 2000. (Computing McGraw-Hill). ISBN 9780071350938. Disponível em: <https://www.artima.com/insidejvm/ed2/index.html>.

WAKART, N. *Safepoints: Meaning, Side Effects and Overheads*. 2015. <http://psy-lob-saw.blogspot.com/2015/12/safepoints.html>. [Online; accessed 18-Oct-2022].

WAKART, N. *Why (Most) Sampling Java Profilers Are Fucking Terrible*. 2016. <http://psy-lob-saw.blogspot.com/2016/02/why-most-sampling-java-profilers-are.html>. [Online; accessed 18-Oct-2022].

WARBURTON, R. *honest-profiler*. 2014. <https://github.com/jvm-profiling-tools/honest-profiler>. [Online; accessed 18-Oct-2022].

ZHAO, Q. et al. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In: *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 2008. (CC'08/ETAPS'08), p. 147–162. ISBN 3540787909.

Appendix

A

PROGRAMMING INTERFACE

This appendix documents the interface of JVMTIPROF in detail. It extends upon the documentation already provided by the JVM Tool Infrastructure (JVMTI). Therefore, familiarity with JVMTI's documentation is required to understand the following content.

JVMTIPROF must be used in conjunction with a JVMTI environment. A JVMTIPROF environment can be created by using the `jvmtiProf_Create` function.

Multiple JVMTIPROF environments can be injected into a single JVMTI environment. Each environment retains its own state; for example, each has its logging verbosity, capabilities, and registered events.

None of these functions is heap callback safe.

A.1 EXTENSION INJECTION

Create

```
jvmtiProfError jvmtiProf_Create(
            JavaVM* vm,
            jvmtiEnv* jvmti_env,
            jvmtiProfEnv** jvmtiprof_env_ptr,
            jvmtiProfVersion version);
```

DESCRIPTION

Creates a JVMTIPROF environment and injects it into the given JVMTI environment.

This function is on the global scope, i.e., not in the environment function table.

Functionalities of `jvmti_env` must not be used before invoking this function.

PHASES
This function may only be called during the OnLoad phase.

CAPABILITIES
Required functionality.

PARAMETERS

`vm` The virtual machine instance on which the `jvmti_env` is installed in.

`jvmti_env` The JVMTI environment to inject JVMTIPROF into.

`jvmtiprof_env_ptr` Pointer through which the injected JVMTIPROF environment pointer is returned.

`version` The version of the JVMTIPROF interface to be created.

ERROR CODES
This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_WRONG_PHASE`: The virtual machine is not in the OnLoad phase.

`JVMTIPROF_ERROR_UNSUPPORTED`: Either the provided JVMTIPROF interface `version` is not supported, or the version of the provided `jvmti_env` interface is not supported.

`JVMTIPROF_ERROR_NULL_POINTER`: Either `vm`, `jvmti_env`, or `jvmtiprof_env_ptr` is `NULL`.

Get Env

```
jvmtiProfError jvmtiProf_GetEnv(
            jvmtiEnv* jvmti_env,
            jvmtiProfEnv** jvmtiprof_env_ptr);
```

DESCRIPTION
Returns the JVMTIPROF environment injected into the given JVMTI environment.

`Create` must have been previously called on `jvmti_env`.

This function is on the global scope, i.e., not in the environment function table.

PHASES
This function may be called during any phase.

CAPABILITIES
Required functionality.

PARAMETERS

`jvmti_env` The JMVTI environment to extract JVMTIPROF from.

`jvmtiprof_env_ptr` Pointer through which the injected JMVTIPROF environment pointer is returned.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: Either `jvmti_env` or `jvmtiprof_env_ptr` is `NULL`.

`JVMTIPROF_ERROR_INVALID_ENVIRONMENT`: `jvmti_env` doesn't have an associated JVMTIPROF environment.

A.2 METHOD INTERCEPTION

Set Method Event Flag

```
typedef enum
{
    JVMTIPROF_METHOD_EVENT_ENTRY = 1,
    JVMTIPROF_METHOD_EVENT_EXIT = 2,
} jvmtiProfMethodEventFlag;

jvmtiProfError SetMethodEventFlag(
            jvmtiProfEnv* jvmtiprof_env,
            const char* class_name,
            const char* method_name,
            const char* method_signature,
            jvmtiProfMethodEventFlag flags,
            jboolean enable,
            jint* hook_id_ptr);
```

DESCRIPTION

Sets whether the given method should generate entry/exit events.

The `jvmtiProfMethodEventFlag` enumeration is a bitmask and can be combined in a single call of this function e.g. `JVMTIPROF_METHOD_EVENT_ENTRY| JVMTIPROF_METHOD_EVENT_EXIT`.

When `JVMTIPROF_METHOD_EVENT_ENTRY` is `enabled`, `SpecificMethodEntry` events are generated for the given method.

When `JVMTIPROF_METHOD_EVENT_EXIT` is `enabled`, `SpecificMethodExit` events are generated for the given method.

This function alone does not enable the events. It must be enabled by `SetEventNotificationMode` and callbacks set in `SetEventCallbacks`. See appendix A.4.

PHASES

This function may be called during any phase.

CAPABILITIES

- `can_intercept_methods` is necessary to enable `JVMTIPROF_METHOD_EVENT_ENTRY` and `JVMTIPROF_METHOD_EVENT_EXIT`.

PARAMETERS

`class_name` The name of the class to intercept.

`method_name` The method's name in the given class to intercept.

`method_signature` The signature of the method (i.e. parameter description) to intercept.

`flags` The flags to apply the `enable` boolean to.

`enable` Whether to enable (or disable) the events.

`hook_id_ptr` Pointer through which an identifier for the method interception is returned. This identifier can be used in the event to distinguish different intercepted methods.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: Either `class_name`, `method_name`, `method_signature` or `hook_id_ptr` is `NULL`.

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `flags` are not valid.

`JVMTIPROF_ERROR_MUST_POSSESS_CAPABILITY`: The environment does not possess the capability `can_intercept_methods`. Use `AddCapabilities`.

A.3 EXECUTION SAMPLING

Set Execution Sampling Interval

```
jvmtiProfError SetExecutionSamplingInterval(
            jvmtiProfEnv* jvmtiprof_env,
            jlong nanos_interval);
```

DESCRIPTION

Sets the CPU-time interval necessary for the `SampleExecution` event to happen.

Every `nano_interval` nanoseconds of CPU-time elapsed causes the `SampleExecution` to be called in the thread that expired the timer.

This function alone does not enable the event. It must be enabled by `SetEventNotificationMode` and callbacks set in `SetEventCallbacks`. See appendix A.4.

PHASES

This function may be called during any phase.

CAPABILITIES

- `can_generate_sample_execution_events` is necessary for this function to be used.

PARAMETERS

`nano_interval` Time interval in nanoseconds necessary for the `SampleExecution` event to be invoked.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: `nanos_interval` is less than `0`.

`JVMTIPROF_ERROR_MUST_POSSESS_CAPABILITY`: The environment does not possess the capability `can_generate_sample_execution_events`. Use `AddCapabilities`.

Get Stack Trace Async

```
typedef struct
{
    jint offset;
    jmethodID method;
} jvmtiProfFrameInfo;

jvmtiProfError GetStackTraceAsync(
            jvmtiProfEnv* jvmtiprof_env,
            jint depth,
            jint max_frame_count,
            jvmtiProfFrameInfo* frame_buffer,
            jint* count_ptr);
```

DESCRIPTION

Get information about the stack of a thread. If `max_frame_count` is less than the depth of the stack, the `max_frame_count` topmost frames are returned, otherwise the entire stack is returned. The topmost frames, those most recently invoked, are at the beginning of the returned buffer.

This function is async-signal-safe and does not require a safepoint, unlike JVMTI's `GetStackTrace`.

The returned information for each frame contains the `method` and bytecode `offset` being executed by the frame. If it is impossible to determine `offset`, its value is set to a

negative value. If set to `-1`, a native method is in execution. If set to `-2`, the reason for being unable to determine the offset is unknown.

The `offset` returned in a frame can be used in the bytecode returned by JVMTI's `GetBytecodes`. It can also be mapped to a line number by JVMTI's `GetLineNumberTable` if and only if JVMTI's `GetJLocationFormat` returns `JVMTI_JLOCATION_JVMBCI`.

See JVMTI's `GetStackTrace` for example usage.

This function can be used in the `SampleExecution` event to obtain the stack trace of the interrupted thread.

PHASES

This function may only be called during the live phase.

CAPABILITIES

- `can_get_stack_trace_asynchronously` is necessary for this function to work.

PARAMETERS

`depth` Maximum depth of the call stack trace.

`max_frame_count` Same as `depth`.

`frame_buffer` On return, this buffer is filled with stack frame information.

`count_ptr` On return, points to the number of records filled in.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_INTERNAL`: It was not possible to obtain the stack trace.

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `depth` or `max_frame_count` is less than `0`.

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `depth` and `max_frame_count` differs.

`JVMTIPROF_ERROR_NULL_POINTER`: `frame_buffer` or `count_ptr` is `NULL`.

A.4 EVENT MANAGEMENT

Set Event Notification Mode

```
jvmtiProfError SetEventNotificationMode(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiEventMode mode,
            jvmtiProfEvent event_type,
            jthread event_thread,
            ...);
```

DESCRIPTION

Controls the generation of events.

If `JVMTI_ENABLE` is given in `mode`, the generation of events of type `event_type` are enabled. If `JVMTI_DISABLE` is given, the generation of events of `event_type` are disabled.

If `event_thread` is `NULL`, the event is enabled or disabled globally; otherwise it is enabled or disabled for a particular thread. An event is generated for a particular thread if it is enabled either globally or at the thread level. No currently implemented event can be controlled at the thread level.

See section A.4 for details on events.

Initially no events are enabled at the thread level or global level.

Any needed capabilities must be possessed before calling this function.

PHASES

This function may only be called during the OnLoad or the live phase.

CAPABILITIES

Certain capabilities are necessary for some `event_types`:

- `can_generate_sample_execution_events` is necessary for `SampleExecution` events.
- `can_intercept_methods` is necessary for `SpecificMethodEntry` and `SpecificMethodExit` events.

PARAMETERS

`mode` Whether to `JVMTI_ENABLE` or `JVMTI_DISABLE` the event.

`event_type` The event to control.

`event_thread` The thread to control. If `NULL`, the event is controlled globally.

... For future expansion.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_INVALID_THREAD`: `event_thread` is non-`NULL` and is not a valid thread

`JVMTIPROF_ERROR_THREAD_NOT_ALIVE`: `event_thread` is non-`NULL` and is not live (has not been started or is now dead)

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: Thread level control was attempted on events that do not allow thread control.

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: Thread level control was attempted on events that do not allow thread control.

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `mode` is not a valid `jvmtiEventMode`.

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `event_type` is not a valid `jvmtiProfEvent`.

`JVMTIPROF_ERROR_MUST_POSSESS_CAPABILITY`: The capability necessary to enable the events is not possessed.

Set Event Callbacks

```
jvmtiProfError SetEventCallbacks(
            jvmtiProfEnv* jvmtiprof_env,
            const jvmtiProfEventCallbacks* callbacks,
            jint size_of_callbacks);
```

DESCRIPTION

Sets the functions called for each event. The callbacks are specified by supplying a replacement function table. The function table is copied, changes to the local copy of the table will have no effect. This is an atomic action, all callbacks are set at once. No events are sent before this function is called. When an entry is `NULL` or when the event is beyond `size_of_callbacks` no event is sent.

An event must be enabled and have a callback in order to be sent. The order in which this function and `SetEventNotificationMode` are called does not affect the result.

See section A.4 for details on events.

PHASES

This function may only be called during the OnLoad or the live phase.

CAPABILITIES

Required functionality.

PARAMETERS

`callbacks` The new event callbacks. If `NULL`, removes the existing callbacks.

`size_of_callbacks` Must be equal `sizeof(jvmtiProfEventCallbacks)`. Used for version compatibility.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTI_ERROR_ILLEGAL_ARGUMENT`: `size_of_callbacks` is less than `0` or not supported.

A.5 CAPABILITY

Get Potential Capabilities

```
jvmtiProfError GetPotentialCapabilities(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiProfCapabilities* capabilities_ptr);
```

DESCRIPTION

Returns the JVMTIPROF features that can potentially be possessed by this environment at this time.

The returned capabilities differ from the complete set of capabilities implemented by JVMTIPROF in two cases: another environment possesses capabilities that can only be possessed by one environment, or the current phase is live, and certain capabilities can only be added during the OnLoad phase. The `AddCapabilities` function may be used to set any or all of these capabilities. Currently possessed capabilities are included.

Typically this function is used in the OnLoad function. Typically, a limited set of capabilities can be added in the live phase. In this case, the set of potentially available capabilities will likely differ from the OnLoad phase set.

PHASES

This function may only be called during the OnLoad or the live phase.

CAPABILITIES

Required functionality.

PARAMETERS

`capabilities_ptr` Pointer through which the set of capabilities is returned.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: `capabilities_ptr` is `NULL`

Add Capabilities

```
jvmtiProfError AddCapabilities(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiProfCapabilities* capabilities_ptr);
```

DESCRIPTION

Set new capabilities by adding the capabilities whose values are set to `1` in `*capabilities_ptr`. All previous capabilities are retained. Typically this function is used in the OnLoad function. A limited set of capabilities can be added in the live phase.

PHASES
This function may only be called during the OnLoad or the live phase.

CAPABILITIES
Required functionality.

PARAMETERS

`capabilities_ptr` Pointer to the capabilities to be added.

ERROR CODES
This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NOT_AVAILABLE`: The desired capabilities are not even potentially available.

`JVMTIPROF_ERROR_NULL_POINTER`: `capabilities_ptr` is `NULL`

Relinquish Capabilities

```
jvmtiProfError RelinquishCapabilities(
            jvmtiProfEnv* jvmtiprof_env,
            const jvmtiProfCapabilities* capabilities_ptr);
```

DESCRIPTION
Relinquish the capabilities whose values are set to `1` in `*capabilities_ptr`. Some implementations may allow only one environment to have a capability. This function releases capabilities so that they may be used by other environments. All other capabilities are retained. The capability will no longer be present in `GetCapabilities`. Attempting to relinquish a capability that the environment does not possess is not an error.

PHASES
This function may only be called during the OnLoad or the live phase.

CAPABILITIES
Required functionality.

PARAMETERS

`capabilities_ptr` Pointer to the capabilities to relinquish.

ERROR CODES
This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: `capabilities_ptr` is `NULL`

Get Capabilities

```
jvmtiProfError GetCapabilities(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiProfCapabilities* capabilities_ptr);
```

DESCRIPTION

Returns the optional JVMTIPROF features which this environment currently possesses. Each possessed capability is indicated by a `1` in the corresponding field of the capabilities structure. An environment does not possess a capability unless it has been successfully added with `AddCapabilities`. An environment only loses possession of a capability if it has been relinquished with `RelinquishCapabilities`. Thus, this function returns the net result of the `AddCapabilities` and `RelinquishCapabilities` calls which have been made.

PHASES

This function may be called during any phase.

CAPABILITIES

Required functionality.

PARAMETERS

`capabilities_ptr` Pointer through which the currently possessed capabilities are returned.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: `capabilities_ptr` is `NULL`

A.6 GENERAL

Dispose Environment

```
jvmtiProfError DisposeEnvironment(jvmtiProfEnv* jvmtiprof_env);
```

DESCRIPTION

Shutdowns a JVMTIPROF connection created with `Create`. Disposes of any resources held by the environment. JVMTIPROF hooks on the associated JVMTI environment will be undone.

Any capabilities held by this environment are relinquished.

Events enabled by this environment will no longer be run. However, event handles currently running will continue to run. Caution must be exercised in the design of event handlers whose environment may be disposed of and thus become invalid during their execution.

This environment may not be used after this call.

PHASES
This function may be called during any phase.

CAPABILITIES
Required functionality.

ERROR CODES
This function returns a universal error code.

Get Jvmti Env

```
jvmtiProfError GetJvmtiEnv(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiEnv** jvmti_env_ptr);
```

DESCRIPTION
Obtains the JVMTI environment associated with this JVMTIPROF environment.

PHASES
This function may be called during any phase.

CAPABILITIES
Required functionality.

PARAMETERS

`jvmti_env_ptr` Pointer through which the associated JVMTI environment is returned.

ERROR CODES
This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: `jvmti_env_ptr` is `NULL`.

Get Version Number

```
jvmtiProfError GetVersionNumber(
            jvmtiProfEnv* jvmtiprof_env,
            jint* version_ptr);
```

DESCRIPTION

Return the JVMTIPROF version.

The version identifier follows the same convention as JVMTI, with a major, minor and micro version. These can be accessed through the `JVMTI_VERSION_MASK_MAJOR`, `JVMTI_VERSION_MASK_MINOR`, and `JVMTI_VERSION_MASK_MICRO` bitmasks applied to the returned value. The version identifier does not include an interface type, as such `JVMTI_VERSION_MASK_INTERFACE_TYPE` is not used.

See JVMTI's `GetVersionNumber` for details on the bitmasks.

PHASES

This function may be called during any phase.

CAPABILITIES

Required functionality.

PARAMETERS

`version_ptr` Pointer through which the JVMTIPROF version is returned.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_NULL_POINTER`: `version_ptr` is `NULL`.

Get Error Name

```
jvmtiProfError GetErrorName(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiProfError error,
            char** name_ptr);
```

DESCRIPTION

Return the symbolic name for an error code.

For example `GetErrorName(env, JVMTIPROF_ERROR_NONE, &err_name)` would return in `err_name` the string `"JVMTIPROF_ERROR_NONE"`.

PHASES

This function may be called during any phase.

CAPABILITIES

Required functionality.

PARAMETERS

`error` The error code.

`name_ptr` Pointer through which the symbolic name is returned. The pointer must be freed with the associated JVMTI `Deallocate` function.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `error` is not a valid error code.

`JVMTIPROF_ERROR_NULL_POINTER`: `name_ptr` is `NULL`.

Set Verbose Flag

```
jvmtiProfError SetVerboseFlag(
            jvmtiProfEnv* jvmtiprof_env,
            jvmtiProfVerboseFlag flag,
            jboolean value);
```

DESCRIPTION

Control verbose output. This is the output which typically is sent to `stderr`.

PHASES

This function may be called during any phase.

CAPABILITIES

Required functionality.

PARAMETERS

`flag` Which verbose flag to set.

`value` New value of the flag.

ERROR CODES

This function returns either a universal error code or one of the following errors:

`JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`: `flag` is not a valid flag.

A.7 EVENTS

Specific Method Entry

```
void JNICALL
SpecificMethodEntry(
    jvmtiProfEnv* jvmtiprof_env,
    jvmtiEnv* jvmti_env,
    JNIEnv* jni_env,
    jthread thread,
    jint hook_id)
```

DESCRIPTION

This event is generated when Java programming language methods specified in `SetMethodEventFlag` are called.

PHASES

This event is sent only during the live phase.

EVENT TYPE

`JVMTIPROF_EVENT_SPECIFIC_METHOD_ENTRY`

CAPABILITIES

`can_intercept_methods` is necessary for this event to be generated.

PARAMETERS

`jni_env` The JNI environment of the event (current) thread.

`thread` Thread that is entering the method.

`hook_id` Identifier of the method as returned by `SetMethodEventFlag`.

Specific Method Exit

```
void JNICALL
SpecificMethodExit(
    jvmtiProfEnv* jvmtiprof_env,
    jvmtiEnv* jvmti_env,
    JNIEnv* jni_env,
    jthread thread,
    jint hook_id)
```

DESCRIPTION

This event is generated when Java programming language methods specified in `SetMethodEventFlag` return to the caller.

PHASES

This event is sent only during the live phase.

EVENT TYPE

`JVMTIPROF_EVENT_SPECIFIC_METHOD_EXIT`

CAPABILITIES

`can_intercept_methods` is necessary for this event to be generated.

PARAMETERS

`jni_env` The JNI environment of the event (current) thread.

`thread` Thread that is leaving the method.

`hook_id` Identifier of the method as returned by `SetMethodEventFlag`.

Sample Execution

```
void JNICALL
SampleExecution(
    jvmtiProfEnv* jvmtiprof_env,
    jvmtiEnv* jvmti_env,
    JNIEnv* jni_env,
    jthread thread)
```

DESCRIPTION

This event is generated when the CPU-time specified in `SetExecutionSamplingInterval` has elapsed. The thread that caused the timer to expire is interrupted, and this callback is invoked.

The operations performed in this callback must be async-signal-safe. It is only limited to primitive operations and a subset of system calls. This implies that the callback must not use standard C library functions or JVMTIPROF functions. It is recommended to publish any necessary information to an async-signal-safe queue and consume it in another thread.

As an exception, the `GetStackTraceAsync` function can be used during this callback to obtain the stack trace of the interrupted thread.

PHASES

This event is sent only during the live phase.

EVENT TYPE

`JVMTIPROF_EVENT_SAMPLE_EXECUTION`

CAPABILITIES

`can_generate_sample_execution_events` is necessary for this event to be generated.

PARAMETERS

`jni_env` The JNI environment of the event (current) thread.

`thread` Thread that expired the timer.

A.8 ERROR CODES

The following error codes are equivalent to the ones presented in the JVMTI documentation:

- `JVMTIPROF_ERROR_NONE`
- `JVMTIPROF_ERROR_NULL_POINTER`
- `JVMTIPROF_ERROR_INVALID_ENVIRONMENT`
- `JVMTIPROF_ERROR_WRONG_PHASE`
- `JVMTIPROF_ERROR_INTERNAL`
- `JVMTIPROF_ERROR_ILLEGAL_ARGUMENT`
- `JVMTIPROF_ERROR_UNSUPPORTED_VERSION`
- `JVMTIPROF_ERROR_NOT_AVAILABLE`
- `JVMTIPROF_ERROR_MUST_POSSESS_CAPABILITY`
- `JVMTIPROF_ERROR_UNATTACHED_THREAD`