JOHANNES KEPLER UNIVERSITY LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

Fast Profiling in the HotSpot Java VM with Incremental Stack Tracing and Partial Safepoints

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements
for the academic degree

Bachelor of Science

in

COMPUTER SCIENCE

Submitted by

David Gnedt

At the

Institute for System Software

Advisor

o. Univ.-Prof. Dr. Dr.h.c. Hanspeter Mössenböck

Co-advisor

Dipl.-Ing. Peter Hofer

Linz, December 2014

Abstract

Runtime profiling is a frequently needed technique to analyse the performance of an application. An efficient and non-intrusive approach to runtime profiling is sampling, which interrupts the application periodically and analyses the current application state. To allow a developer to inspect how much time is spent in which part of an application and in which context, it is common practice to take entire stack traces at every interruption and combine them to a calling context tree.

This thesis proposes novel techniques for improving the efficiency of sampling stack traces in the HotSpot Java VM. *Incremental stack tracing* avoids duplicated stack tracing effort when taking many stack traces in a short period, as it is common with sampling profilers. Furthermore, this thesis describes interrupts with *partial safepoints*, which are a variation of safepoints that are highly optimized for sampling. Additionally, an experimental technique using Unix signals for interruption is presented. These new techniques are then compared to existing techniques in HotSpot, such as JVMTI and AsyncGetCallTrace, in terms of overhead, latency, stability and accuracy.

Kurzfassung

Runtime Profiling ist eine häufig benötigte Methode, um die Leistung einer Anwendung zu analysieren. Ein effizienter und nicht-invasiver Ansatz für Runtime Profiling ist Sampling, welches die Anwendung periodisch unterbricht und den aktuellen Anwendungszustand analysiert. Um einem Entwickler eine Beurteilung der Laufzeitverteilung in einer Anwendung zu ermöglichen, ist es üblich, ganze Stack Traces bei jeder Unterbrechung zu erzeugen und diese in einem Calling Context Tree zu vereinen.

Diese Bachelorarbeit stellt neuartige Methoden vor, um das Sampling von Stack Traces in der HotSpot Java VM effizienter zu gestalten. *Incremental Stack Tracing* verhindert mehrfachen Stack-Tracing-Aufwand, wenn viele Stack Traces in kurzer Zeit erzeugt werden, was eine übliche Vorgehensweise bei Sampling Profilern ist. Weiters werden Unterbrechungen mit *Partial Safepoints* beschrieben, welche eine Variation von Safepoints sind, die speziell für Sampling optimiert ist. Zusätzlich wird eine experimentelle Methode vorgestellt, welche Unix-Signale zur Unterbrechung verwendet. Anschließend werden diese neuen Methoden mit bestehenden Methoden wie JVMTI und AsyncGetCallTrace bezüglich der Performanceauswirkungen, Latenzzeit, Stabilität und Genauigkeit verglichen.

Contents

Chapter 1

Introduction

Profiling is the art of performance-related analysis of programs at runtime. It is an important task in software development and operation. When performance issues arise, profiling helps to reveal the parts of an application causing the issues. Runtime analysis is the most frequently used form of profiling. It analyses the frequency and duration of method calls. One approach to do runtime analysis is sampling, i. e., interrupting the program periodically and analysing its state.

This thesis focuses on runtime analysis in Oracle's HotSpot Java VM, which is one of the most widely used Java VMs. The HotSpot Java VM allows runtime analysis through sampling via its implementation of the Java Virtual Machine Tool Interface (JVMTI). HotSpot uses the safepoint mechanism to interrupt the program. However, safepoints are not optimized for sampling, as they were originally developed for garbage collection. They require to pause all application threads and therefore they cause high overhead, especially at shorter sampling intervals. Moreover, safepoints only allow program interruption at certain points, leading to distortion of the application profile. Furthermore, JVMTI has no mechanism to limit profiling to running threads, although running threads are the most interesting threads for profiling.

This thesis introduces new profiling techniques that improve over the usual JVMTI technique. *Incremental stack tracing* tries to reduce the overall stack tracing overhead by avoiding duplicated work on consecutive stack traces. *Partial safepoints* are a new sampling mechanism with improved latency and the ability to approximate sampling of running threads. Part of this thesis is also to implement and evaluate these techniques for the Linux x86-64 platform in the HotSpot Java VM. The profiling techniques and results described in this thesis were also published at the International Conference on Performance Engineering 2015 [4].

Outline

Chapter 2 describes basics of profiling with sampling, the HotSpot Java VM and shows an example for how profiling in the HotSpot Java VM works. Chapter 3 discusses incremental stack tracing. Chapter 4 describes mechanisms for interrupting a program for sampling, including the existing safepoint mechanism, the new partial safepoints technique, and Unix signals. Chapter 5 presents results from benchmark experiments with the new techniques.

Chapter 2

Background

This chapter discusses the most important facts about profiling and the HotSpot Java VM. An example shows how profiling can be done in a Java VM and finally the internals of HotSpot regarding JVMTI stack traces are described.

2.1 HotSpot Java VM

HotSpot[7] is a Java Virtual Machine (VM) by Oracle, which can execute Java bytecode. In Java, source code is not directly compiled into machine code, instead Java bytecode is used as intermediate representation. Figure 2.1 shows how Java applications are compiled. First the Java compiler (javac) transforms the source code into bytecode. A Java VM then executes the bytecode. The HotSpot Java VM can either interpret the bytecode or compile it to machine code using a just-in-time (JIT) compiler.

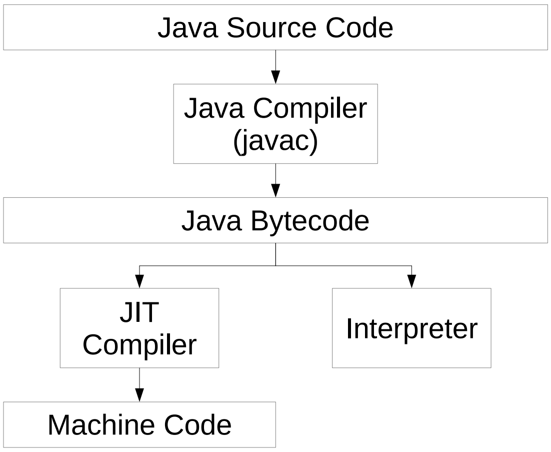


Figure 2.1: Compilation and execution of Java methods

Figure 2.2 shows the relevant parts of the architecture of the HotSpot Java VM. It consists of an interpreter and multiple JIT compilers to execute bytecode. JIT compiled methods are stored in the code cache. On startup the interpreter executes the application's bytecode. The VM monitors the interpreted methods and determines which methods are most used. A JIT compiler then compiles these hot methods while the application is running. Usually, the server compiler (C2, opto) is used on desktop and server systems. It provides better optimizations compared to the client compiler (C1), but is a lot slower. Therefore, tiered compilation was introduced to reduce the application's startup time. The VM first compiles methods quickly using the client compiler and monitors them similar to interpreted methods. When they are still hot, the VM compiles a fully optimized version using the server compiler.

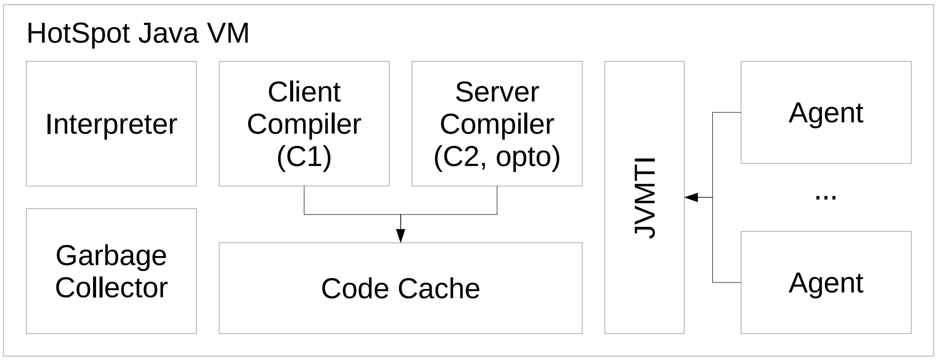


Figure 2.2: Components of the HotSpot Java VM

The VM uses a garbage collector for managing the application's heap memory. Multiple interchangeable garbage collector implementations are available, which can be selected. Furthermore, the VM can be extended using agents. These agents are dynamically-linked native libraries that can be loaded at runtime. They run inside the VM process and can access the VM through the Java Virtual Machine Tool Interface (JVMTI) API [8]. JVMTI allows an agent to start its own threads and register for VM events. When a registered event occurs the VM notifies the agent through callback functions. JVMTI is a standard that is also supported by various other Java VMs.

2.2 Profiling

Profiling analyses programs at runtime to gather performance-related information like method call counts and execution time. Basically profiling data can be obtained with two different methods: instrumentation and sampling.

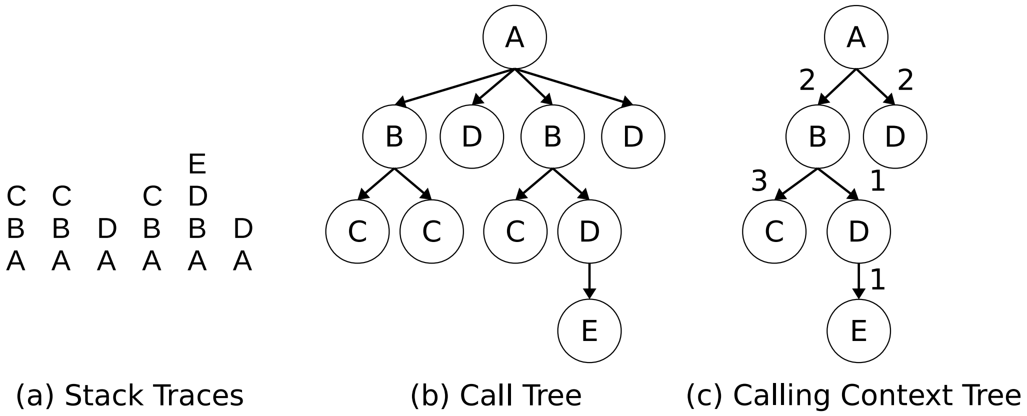
With instrumentation, the program code is modified to enable the collection of profiling data, for example by adding code at method calls and returns. These modifications are not necessarily at source code level, but can be applied at every level from source code to machine code. In the case of Java, the most useful level is at bytecode, because the source code is not always available and bytecode can be instrumented easily during class loading. By modifying the program code itself, the profile of the analysed program can differ extremely from the original program. This is mainly due to changes in possible compiler optimizations. For example, when a simple getter is instrumented, the compiler cannot reduce it to a simple memory read any more and has to treat it like a normal function.

In contrast to instrumentation, which can see each and every method call, sampling is a statistical approach, where the program is interrupted and the state is analysed in certain intervals. Therefore, it cannot determine exact method call counts, but only a ratio of how often a certain method was seen executing. Generally speaking, the sampling overhead is usually lower than the instrumentation overhead, and most of the time exact call counts are not necessary.

Calling Context

Often it is not enough to know only the top executing methods, because the problem is in a caller method and not in the hot executing method itself. Therefore, a complete calling context needs to be recorded for each call or sample, depending on which method is used. A calling context is a stack trace, which consists of the top executing method and all caller methods down to the program's main method. Figure 2.3 (a) shows multiple stack traces, where each column denotes a single stack trace with the executing method at the top. The first stack trace shows that the program's main method was A, which called method B, which then further called C.

The stack traces can also be visualized in a call tree like shown in Figure 2.3 (b). Each invocation is a node and the root of the call tree is the program's main method. The childs represent methods that were called. If execution times of individual calls are measured, they can be indicated as edge weights in the tree. With sampling



(a) Stack Traces (b) Call Tree (c) Calling Context Tree

Figure 2.3: Stack traces, call tree and calling context tree

profilers it is difficult to build call trees, because individual invocations cannot be easily distinguished.

Furthermore, Figure 2.3 (c) shows a calling context tree (CCT, [1]). It can be derived from a call tree by merging calls of the same method with the same parent and retaining a count or sum of execution times. With instrumentation the call counts and execution times are absolute. When sampling is used with a uniform sampling interval the counts should be normalized to the total sample count and interpreted as relative execution times. The resulting CCT is usually considerably more compact than a call tree and therefore also more helpful in practice.

2.3 Profiling in the HotSpot Java VM

In the HotSpot Java VM, profiling is done through agents. The VM loads a profiling agent, which obtains the profiling data through JVMTI. For sampling profilers, JVMTI provides some convenient functions to retrieve stack traces:

`GetStackTrace` allows obtaining the stack trace of a single thread,

`GetAllStackTraces` allows obtaining stack traces from all threads, and

`GetThreadListStackTraces` allows obtaining stack traces for multiple threads [8].

Listing 2.1 shows an example of a profiling agent using sampling. For simplicity it only obtains the maximum number of frames on the stack, also known as stack depth. Nevertheless, it can be extended to build a calling context tree and therefore intentionally uses `GetAllStackTraces` instead of the more special function `GetFrameCount`. To further simplify the example, it does no error checking and thread synchronisation.

```c
#include <string.h>
#include <unistd.h>
#include <pthread.h>

#include "jvmti.h"
#include "agent.h"

#define SAMPLE_PRIORITY 10
#define SAMPLE_INTERVAL 10000
#define MAX_FRAMES      1024

jvmtiEnv *jvmti;
volatile bool workerShouldRun = true;

JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM* vm, char* options, void* reserved) {
    printf("[simple-agent]␣OnLoad\n");

    // get environment (1)
    vm->GetEnv((void**) &jvmti, JVMTI_VERSION_1_2);

    // register callbacks (2)
    jvmtiEventCallbacks callbacks;
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.VMInit = OnVMInit;
    callbacks.VMDeath = OnVMDeath;
    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));

    // enable specific events (2)
    jvmtiEvent events[] = {JVMTI_EVENT_VM_INIT, JVMTI_EVENT_VM_DEATH};
    for (int i = 0; i < sizeof(events) / sizeof(events[0]); i++) {
        jvmti->SetEventNotificationMode(JVMTI_ENABLE, events[i], NULL);
    }

    return 0;
}

void JNICALL OnVMInit(jvmtiEnv* jvmti, JNIEnv* jni, jthread thread) {
    printf("[simple-agent]␣OnVMInit\n");

    jthread samplingThread = AllocThread(jni);
    // start sampling thread (4)
    jvmti->RunAgentThread(samplingThread, SamplingWorker, NULL, SAMPLE_PRIORITY);
}

jthread AllocThread(JNIEnv* jni) {
    // create thread object (3)
    jclass threadClass = jni->FindClass("java/lang/Thread");
    jmethodID ctor = jni->GetMethodID(threadClass, "<init>", "()V");
    return jni->NewObject(threadClass, ctor, "Agent␣Thread");
}

void SamplingWorker(jvmtiEnv* jvmti, JNIEnv* jni, void* arg) {
    long samples = 0;
    int maxStackDepth = -1;

    printf("[simple-agent]␣sampleInterval:␣%d␣us\n", SAMPLE_INTERVAL);

    // sampling loop (5)
    while (workerShouldRun) {
        jint curMaxStackDepth = GetMaxStackDepth();
        // find overall max stack depth (8)
        if (curMaxStackDepth > maxStackDepth) {
            maxStackDepth = curMaxStackDepth;
        }
        samples++;
        // (9)
        usleep(SAMPLE_INTERVAL);
    }

    // (11)
    printf("[simple-agent]␣samples:␣%ld\n", samples);
    printf("[simple-agent]␣maxStackDepth:␣%d\n", maxStackDepth);
}

jint GetMaxStackDepth() {
    jvmtiStackInfo *stackInfo;
    jint threadCount;
    int maxStackDepth;

    // capture stack traces (6)
    jvmti->GetAllStackTraces(MAX_FRAMES, &stackInfo, &threadCount);

    // find current max frame count(7)
    for (int i = 0; i < threadCount; i++) {
        if (stackInfo[i].frame_count > maxStackDepth) {
            maxStackDepth = stackInfo[i].frame_count;
        }
    }

    jvmti->Deallocate((unsigned char*) stackInfo);

    return maxStackDepth;
}

void JNICALL OnVMDeath(jvmtiEnv* jvmti, JNIEnv* jni) {
    printf("[simple-agent]␣OnVMDeath\n");

    // stop sampling thread (10)
    workerShouldRun = false;
}

JNIEXPORT void JNICALL Agent_OnUnload(JavaVM* vm) {
    printf("[simple-agent]␣OnUnload\n");

    // dispose environment (12)
    jvmti->DisposeEnvironment();
}
```

Listing 2.1: Simple JVMTI Sampling Agent

Every agent has a function called `Agent_OnLoad`. It is executed before the VM is initialized and allows registering callbacks and capabilities. Usually first the JVMTI environment is requested (1), which is then used to interact with the VM. In the given example the callbacks `VMInit` and `VMDeath` are registered (2). `VMInit` is issued when the VM initialization is finished and `VMDeath` when the VM shuts down. Therefore, in the example the function `OnVMInit` is called once the VM initialization is finished. It first creates a new thread object, which requires a reference to the class and its constructor (3) and then starts an agent thread for sampling (4). The sampling thread then periodically retrieves the stack traces of all threads (5). As previously described, it uses the JVMTI function `GetAllStackTraces` to accomplish this task (6). For each stack trace the stack depth is checked and the maximum is kept (7, 8). In this example a sampling interval of 10 ms is used (9). It should be kept in mind that the sampling itself also takes some time. Therefore, usually the sampling delay needs to be compensated by subtracting it from the sampling interval. For the sake of simplicity the sampling delay is not compensated in this example. When the application exits, the VM issues the `VMDeath` event and therefore calls the `OnVMDeath` function of the simple agent. It stops the sampling thread (10). Before stopping the sampling thread prints the maximum frame count (11). Finally, the VM calls the `Agent_OnUnload` function, which is used for some necessary cleanup tasks (12).

Listing 2.2 shows a test of the simple agent with the sunflow benchmark from the DaCapo benchmark suite [2]. The command line is denoted as $ and the user input is bold. The `-agentpath` parameter specifies the agent. The agent reveals a maximum stack depth of 51 for this benchmark.

```
$ java -agentpath:libsimple-agent.so -jar scala-benchmark-suite.jar -n 4 sunflow
[simple-agent] OnLoad
[simple-agent] OnVMInit
[simple-agent] sampleInterval: 10000 us
Using scaled threading model. 8 processors detected, 8 threads used to drive the ↩
    ↪workload, in a possible range of [1,256]
===== DaCapo 0.1.0-SNAPSHOT sunflow starting warmup 1 =====
===== DaCapo 0.1.0-SNAPSHOT sunflow completed warmup 1 in 2513 msec =====
===== DaCapo 0.1.0-SNAPSHOT sunflow starting warmup 2 =====
===== DaCapo 0.1.0-SNAPSHOT sunflow completed warmup 2 in 1987 msec =====
===== DaCapo 0.1.0-SNAPSHOT sunflow starting warmup 3 =====
===== DaCapo 0.1.0-SNAPSHOT sunflow completed warmup 3 in 1998 msec =====
===== DaCapo 0.1.0-SNAPSHOT sunflow starting =====
===== DaCapo 0.1.0-SNAPSHOT sunflow PASSED in 1917 msec =====
[simple-agent] OnVMDeath
[simple-agent] OnUnload
[simple-agent] samples: 746
[simple-agent] maxStackDepth: 51
```

Listing 2.2: Simple JVMTI Sampling Agent Output

Safepoint Mechanism

An interesting aspect is how the `Get*StackTraces` functions are implemented in HotSpot. First of all these functions need to get stack traces of other threads, which are different from the currently running (agent) thread. As all threads run in a single process the current thread can access the stack memory of those threads, but when they keep running, the stack changes permanently and it cannot be guaranteed that a valid stack trace is captured. Therefore, HotSpot needs to suspend these threads, capture their stack traces and resume them again.

For this purpose HotSpot uses the safepoint mechanism. It was originally introduced for garbage collection, to have a point where the heap is in a consistent state and only the GC modifies it (stop-the-world). This consistent state is reached by suspending all application threads. The safepoint mechanism is implemented generically, such that arbitrary operations can be executed in a safepoint. It is managed by the VM thread, which can be seen as the VM's main thread. The VM thread should not be mistaken for the application's main thread, they are different threads. The VM thread has a work queue in which other threads can put operations. An operation can specify if it needs to be executed at a safepoint. The `Get*StackTraces` functions also put their own operations into this work queue and wait until they are finished.

HotSpot has a global safepoint state, which determines if the VM is currently at a safepoint. When HotSpot is at a safepoint, all application threads are suspended. The safepoint mechanism works by having each thread periodically poll if it should suspend. In JIT compiled code this polling is implemented efficiently by reading from a special memory page, which is called the polling page. Usually it is readable and the application threads only do some additional memory reads. Caching helps to dramatically reduce performance penalty of this safepoint check. Once the VM wants to enter a safepoint, for example because a stack trace was requested, the polling page is set protected. Subsequent reads on the page result in a segmentation fault, which is caught using a signal handler. The signal handler detects that the segmentation fault happened while reading the polling page and the VM wants to enter a safepoint, therefore the thread blocks itself. The blocking and unblocking of threads is realized using locks. The VM waits until all application threads are blocked. Application threads currently in native code e. g., doing I/O, are an exception from this rule. They can continue execution because they are not allowed to modify the Java stack and heap. When they return from native code while the VM is still at a safepoint, the threads immediately block. When all application threads are in a blocked or another safe state, the VM executes the operation, which for example captures stack traces of all threads. After the operation finishes the VM exits the safepoint state by setting the polling page readable again and unblocking all threads. The application continues its usual program flow.

Chapter 3

Incremental Stack Tracing

This chapter describes the idea of incremental stack tracing and walks through the details of the algorithm, data structures, interface and implementation.

Idea

When a sampling profiler uses stack tracing, it takes many stack traces in a short time period. A stack frame can only change while it is on top of the stack and instructions of the owning method are executed. Therefore, most of the time only few stack frames change between two successive stack traces. Nevertheless, the HotSpot Java VM walks all stack frames for each stack trace.

Figure 3.1 shows an example of profiling with full stack traces. Method `a` is called, which calls method `b` that finally calls method `c`. While in method `c` the profiler takes a

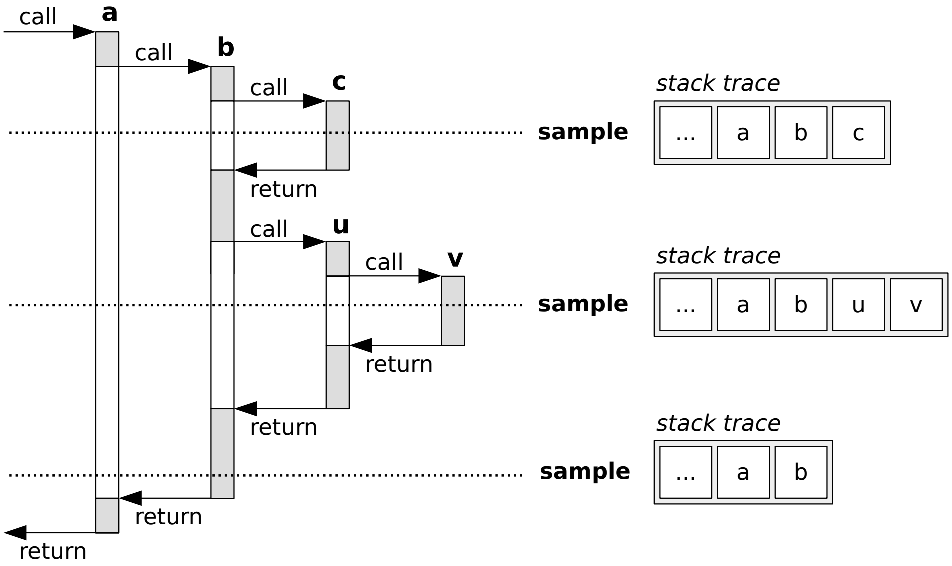


Figure 3.1: Sampling with full stack traces

sample with stack trace `a`, `b`, `c`. The execution continues, `c` returns and `b` calls method `u` that in turn calls `v`. Again the profiler takes a sample with stack trace `a`, `b`, `u`, `v`. After returning from `v` and `u` the profiler takes a third sample. The stack frames of method `a` and all its callers have not changed, but were walked three times in this example.

Incremental stack tracing avoids walking unchanged stack frames multiple times. It splits up the whole stack tracing work, so that it can perform it individually for each stack frame in a lazy fashion just before a stack frame can change. As a result, when a profiling agent makes a request, the stack trace is not immediately available, but can be retrieved at a later point. Nevertheless, the request itself is cheaper than a full stack trace, because it only needs to save the top stack frame. Incremental stack tracing is based on a similar method for implementing continuations in a Java VM [10].

3.1 Data Structures

Incremental stack tracing needs to keep track of the requested stack traces so they can be retrieved later. The algorithm employs a tree-like structure for this purpose as shown in Figure 3.2. Stack trace objects act as anchor points for individual stack traces. Each stack trace object has an identifier and references the next stack trace object, forming a linked list of all requested stack traces. Furthermore, a stack trace object references the frame object of the method that was on top while the sample was taken as `top frame`. Frame objects hold informations about stack frames including the frame's address on stack and the method they belong to. Each frame object references another frame object further down in the stack trace as `parent`. Frame objects can be either filled or skeleton frame objects. If the parent of a frame object is filled, then it

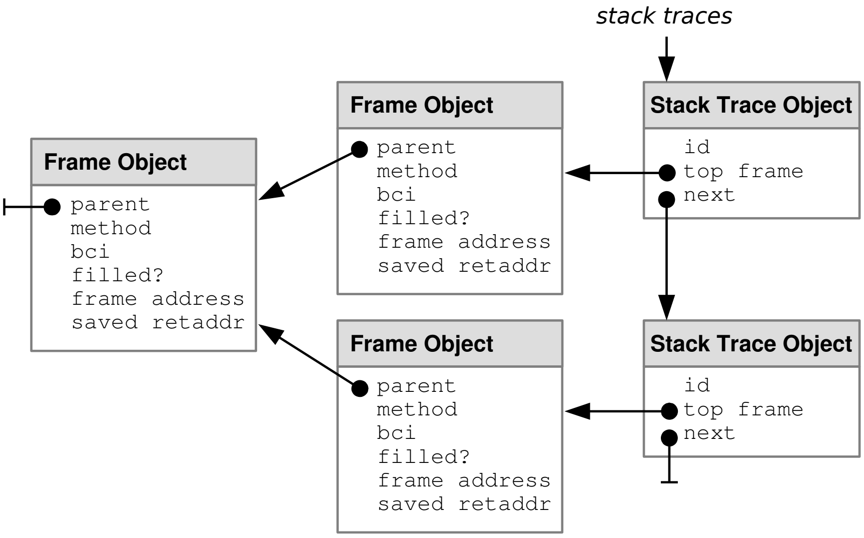


Figure 3.2: Data structures for incremental stack tracing

is always the caller of the frame object. Otherwise, some frame objects may still be missing. The frame object also stores the executing bytecode index (bci) that points to the position in the method. This has major implications on the semantics of frame objects. Strictly speaking a frame object corresponds to a stack frame at a certain time. When a stack frame changes, there may be multiple frame objects corresponding to the same stack frame at different times. These frame objects would then belong to the same method, but have different bytecode index positions. Furthermore, a frame object also stores the saved return address (saved retaddr), which will be described in the next section.

When using incremental stack tracing with the example shown in Figure 3.1, the resulting data structure looks like Figure 3.3. Rectangles indicate frame objects and circles indicate stack trace objects. In the example, the profiler took three samples with identifiers 1 through 3. Each stack trace shares the frame objects of method `a` and below. Method `b` is not shared because its execution continues between the samples. Therefore, frame object `b`, `b′` and `b″` each have different bytecode indices.

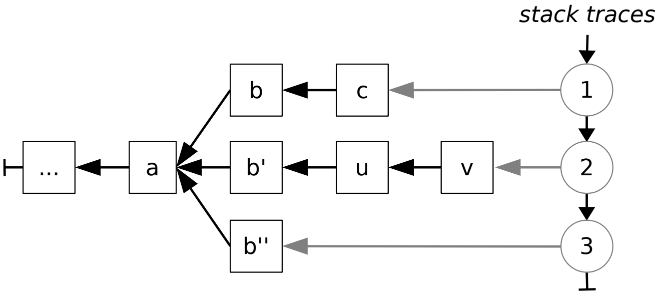


Figure 3.3: Tree-like data structure with shared frame objects

3.2 Algorithm

A request interrupts the normal program flow and triggers the execution of the incremental stack tracing algorithm. The algorithm then saves the top stack frame and intercepts the method return of the active invocations to incrementally save the following stack frames. Therefore, the algorithm needs to change the control flow of the program. It does not change the control flow through modification of the program code, but through modifying the return address of methods stored on the stack. This has the advantage that the modification is thread-local and therefore does not affect any other thread running the same code. When the profiler requests a stack trace, we create a new stack trace object and frame object. The frame object records the state of the current top frame on stack, so we call it top frame object (TFO). We decode the top stack frame and fill the method identifier and bytecode index (bci) of the TFO. The frame address and saved return address fields are unused in a filled frame object. We create a second frame object for the caller and make it the parent of the TFO. We also define it as the current skeleton object (CSO). It saves the original return address of the top stack frame and the caller's frame address. A skeleton frame object does not use the method and bci fields. For intercepting the next method return, we patch the return address of the top stack frame to point to a piece of trampoline code that we generate during the VM startup. Figure 3.4 (a) shows the data structure in the current state. When the method returns we decode the stack frame, fill the method identifier and bytecode index of the CSO and mark it filled. Then we create a new skeleton frame object as parent, save the original return address and the caller's frame address and make it the new CSO. For intercepting the next method return, we again patch the return address to point to our trampoline code. Figure 3.4 (b) shows the data structure after this.

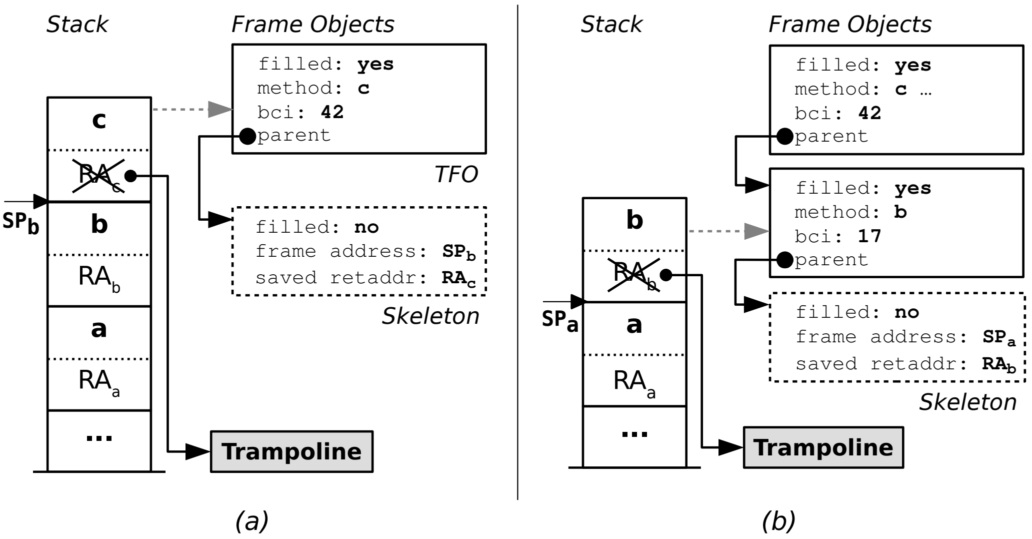


Figure 3.4: Incremental stack tracing data when (a) requesting a new stack trace and (b) intercepting a method return

The incremental stack tracing algorithm is therefore split into two parts, requesting a new stack trace and intercepting a method return. When requesting a new stack trace, we perform the following steps:

1. We decode the top stack frame, create a new TFO, and fill in the method identifier and bci from the top stack frame.

2. If the CSO is not set or the frame address of the CSO does not match the top stack frame's caller, then we create a new skeleton object and fill in the original return address and the caller's frame address. The CSO becomes the parent of the new skeleton object. We patch the return address to point to our trampoline code and make the new skeleton object the CSO.

3. The CSO becomes the parent of the TFO.

Intercepting a method return works as follows:

1. The CSO's frame address always matches the top stack frame. We decode the top stack frame and fill in the method identifier and bci in the CSO.

2. If the CSO's parent is not set or the frame address of the CSO's parent does not match the top stack frame's caller, then we create a new skeleton object and fill in the original return address and the caller's frame address. Next, we insert the new skeleton object between the CSO and the CSO's parent, so it becomes the parent of the CSO. Then we patch the return address to our trampoline code and make the new skeleton object the CSO.

Figure 3.5 shows the example from Figure 3.1 with incremental stack tracing. At the beginning the data structure of the algorithm is empty and therefore also no CSO is set.

(1) At the first sample request we create a new TFO and fill it with information from the top stack frame of method `c`. We also create a new skeleton object and save the original return address. We patch the return address, make the CSO the parent of the TFO and define the new skeleton object as the new CSO.

(2) When method `c` returns, control flow continues in our trampoline. The trampoline executes our intercept operation, which fills the CSO with the corresponding information from the stack frame of method `b` and creates a new skeleton object with the original return address of method `b`. The new skeleton object becomes the parent of the CSO. We then patch the return address of method `b` and make the new skeleton object the CSO. The trampoline then continues execution at the saved original return address.

(3) At the second sample request we create a new TFO with information from method `v`. We see that the caller's frame address does not match the CSO's frame address, therefore we create a new skeleton object with the original return address of method `v` and frame address of `u`. The skeleton object becomes the parent of the TFO and the CSO becomes the parent of the new skeleton object. Finally, we patch the return address of `v` and make the new skeleton object the CSO.

(4) When method `v` returns, we fill the CSO with information from method `u`. The CSO's parent frame object `a` has a different frame address than the stack frame's caller, therefore we introduce a new skeleton object `b'` and store the original return address of `u` and the frame address of `b`. We insert the new skeleton object `b'` between `u` and `a`. Further we patch the return address and make the new skeleton object `b'` the CSO.

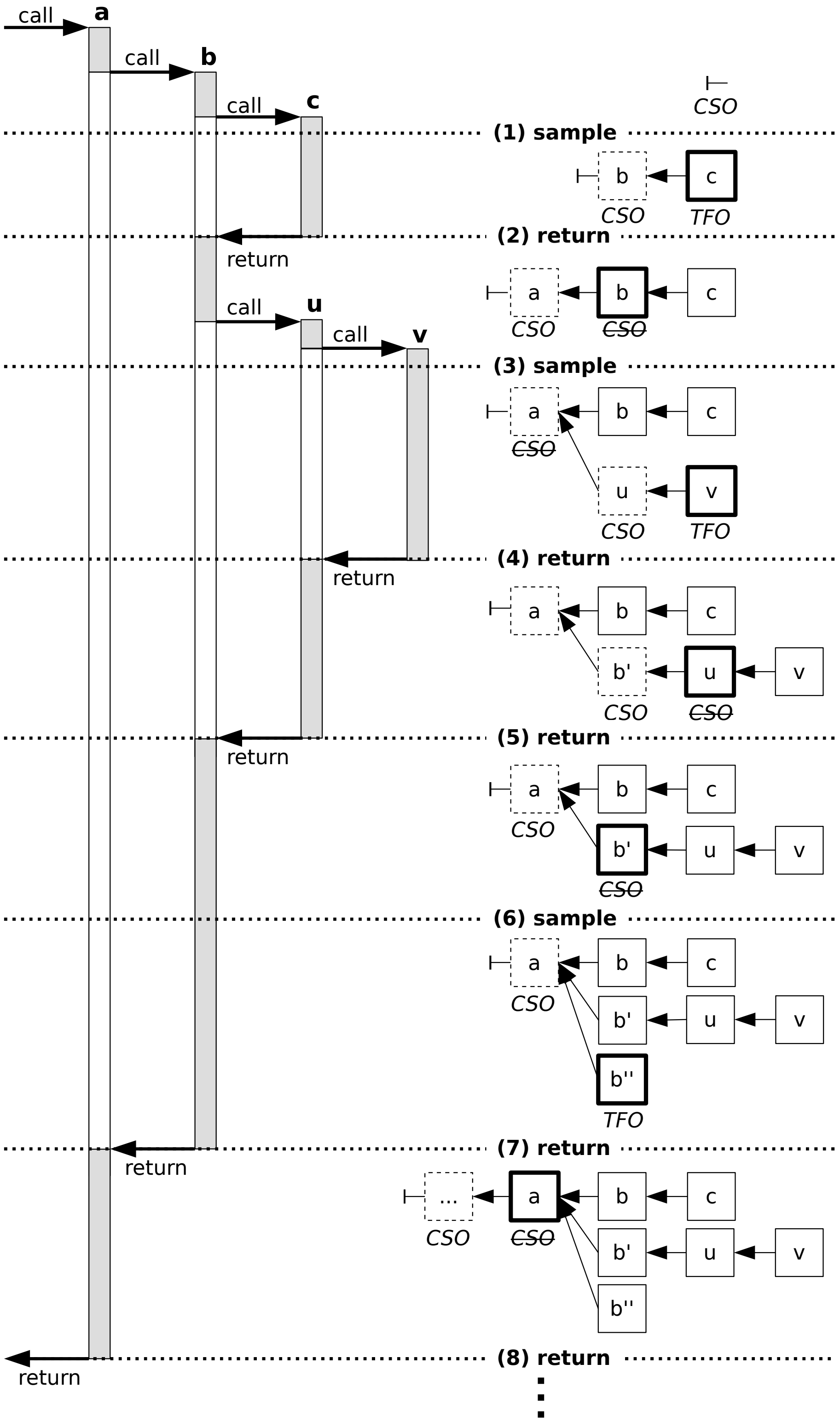


Figure 3.5: Sampling with incremental stack traces

(5) When method `u` returns, we fill the CSO from the stack frame of method `b`. The CSO's parent frame address equals the caller of method `b`, therefore we only make the CSO's parent the CSO and the return address is already patched.

(6) At the third sample request we create a new TFO with the CSO as parent. The frame address of the CSO matches the caller's frame address, therefore no new skeleton object or return address patching is necessary.

(7) When method `b` returns we fill the CSO, create a new skeleton object, make it the parent of the CSO, patch the return address and make the new skeleton object the CSO.

(8) The method return interception continues until it reaches the entry method (e. g., the program's main method).

3.3 Implementation

When using the incremental stack tracing algorithm with the HotSpot Java VM, multiple challenges arise. The algorithm depends on many internal details not exposed through the JVMTI interface. Therefore, implementing it as an agent is very difficult. A better solution is to extend the VM instead.

Interface

Although incremental stack tracing is implemented directly inside the VM, profiling data should still be collected in a profiling agent. The asynchronous nature of the incremental stack tracing algorithm implies that the usual `Get*StackTraces` interface provided by JVMTI is insufficient. It is required to split this operation into a request and retrieve part. JVMTI provides an extension mechanism that allows a VM to export non-standardized methods and an agent to query for such extension methods. The following two extension methods allow an agent to use incremental stack tracing:

`RequestStackTraces` allows requesting a stack trace for a provided set of threads. The agent can specify an identifier for the resulting stack traces.

`RetrieveStackTraces` allows retrieving internally stored stack traces for the given threads. It forces the algorithm to complete the internal stack trace representation down to the program's main method. The returned result is a tree-like structure similar to the internal representation, which contains all previously requested stack traces on the given threads. The agent can match the different stack traces to its requests using the identifier provided during the request. After retrieving the stack traces, the algorithm clears them from the internal data structure.

Typically an agent would periodically request stack traces with high frequency through `RequestStackTraces` and then retrieve the stack traces with much lower frequency through `RetrieveStackTraces`. The agent has to retrieve the stack traces before a thread ends, otherwise the VM releases them when freeing the thread's resources. This can be accomplished by registering for the JVMTI `ThreadEnd` event.

Requesting a stack trace

The agent can request stack traces from multiple threads with a single call to `RequestStackTraces`. Section 3.2 only described what happens in a single thread. To run incremental stack tracing on multiple threads, we keep the state thread-local, i. e., each thread has a separate CSO and list of stack traces. When the sampling mechanism ensures that it does not interrupt a potentially running return intercept, no synchronization is necessary and the algorithm can run as-is on an arbitrary number of threads.

When an agent requests a stack trace, the extension method instantiates a custom VM operation, submits it to the VM thread and waits for its completion. The VM operation requires a safepoint, therefore the VM thread first triggers a global safepoint before executing the VM operation (see section 2.3). The execution of all threads is stopped, therefore the VM thread can safely decode stack frames and patch return addresses as previously described in section 3.2. When the VM thread finishes, it exits the safepoint state and the agent and the application can continue execution.

Return Interception

The incremental stack tracing algorithm needs the ability to intercept the return of certain method invocations. It is important to differentiate between the return of a single method invocation and returns of a method in general. As threads usually share code, multiple active method invocations possibly exist at the same time. Therefore, it is hard to only intercept a single invocation with code modifications. An alternative to modifying the code is to modify the return address on stack. As every method invocation has its own stack frame, incremental stack tracing can easily intercept the return of a single method invocation.

The algorithm description in section 3.2 already assumes that method returns are intercepted using return address patching. The algorithm saves the original return address from the stack and replaces it with the address to a piece of trampoline code. The whole VM only has one instance of this trampoline code, which is created during VM startup. Therefore, all threads use the same piece of trampoline code.

The trampoline code saves the current processor state, calls into the incremental stack tracing algorithm, then restores the processor state and continues execution at the original return address. This makes the return interception transparent to the executing Java code. Although it makes the assumption that the return address is only used for subroutine handling, Java code cannot directly access the return address, therefore this assumption generally holds. Nevertheless, the VM uses the return address itself for various purposes, which are explained later in this chapter.

The optimizing compiler (C2) makes heavy use of inlining. As inlined methods have no separate stack frame and no return address on stack, return address patching cannot intercept them. This must be considered by extending the incremental stack tracing algorithm. The JIT compilers store debugging metadata in the code cache, which also contain informations about inlining at specific method positions. Usually incremental stack tracing only fills a single frame object with each request or intercept operation. When it detects inlining, it has to create multiple frame objects and fill them using the metadata information. The algorithm has to insert them between the filled frame object and the CSO. Therefore, the data structure looks similar like when inlining is disabled. This change does not affect the return address patching.

Challenges of return address patching

The HotSpot Java VM mixes different frame layouts on the stack. The JIT compilers use a frame type called compiled frame, which is very similar to an optimized native compiled frame (e. g., the frame of a C program compiled with `gcc -O2`). It does not contain a frame pointer or any other frame linkage information. The JIT compilers store some additional information in the code cache, which allow walking the stack based on the stack pointer register and the return address information found on stack. Therefore, when incremental stack tracing patches the return address of a compiled frame, the VM cannot walk it anymore. Various VM services, including the garbage collector, require a walkable stack. Therefore, some additional modifications are required to allow the VM to walk patched compiled frames again. When the VM sees a patched return address, it then retrieves the original return address from the incremental stack tracing data structure.

The interpreter uses a frame type called interpreted frame that is very different from a compiled frame. It provides frame linkage information, but the return address is stored at a different position. Therefore, the patching procedure needs to consider which frame type it patches. The trampoline code itself can be the same as for compiled frames because when the return happens, the interpreter already considers that the VM may execute non-interpreted code next.

Exception handling is another challenge. When a method throws an exception and does not catch it, the method does not return through its usual program flow. The VM unwinds the stack while it searches for a matching exception handler in the callers. This means it removes frames from the stack without regular returns. If multiple successive methods do not catch the exception, then the VM also unwinds multiple stack frames. When the VM finds a matching exception handler, it passes control flow to this exception handler and control flow never returns to the unwound methods. Therefore, when the VM unwinds a patched stack frame, incremental stack tracing misses the return and its data structure becomes broken, as the CSO does not match the next intercepted return anymore. The VM's exception handling uses the return address of methods to search for exception handlers. As a solution, when the VM sees a patched return address, it executes a special exception handler with a similar purpose as the trampoline code. It saves the current processor state, calls into the incremental stack tracing algorithm, restores the processor state, and continues the search for an exception handler using the original return address. This continues for all stack frames until an exception handler is found. Hence, incremental stack tracing does not miss any stack frame unwinds and keeps its data structure up-to-date like when a method returns normally.

The optimizing compiler (C2) uses adaptive optimization. This means it skips certain uncommon parts of a method to reduce the code size and increase the cache locality. These uncommon parts are determined by profiling the methods when they run in the interpreter or less optimized code from the C1 compiler. Although code compiled with assumptions is faster, an assumption may not hold throughout the entire program runtime [5]. For example, a method is typically not called with a null value as argument, but it may be passed null at some point. When assumptions fail, so-called deoptimization must happen. Deoptimization is the process of transitioning from more optimized code to less optimized code or to the interpreter. When switching from a C2-compiled method to the interpreter, it is necessary to convert from a compiled frame to an interpreted frame. Therefore, incremental stack tracing needs to ensure that if it patched the compiled frame, the resulting interpreted frame stays patched. Moreover, deoptimization also makes use of return address patching. Therefore, deoptimization might need to patch a return address that was already patched by incremental stack tracing, or the other way around. Incremental stack tracing should consider the original frame before deoptimization. So when deoptimization wants to patch a return address already patched by incremental stack tracing, it has to modify the original return address in the incremental stack tracing data structure instead of the return address on stack. In the other direction, when incremental stack tracing patches a deoptimization patched return address, no changes are necessary.

The VM also has a mechanism called on-stack replacement (OSR), which can be seen as ”reverse deoptimization“. Usually only new method invocations use more optimized code after a method is compiled and running invocations stay in less optimized code. OSR enables transition from less optimized code to a more optimized variant. It cannot use existing more optimized code, but needs to compile a special OSR variant. Therefore, it is only used in some cases, especially with long-running methods containing hot loops. When OSR replaces an interpreted frame with a compiled frame, the frame address can change due to the different frame layouts. Incremental stack tracing needs to keep track of these changes.

Chapter 4

Sampling Mechanisms

This chapter discusses methods to interrupt a running program for sampling. It first describes the common sampling method using safepoints, followed by a new variant called partial safepoints and finally sampling using Unix signals.

The sampling mechanism is one of the most important things in the sampling process. It is independent from what is sampled, but poses certain limits on the performance and accuracy that can be reached. The safepoint mechanism is rather heavyweight, because it requires synchronization of all threads, and therefore typically has a high overhead. Moreover, the safepoint mechanism cannot interrupt a program at an arbitrary position, but only at safe positions, where it checks for the safepoint state. Therefore, the possible different program positions seen in a safepoint are limited, which affects the accuracy of the resulting profile.

Safepoints

Section 2.3 already described how the safepoint mechanism works in detail. Figure 4.1 (a) shows the timeline for taking a sample from an example application. The example application has four application threads T_1, T_2, T_3, and T_4 out of which T_1, T_2, and T_3 are runnable. The application thread T_4 is doing I/O, i. e., it is in native code (marked with a dotted line). T_{VM} is the VM thread. The agent requests samples of the three application threads T_1, T_3, and T_4 by enqueuing a VM operation via `GetThreadListStackTraces`. When the VM thread dequeues this VM operation, which requires a safepoint, it sets the safepoint state to *safepointing*, signals a safepoint by setting the polling page protected and waits until all runnable threads reached a safepoint check. T_1 first comes to a safepoint check (denoted as square) and parks itself (dashed line), followed by T_3. Although the agent does not want to sample T_2, the safepoint mechanism waits until T_2 reaches a safepoint check and parks itself. All threads are now parked or in native code, hence the VM sets the safepoint state to *in safepoint*. The VM thread can now execute the VM operation, which sequentially takes samples from the requested threads. It is safe for the VM thread to sample threads that are currently in native code, because only the Java part of the stack is sampled, which the native code is not allowed to change. Furthermore, the VM ensures that a thread cannot change its state without doing a safepoint check. While taking the samples the thread T_4 returns from its I/O request in native code, the VM does a safepoint check, and the thread parks itself. When the VM operation finishes sampling, the VM thread exits the safepoint by resetting the safepoint state, setting the polling page unprotected and resuming all application threads. Figure 4.1 (b) and (c) is discussed later in this chapter.

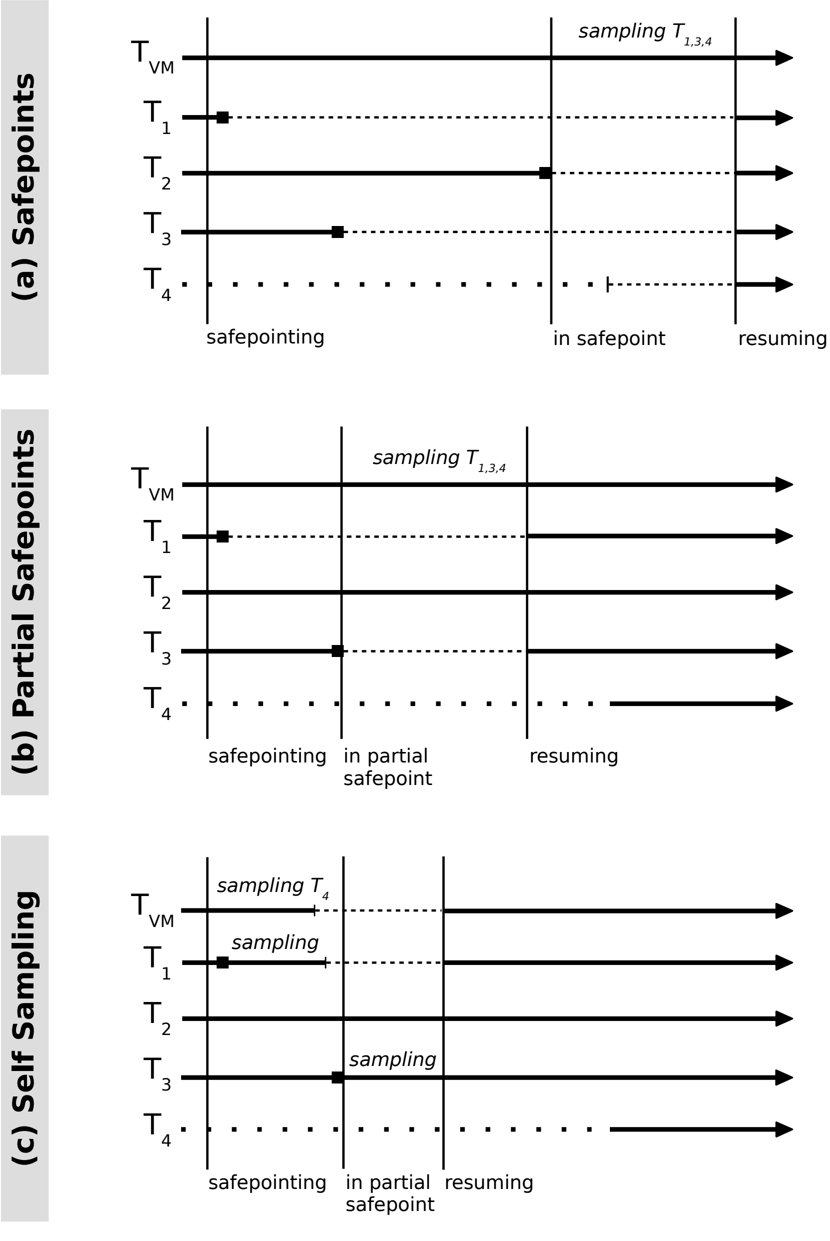


Figure 4.1: Example timeline for sampling three threads with (a) Safepoints, (b) Partial Safepoints, and (c) Partial Safepoints with Self Sampling

Apart from synchronizing all threads, another problem of the safepoint mechanism is the position of the safepoint checks. Usually, JIT compilers place these checks at the end of methods and loop iterations. Although these checks are very efficient, they can become a bottleneck in very hot loops. Moreover, safepoints prevent certain other optimizations like reordering of instructions. Therefore, the compiler tries to move safepoints out of loops or eliminate them completely from inlined code. All these optimizations come at the cost of increasing the worst case safepointing delay. Furthermore, when the compiler eliminates safepoints, the profiler can also see less program positions.

4.1 Partial Safepoints

As already discussed, safepoints are not very efficient for sampling because they require all threads to synchronize. For most sampling profilers it is acceptable or even wanted to not sample all threads. Usually running threads are the most interesting threads, because these threads are consuming resources. Thus, it would be a good idea to only sample running threads. The problem is that only the operating system scheduler knows the currently running threads. The Java VM only knows which threads are runnable (i. e., which threads are ready to run), but not which are running.

Partial safepoints allow reducing the safepointing delay by only waiting for a certain number of threads to enter a safepoint. The agent provides a list of threads and a number of threads n that should be sampled. The partial safepoint algorithm then waits for n threads from the list to enter a safepoint. Therefore, the algorithm selects those threads that enter a safepoint fastest. When n threads are in the safepoint, the VM thread takes the stack traces of these threads. When using the number of processor cores for n, the algorithm allows approximating sampling of running threads. Certainly this is not exact, as less than n application threads might be running because other processes require CPU time or the scheduler does a context switch while the VM is safepointing. Nevertheless, it is a good approximation for an userspace process without scheduling information and usually the running threads should be the first threads doing a safepoint check. In the worst case we sample a thread that was only runnable instead of running, which should be acceptable.

Sometimes it is useful to also sample waiting threads, for example to reveal bottlenecks with synchronization and I/O. The partial safepoint concept described above does not sample waiting threads, but it can be extended to also include waiting threads. Instead of sampling n runnable threads, the algorithm determines the ratio between runnable and waiting threads and distributes the n samples according to this ratio. The selection of runnable threads works as before with the first-come first-served principle, while the waiting threads are selected randomly. For an application with seven runnable and three waiting threads, the algorithm would take samples of three runnable and one waiting thread on a quad-core machine. Figure 4.1 (b) shows the previously discussed example with partial safepoints. We have four application threads, where one is in I/O (i. e., in native code) and the remaining three are runnable. The agent wants to sample three threads and requests partial safepoint sampling by calling a JVMTI extension method similar to `GetThreadListStackTraces`. It supplies all four application threads in the thread list and therefore does not limit the algorithms choice of threads. Based on the calculated ratio the algorithm decides to sample two runnable threads and one waiting thread. The VM thread sets the safepoint state to *safepointing*, signals a safepoint by setting the polling page protected and waits until two runnable threads run into a safepoint check. T_1 is the first thread running into the safepoint check and parking itself, then T_3 follows. Now that two runnable threads are in a safepoint, the VM is in a partial safepoint and does not need to wait for T_2. It sets the polling page unprotected, so that no further threads are parked when they run into a polling page check, but the safepoint state still remains at *safepointing.* The VM thread now samples the two runnable threads T_1, T_3 and the waiting thread T_4 that is the only possible choice in this example. After sampling, the VM exits the partial safepoint by resetting the safepoint state and resuming all application threads. When T_4 returns from its I/O request, no safepoint is active and the thread can immediately transition to the runnable state. This example showed that a full safepoint state is unnecessary for sampling stack traces.

Self Sampling

With partial safepoints the VM thread still does all the sampling work, while the application threads remain parked. This is suboptimal in a multi-core environment, where it would be better to distribute the work among multiple threads instead of having a single thread doing all work sequentially. As incremental stack tracing works independently for each thread, it can run in parallel without any difficulty. Full stack tracing could also be implemented in a parallel way.

Self sampling is an extension to partial safepoints, where each runnable thread samples itself, instead of having the VM thread sample all threads. Additionally, each thread does not need to wait until enough threads have reached a safepoint check, but can start sampling itself when it enters a safepoint. To guarantee the correct number of sampled threads a ticket system is used for synchronization. Each thread draws a ticket, which determines the order they reached the safepoint check. A thread only starts self sampling when not enough other threads took a sample before. The ticket also determines which thread is the last thread running into a safepoint check. The last thread has to notify the VM thread, so it can unprotect the polling page to avoid that further threads run into a safepoint. Waiting threads, which are currently blocking in a system call or executing some native library code, cannot sample themselves and the VM thread needs to sample them. Nevertheless, it can sample the waiting threads immediately after signalling a safepoint and before waiting for the runnable threads to reach a safepoint check or finish self sampling. Hence, sampling waiting threads can run in parallel to the self sampling of runnable threads.

Figure 4.1 (c) shows the previous example with self sampling. The agent again requests three samples and lets the partial safepoint algorithm choose from all four application threads. The algorithm decides to sample two runnable and one waiting thread. Then the VM thread changes the safepoint state to *safepointing*, signals a safepoint by setting the polling page protected and then begins to sample the waiting thread T_4. In the meantime, T_1 runs into a safepoint check as first thread. It draws a ticket and starts self sampling. The VM thread finishes sampling of T_4 and now waits for a second thread to run into a safepoint. Meanwhile, T_1 finishes self sampling and parks itself. T_3 runs into a safepoint check and draws a ticket. As it is the last thread reaching the safepoint, it notifies the VM thread, which unprotects the polling page and then waits for all self sampling threads to finish. The safepoint state stays at *safepointing.* The VM is now in a partial safepoint, but only remains there until T_3 finishes self sampling. When all threads have finished self sampling, the VM thread exits the partial safepoint by resetting the safepoint state and resuming all application threads.

4.2 Unix Signals

The partial safepoint and self sampling technique optimizes the latency of the safepoint mechanism, but it does not improve the accuracy. It still can only sample the program at the location of safepoint checks, which the JIT compiler tries to avoid because of performance reasons.

Unix signals are an inter-process communication mechanism that allows processes to send asynchronous notifications to other processes or threads [3]. Signals can interrupt a program at any position and therefore can greatly improve the accuracy of sampling. Nevertheless, interrupting a thread, especially a Java thread, at any position has various problems described later in this section. Unix signals are available on many operating systems, e. g., Linux, Solaris, and Mac OS X, but not on all target platforms of the Oracle HotSpot VM.

Sampling using Unix signals works as follows: The VM must register a signal handler for a special sampling signal (e. g., SIGPROF). When the agent wants to take a sample of an application thread, it sends the sampling signal to this thread. The operating system interrupts the thread and the control flow continues in the previously registered signal handler. The signal handler then either takes the sample directly or, like with incremental stack tracing, does the initial work for it. When the signal handler finishes, the normal program flow continues. An advantage is that taking a sample of a thread only affects this one thread and no other threads. However, Unix signals are unidirectional and usually do not carry additional information with them. Although the operating system allows sending a single integer number with a signal, the sample itself must be returned to the agent differently. With incremental stack tracing, the agent sends a request identifier with the signal and the incremental stack tracing algorithm stores the sample in the thread's internal data structure. The agent later retrieves the samples via the `RetrieveStackTraces` VM operation requiring a safepoint. The performance impact of this VM operation should be negligible because the stack traces are retrieved less often than new stack traces are requested.

Because signals can interrupt a program at any position, signal handlers must be careful about what operations they can perform. The signal handler must be async-signal-safe, which means when a signal interrupts a function at any position, it must be safe to also call the function in the signal handler. Therefore, it is very limited what functions the signal handler is allowed to call. Async-signal-safety is different from thread-safety. A thread-safe method does not have to be async-signal-safe and an async-signal-safe function is not automatically thread-safe. The POSIX standard defines some async-signal-safe methods, which can be used in signal handlers [6]. Nevertheless, it does not include often used functions like `malloc()`, `free()` or `printf()`, so they are not safe to use in signal handlers.

Implementation

An attempt was made to implement incremental stack tracing using Unix signals. This section describes the challenges that were encountered.

In the HotSpot VM, walking the stack is not async-signal-safe, thus it is hard to guarantee the async-signal-safety of the signal handler. One way of doing this is to detect all unsafe situations and ignore the sampling request in these situations. The main disadvantage is that there are a lot of different unsafe situations, which all must be detected. Although many situations do not happen very often, if one situation is missed the VM can enter an undefined state and most likely will crash immediately or at a later time. For example incremental stack tracing should not interrupt itself, because a request and a return intercept both change the CAO. When a request would interrupt a return intercept, the seen return would not match the CAO anymore and the algorithm is in an undefined state. Therefore, the algorithm must guarantee that new requests do not interfere with the return interception or the retrieve operation. It can achieve this by setting a flag when return interception is active and let the signal handler check for it. Other unsafe situations like on-stack replacement, deoptimization, garbage collection and exiting threads can be detected similarly. Nevertheless, there are more complicated cases.

As already described in chapter 3 the HotSpot Java VM does not store frame linkage information in stack frames of all types and therefore it is hard to walk the stack. This also implies that the VM can only walk the stack in safe situations, when its metadata matches the stack. During the stack frame construction in the method prolog and the stack frame destruction in the method epilogue, the size of the frame on the stack changes. The metadata contains only one frame size for each method, which therefore does not match during the method prolog and epilogue. In the case of the prolog, the VM can detect this situation with the `frame_complete_offset` stored in the method's metadata, which points to the end of the prolog. However, there is no equivalent for the method epilogue and methods often have multiple epilogues intermixed within the normal code, so a single offset is insufficient. An attempt to detect method epilogues is to scan for return instructions in the compiled code, starting from the current position. The problem with this approach is that a method epilogue not necessarily contains a return instruction. Sometimes the JIT compilers also use jump instructions for returns, which cannot easily be distinguished from jump instructions used for conditions and loops.

Another problem is compiler to interpreter (C2I) and interpreter to compiler (I2C) adapter code. When compiled code calls into the interpreter or the other way around, adapter code rearranges the stack to match the interpreter's or the JIT compiler's calling convention. The adapter code also reads the return address from the stack into a register and later writes it back to the stack. When a signal interrupts the VM before the return address is written back to the stack, the state looks safe, but when incremental stack tracing patches the return address on the stack, the adapter code later overwrites the patched return address with the original return address. Therefore, the algorithm misses the return and the data structure is not updated anymore. A solution to this is finding all code locations in the VM with such behaviour and guard them with a flag. Many such code locations were guarded, but checking the whole HotSpot code base for this behaviour is a too time-consuming task.

Overall, the HotSpot VM is not built for stack tracing at arbitrary points, and the undocumented `AsyncGetCallTrace` function using Unix signals for stack tracing is also not bulletproof. The implementation proves the concept of incremental stack tracing with Unix signals, but is not very stable because of the heuristic method epilogue detection. To improve this it would be necessary to maintain more extensive metadata about methods which specifies the exact frame size for each method part. Generating this metadata requires substantial changes in the JIT compilers.

Chapter 5

Experimental Results

This chapter describes the conducted experiments for comparing

- full safepoint sampling, referred to as: JVMTI,
- incremental sampling in full safepoints (incremental),
- self sampling in partial safepoints (partial),
- incremental self sampling in partial safepoints (partial incremental),
- full Unix signal-triggered sampling (AsyncGetCallTrace; AGCT signal), and
- incremental Unix signal-triggered sampling (incremental signal).

All experiments use the DaCapo 9.12 benchmark suite [2] and the Scala 0.1.0 benchmark suite [9]. The DaCapo suite consists of 14 Java benchmarks based on real-world applications. As the batik and eclipse benchmarks do not run with OpenJDK 8, they were excluded and only the 12 remaining benchmarks were used. The Scala suite consists of 12 Scala benchmarks based on real-world applications. Table 5.1 describes the used benchmarks shortly.

All benchmarks were executed with the default workload provided by DaCapo and Scalabench. An unmodified OpenJDK version 8u5-b13 was used for the reference runs without sampling and the existing sampling techniques JVMTI and AsyncGetCallTrace. For the novel techniques described in this thesis, a modified OpenJDK with the developed VM extension methods based on the version 8u5-b13 was used. Both the modified and unmodified OpenJDK were compiled in an identical way. Depending on the sampling technique, the VM was executed with an agent using either JVMTI, AsyncGetCallTrace or the developed VM extension methods for sampling. The agent takes samples in a specified sampling interval and builds CCTs from it. It tries to maintain a constant sampling interval by compensating the latency of sampling, i. e., subtracting the sampling latency from the wait time between samples. For the partial safepoint variants the sampling of waiting threads was enabled to allow comparison with full safepoint-based techniques that cannot restrict sampling to running threads. Furthermore, the number of cores (4) was chosen as number of samples n. Each benchmark was executed in 10

avrora	simulates a number of programs executing on a grid of AVR microcontrollers
fop	produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
h2	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application
jython	interprets the pybench Python benchmark
luindex	uses lucene to index a set of documents; the works of Shakespeare and the King James Bible
lusearch	uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyzes a set of Java classes for a range of source code problems
sunflow	renders a set of images using ray tracing
tomcat	runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages
tradebeans	runs the daytrader benchmark via a Java Beans to a GERONIMO backend with an in-memory h2 as the underlying database
tradesoap	runs the daytrader benchmark via a SOAP to a GERONIMO backend with in-memory h2 as the underlying database
xalan	transforms XML documents into HTML
actors	Trading sample with Scala and Akka actors
apparat	Framework to optimize ABC, SWC, and SWF files
factorie	Toolkit for deployable probabilistic modeling
kiama	Library for language processing
scalac	Compiler for the Scala 2 language
scaladoc	Scala documentation tool
scalap	Scala classfile decoder
scalariform	Code formatter for Scala
scalatest	Testing toolkit for Scala and Java programmers
scalaxb	XML data-binding tool
specs	Behaviour-driven design framework
tmt	Stanford Topic Modeling Toolbox

Table 5.1: Benchmarks used from the DaCapo and Scala benchmark suite [2, 9]

rounds, where each round consisted of 30 iterations in one VM process. To consider the VM's startup phase, the first 20 iterations were discarded and only the last 10 iterations were used, resulting in a total of 100 iterations per benchmark (10 rounds and 10 used iterations per round). The agent has to keep track of the start and end of iterations, so it can collect the statistics and build the CCT for each iteration individually. The distinction between rounds and iterations helps to factor out the bias of optimization decisions encountered during multiple iterations in the same VM process.

The experiments were run on a x86-64 system with Intel Core i7-3770 quad-core processor and 16 GiB memory. The system was running Ubuntu Linux 14.04 LTS with only basic system services, but no other applications executing. Moreover, hyper-threading, dynamic frequency scaling and turbo boost were disabled to get more consistent data.

5.1 Overhead

Figure 5.1 shows the median overhead of all six techniques with a sampling interval of 10 ms. The overhead is always given relative to no sampling, also for the following overhead figures. The bars are grouped into DaCapo and Scala benchmarks. The error bars represent the first and third quartiles. The bars labeled with *G. Mean* show the geometric mean over the median overhead of individual benchmarks with the error bars representing a 50% confidence interval. Figure 5.2 shows the overhead at a sampling interval of 1 ms and Figure 5.3 shows the overhead at a sampling interval of 0.1 ms. The *actors* benchmark does not succeed when sampled with JVMTI at 0.1 ms interval, as the benchmark has a time limit and JVMTI slows down the benchmark too much, making it exceed this time limit. Some benchmarks like *avrora* and *scalap* show a speedup with certain sampling techniques. This is not caused by the technique itself, but due to the fact that the operating system's thread scheduling changes slightly with sampling, which may have positive effects on synchronization.

Overall, partial incremental sampling provides the best performance at all three tested sampling intervals. Surprisingly, even the signal-based techniques AGCT signal and incremental signal have a higher mean overhead. This is mainly caused by the preallocation of method handles at class load, necessary for the signal-based techniques, as it cannot be done safely in a signal handler. For safepoint-based techniques it is not required. Especially at the *scalatest* benchmark, which loads a high number of classes in a short time, an extremely high overhead was measured for the signal-based techniques.

At a sampling interval of 10 ms, partial incremental is on par with partial sampling and incremental signal is on par with AGCT signal sampling. However, the incremental techniques cannot show a real benefit at this long sampling interval. The incremental technique using full safepoints shows even higher overhead than JVMTI sampling.

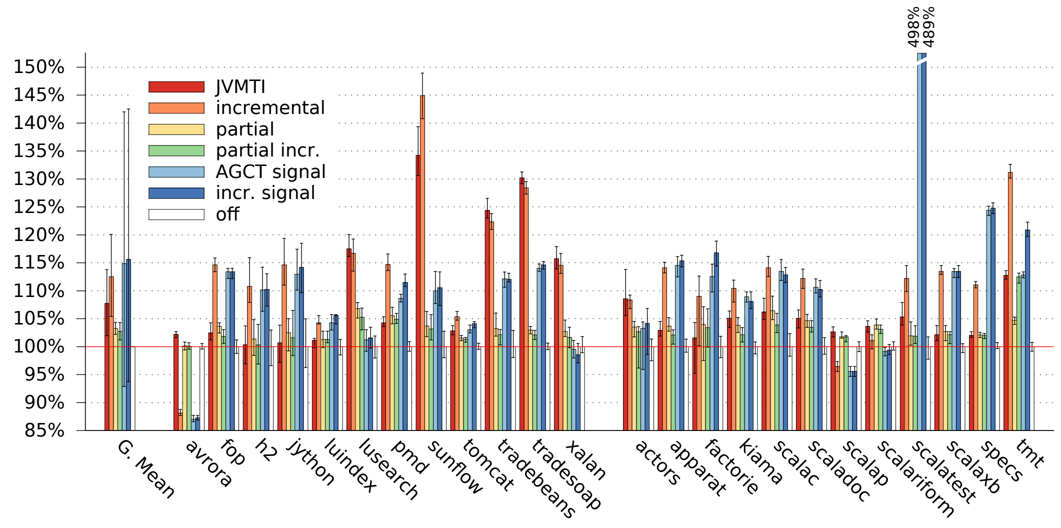


Figure 5.1: Overhead for benchmarks with 10 ms sampling interval

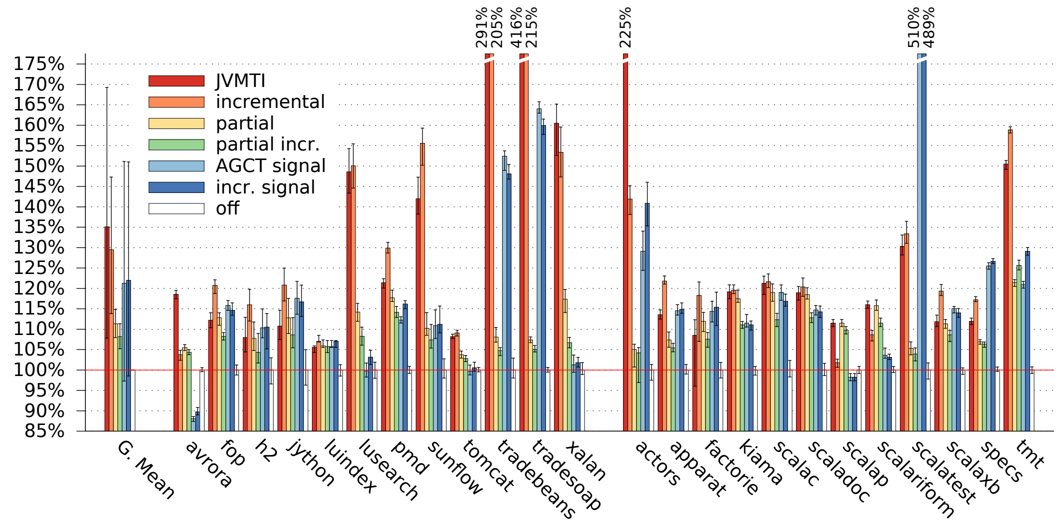


Figure 5.2: Overhead for benchmarks with 1 ms sampling interval

Nevertheless, at shorter sampling intervals the incremental techniques can greatly improve the overhead, especially in combination with partial safepoints at a sampling interval of 0.1 ms. One exception is the *tmt* benchmark, which has many short-lived threads. The incremental techniques have to retrieve the stack tracing data before a thread exits, therefore they have a disadvantage at such applications.

For the benchmarks *lusearch*, *sunflow*, *tradebeans*, *tradesoap*, *xalan*, and *actors* the measurements of the full safepoint-based techniques at sampling intervals 1 ms and 0.1 ms are not directly comparable to the other techniques, as the safepoint mechanism is already at its limit. In these scenarios taking a sample often already takes longer

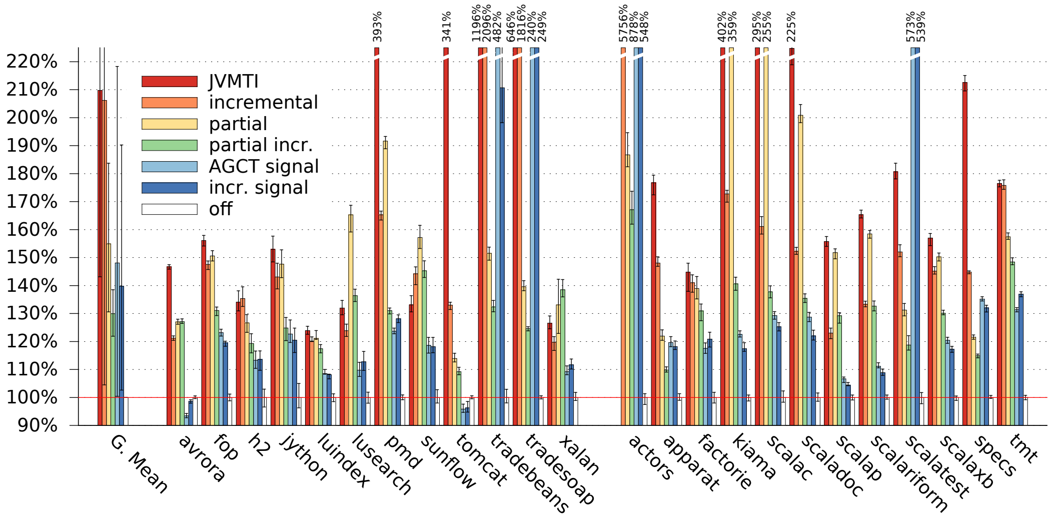


Figure 5.3: Overhead for benchmarks with 0.1 ms sampling interval

than the sampling interval. Thus, fewer samples than expected can be taken, resulting in an effective sampling interval longer than intended. This distorts the overhead, as fewer samples have to be processed. Section 5.2 shows this problem in more detail by analysing the latency of the sampling techniques.

Partial safepoints show a high performance gain, especially at benchmarks with a high number of threads like *tradebeans*, *tradesoap* and *actors*. Although partial incremental is not the fastest technique for every benchmark, it clearly provides the most stable overhead over all benchmarks. It performs worst for the actors benchmark at 0.1 ms sampling interval with about 67% overhead, while the worst performance of every other technique is beyond 350%. At the longer sampling intervals 10 ms and 1 ms, the partial technique without incremental stack tracing performs best in terms of worst performance at individual benchmarks, because of the *tmt* benchmark with its short-lived threads, as described before. For 10 ms sampling interval this is the *lusearch* benchmark with about 7% overhead. For 1 ms sampling interval this is the *tmt* benchmark with about 21% overhead. In contrast, the partial incremental techniques perform worst in the *tmt* benchmark with about 12% and 26% overhead at sampling intervals 10 ms and 1 ms. On average the partial incremental technique is still the fastest with 2.8%, 8.2%, and 29.9% overhead for sampling intervals 10 ms, 1 ms, and 0.1 ms.

5.2 Latency

Figure 5.4 shows box plots of the sampling latency, i. e., how long it takes to request one sample. For synchronous techniques, such as JVMTI and partial, this includes interruption of the threads, capturing the full stack traces and inserting them in the

CCT. For incremental and partial incremental, it includes interruption of the threads and capturing the top frame, while for incremental signal it only contains the time necessary to send the Unix signals. AGCT signal was not measured, because requesting a sample works similar to incremental signal. The whiskers indicate the 2.5% and 97.5% percentiles. In essence the Figure shows by how much the wait time between samples is reduced to compensate the sampling latency, so that the effective sampling interval stays constant. The horizontal line at 1 ms marks the used sampling interval. When the sampling latency exceeds the sampling interval, the agent cannot compensate it anymore, the effective sampling interval increases and fewer samples than expected are taken.

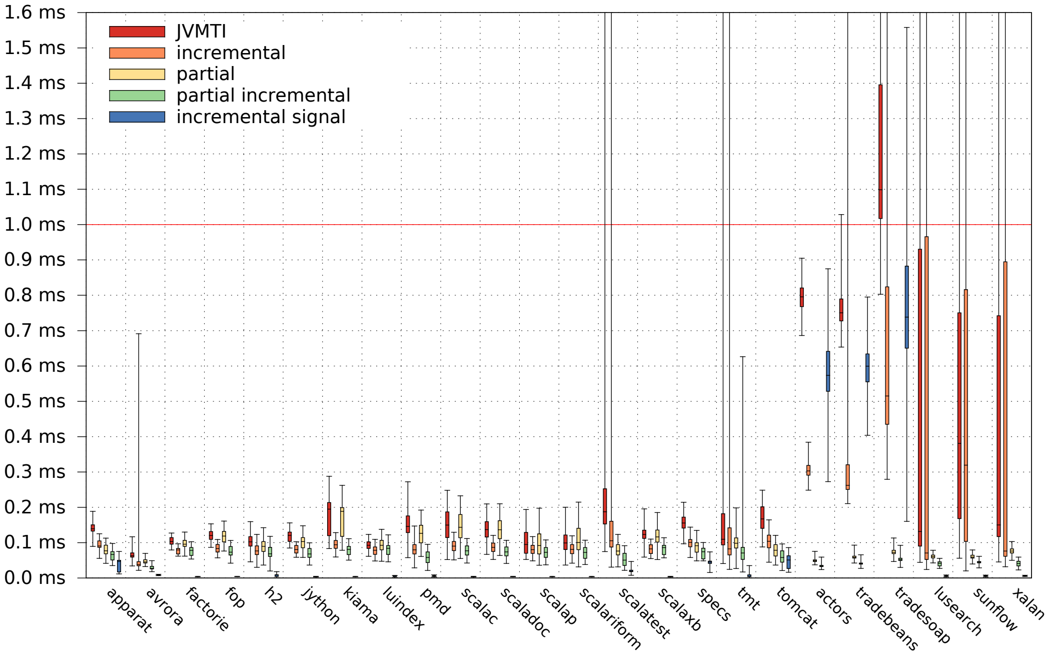


Figure 5.4: Sampling latency of benchmarks with 1 ms sampling interval

Most techniques stay far below the sampling interval for most benchmarks. However, there are benchmarks at which the full safepoint-based techniques JVMTI and incremental are at its limit. There are two groups of benchmarks, which are problematic with those techniques.

The first group consists of benchmarks like *actors*, *tradebeans* and *tradesoap*, which have a huge amount of threads. With their number of threads, it takes too long until all threads reach a safepoint. Unexpectedly, the incremental signal technique is also problematic with many threads, although it only does a single system call per thread.

The second group of benchmarks consists of *lusearch*, *sunflow* and *xalan*, which only have a few threads, but are CPU-intensive and contain very hot loops. The compiler aggressively optimizes these hot loops and therefore also eliminates safepoint checks. With the greater distance between safepoint checks, the safepointing delay also increases, which results in a high sampling latency. Although the median sampling latency for these benchmarks is not extremely high, it is not stable and often exceeds 10 ms. Therefore, in a time period where the agent was expected to take 10 samples, it effectively only took one sample.

With few threads, the incremental signal technique provides an incredibly low latency. However, with a larger number of threads, this benefit vanishes. Only the partial safepoint techniques partial and partial incremental can provide a stable sampling latency across all benchmarks. On average partial incremental is faster than the partial technique. Moreover, partial incremental is also the only technique with a median sampling delay below 0.1 ms for each benchmark.

5.3 Accuracy

Analysing the absolute accuracy of a profiler is a hard task. Ideally, an exact profile would be available for comparison. However, such an exact profile cannot be obtained because each profiler influences the profiled application in some way. Although instrumenting profilers can obtain a complete application profile with all function calls, time measurements of such a profile are heavily distorted, because the instrumentation adds considerable overhead in short methods. Hence, it is no use for comparison with profiles obtained through sampling profilers, which may be incomplete, but are not distorted by short-running methods.

One viable solution is to compare different sampling profilers to each other. For this comparison a sampling interval of 1 ms was used, but other sampling intervals (10 ms and 0.1 ms) produced similar results. The developed agent yields individual CCTs for each benchmark iteration, which is inconvenient for comparison, because of differences between iterations. Hence, the idea is to merge the CCTs of individual benchmark iterations to an averaged CCT per benchmark. Some benchmarks use dynamically generated classes, which get different names with each run. The used analysis tools consider this by applying heuristics when matching edges of CCTs. The averaged CCTs per benchmark are then used to compare the sampling techniques. Popular metrics for comparing CCTs are degree of overlap and hot-edge coverage [11]. The degree of overlap compares the weight of all edges between the CCTs, meaning that the whole CCTs are considered. Figure 5.5 shows the definition of the degree of overlap for two CCTs CCT_1 and CCT_2. $weight(e, cct)$ gives the relative weight of the edge e in cct. For performance analysis of applications, only the hot methods are typically useful and the remaining CCT is not of interest, which is better represented by the hot-edge coverage. Figure 5.6 shows the definition of the hot-edge coverage of CCT_1 over CCT_2: it calculates a set of hot edges in both CCTs and compares to which degree these sets match. The set of hot edges is determined via a relative threshold T from the hottest method. For the following comparisons a threshold of $T = 0.1$ was used.

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

Figure 5.5: Degree of overlap between CCT_1 and CCT_2

$$\begin{aligned} hotcover(CCT_1, CCT_2, T) &= \frac{|hotset(CCT_1, T) \cap hotset(CCT_2, T)|}{|hotset(CCT_2, T)|} \\ hotset(CCT, T) &= \forall e \in CCT \wedge weight(e, CCT) \geq T * hottest(CCT) : e \\ hottest(CCT) &= max(\forall e \in CCT : weight(e, CCT)) \end{aligned}$$

Figure 5.6: Hot-edge coverage of CCT_1 over CCT_2 with threshold T

Sampling applications with a rather short runtime only yields a low number of samples in total. If an application has a vast number of different states and too few samples are available, this can be problematic due to the statistical nature of sampling profilers. Two runs of such applications can produce CCTs with a low degree of overlap, as it is unlikely that both CCTs contain the same application states. Most problematic are deeply recursive applications, which have an extremely high number of different states that can be captured. The hot-edge coverage is less error-prone in this regard, as only hot parts are considered. Still, it can result in problems with applications that have many methods close to the determined threshold, which only sometimes are within the hot set and sometimes not.

Figure 5.7 shows the median overlap and hot-edge coverage of the individual CCTs to the averaged CCT, i. e., it shows the stability of the various profiling techniques per benchmark. The error bars show the first and third quartiles. The overlap and hot-edge coverage of the safepoint-based techniques JVMTI, incremental, partial and partial incremental is very similar. The low values at the *fop*, *kiama* and *scalaxb* benchmarks can be explained with their very short runtime (below 300 ms). Additionally, the *kiama*, *scalac* and *scaladoc* benchmarks are deeply recursive benchmarks, where it is hard to capture the same state multiple times. AGCT signal is worse at certain benchmarks, which should not be misinterpreted as disadvantage. The differences are mainly caused by its ability to capture more diverse application states than the safepoint-based techniques, which results in a lower overlap and hot-edge coverage especially at short running benchmarks. The incremental signal technique is only shown for completeness, since its various heuristics produce unstable results at some benchmarks.

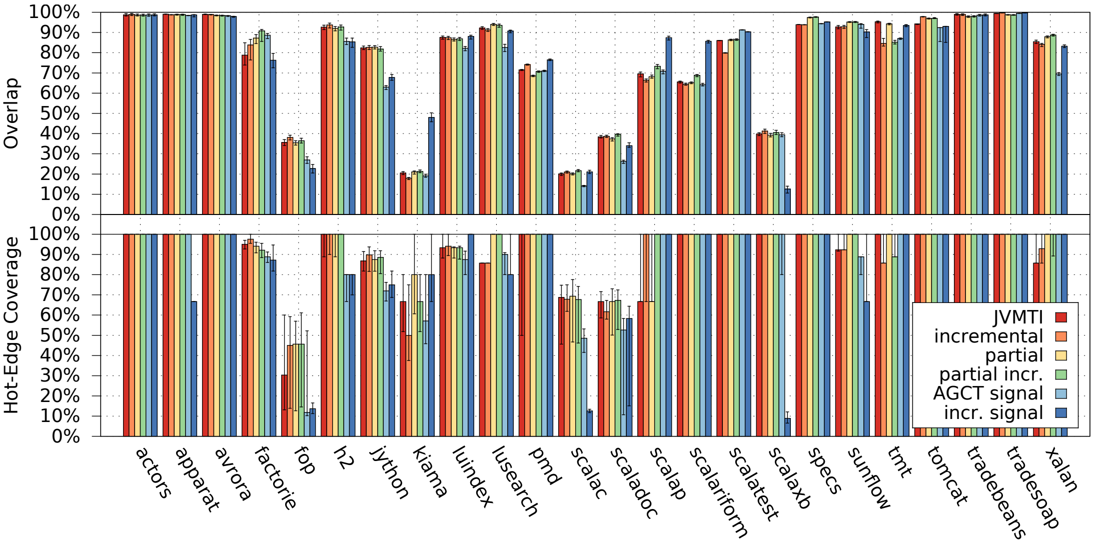


Figure 5.7: Median overlap and hot-edge coverage of individual run CCTs with the merged CCTs

Figure 5.8 shows the overlap and hot-edge coverage when comparing the merged CCTs of JVMTI with the other merged CCTs, i. e., it shows the agreement between JVMTI and the other techniques. The overlap of all safepoint-based techniques is above 70%, the only exceptions are the deeply recursive benchmarks *kiama*, *scalac* and *scaladoc*. The *kiama* benchmark has the deepest stacks of all benchmarks, which often already exceed the frame limit of 256 frames used for the full stack tracing methods JVMTI, partial and AGCT signal. These three techniques require a frame limit, as the number of frames currently on the stack is not known in advance and for performance reasons the memory is fully allocated in advance. A limit of 256 was chosen, as the stack of all benchmarks, except *kiama*, can fit in nearly all cases. The stacks of *kiama* would require a frame limit beyond 512 to fit most of the time and even with a frame limit of 1024 not all stacks would fit. The performance penalty of using such big frame limits is too big for a fair comparison. In contrast, all incremental techniques do not suffer from this problem, as they have no hard frame limit.

Figure 5.9 shows the overlap and hot-edge coverage when comparing the merged CCTs of AGCT signal with the other merged CCTs, i. e., it shows the agreement between AGCT signal and the other techniques. As expected, overlap and hot-edge coverage are consistently lower than in the comparison with JVMTI. Only the incremental signal technique tends to agree more with AGCT signal than with JVMTI, which is also expected. The reason, as described before, is that all Unix signal-based techniques are able to see more diverse application states than the safepoint-based techniques. Similar to the comparison with JVMTI, the incremental signal technique also provides no consistent results when compared to AGCT signal, which emphasises its experimental nature.

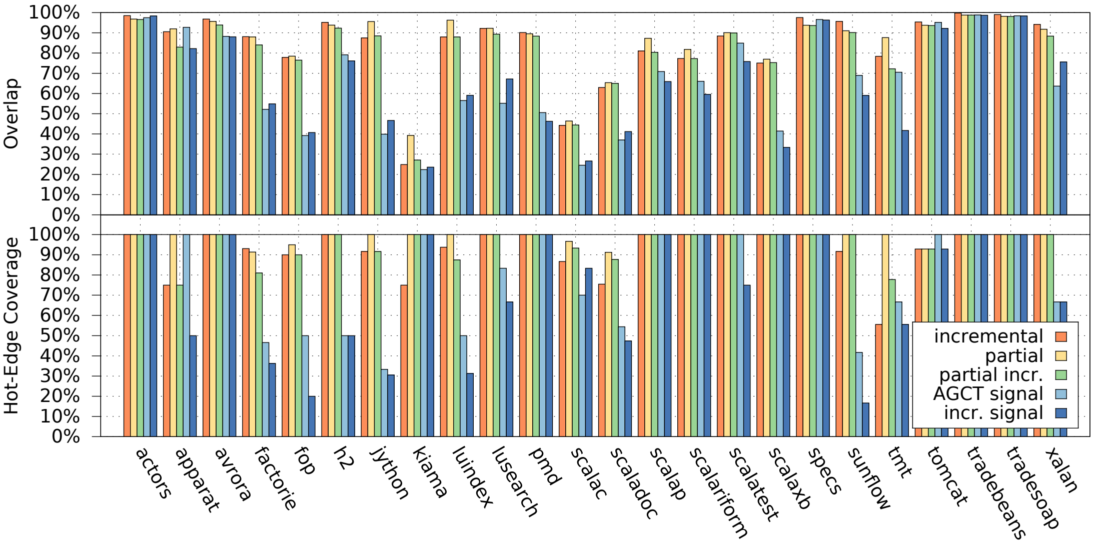


Figure 5.8: Overlap and hot-edge coverage of merged CCTs with the JVMTI merged CCTs

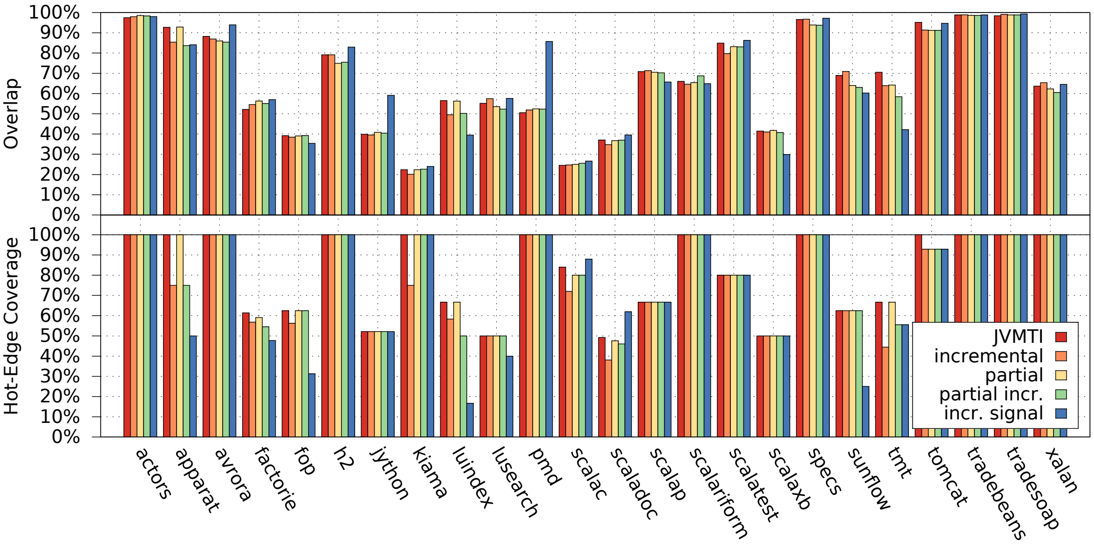


Figure 5.9: Overlap and hot-edge coverage of merged CCTs with the AGCT signal merged CCTs

Chapter 6

Conclusion

This thesis showed how to improve profiling in the HotSpot Java VM. The novel technique *incremental stack tracing* builds stack traces lazily, avoiding work that JVMTI would do redundantly, reducing the total stack tracing overhead. *Partial safepoints with self sampling* are a variant of safepoints that are optimized for sampling. They help to reduce the delay of taking a sample and therefore also reduce the overall overhead. Moreover, they allow approximating sampling of running threads, which is not easily possible in a userspace process without scheduling information.

The experimental results showed that incremental stack tracing and partial safepoints with self sampling can dramatically reduce the profiling overhead and average sampling delay. Especially with shorter sampling intervals and highly multi-threaded applications, JVMTI sampling is practically unusable. Incremental stack tracing and partial safepoints with self sampling still provide high performance in these scenarios.

In general, the accuracy of the application profiles is tied to the sampling mechanism and partial safepoints cannot improve over regular safepoints. Nevertheless, recursive applications with very deep stacks can benefit from incremental stack tracing regardless of the sampling mechanism, as it has no stack depth limit.

For improving the accuracy beyond that what safepoints can provide, a sampling mechanism using Unix signals would be a good option, as it can interrupt a program at any position. This ability to interrupt a program at any position is rather problematic, as the VM cannot walk the stack in an arbitrary state. `AsyncGetCallTrace` works around this issue by detecting unsafe situations and returning an error. Nevertheless, often the VM cannot detect an unsafe situation right away, but it detects it while walking the stack. This is no problem for `AsyncGetCallTrace`, but for incremental stack tracing it is critical, as it does not walk the stack at once and needs to patch return addresses on the stack. Therefore, incremental stack tracing has much stricter requirements on the detection of unsafe situations. It would be necessary to have a lot more metadata about the compiled code to detect unsafe situations immediately.

Another approach would be to introduce sampling points, which could work similar to safepoints. Safepoint checks limit the optimizations a compiler can do and therefore the compiler only injects few of them. In contrast, sampling point checks would have fewer limits on the compiler and let it freely optimize code across sampling points. The compiler can then combine and reorder instructions arbitrarily. This would also mean that a single bytecode index may not suffice to express the real program position. A developer usually does not think in this way about a program, thus it is necessary to abstract the program position at sampling points. A sampling point technique allows to freely adjust the accuracy by injecting either more or fewer sampling point checks. The sweet spot between performance overhead and profiling accuracy has to be found.

List of abbreviations

VM Virtual Machine

JVMTI Java Virtual Machine Tool Interface

JIT Just-In-Time

API Application Programming Interface

CCT Calling Context Tree

GC Garbage Collector

TFO Top Frame Object

CSO Current Skeleton Object

bci ByteCode Index

OSR On-Stack Replacement

AGCT AsyncGetCallTrace

List of Figures

Bibliography

[1] 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, New York, NY, USA, 1997. ACM.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[3] The Open Group. POSIX.1-2008: The Open Group Base Specifications Issue 7. Also published as IEEE Std 1003.1-2008, July 2008. `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[4] Peter Hofer, David Gnedt, and Hanspeter Mössenböck. Lightweight Java profiling with partial safepoints and incremental stack tracing. In *Proceedings of the 6th Int'l Conference on Performance Engineering*, 2015.

[5] 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 (TACO)*, 5(1):7, 2008.

[6] Linux Programmer's Manual: signal(7): Overview of signals. `http://man7.org/linux/man-pages/man7/signal.7.html`.

[7] Oracle. OpenJDK HotSpot Group. `http://openjdk.java.net/groups/hotspot/`.

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

[9] 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 26th Conference on Object-Oriented Programming, Systems,*

Languages and Applications, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM.

[10] 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, New York, NY, USA, 2009. ACM.

[11] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM.