Efficient and Accurate Stack Trace Sampling in the Java Hotspot Virtual Machine (Work in Progress Paper)

Peter Hofer
Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems
Johannes Kepler University Linz, Austria
peter.hofer@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Sampling is a popular approach to collecting data for profiling and monitoring, because it has a small impact on performance and does not modify the observed application. When sampling stack traces, they can be merged into a calling context tree that shows where the application spends its time and where performance problems lie. However, Java VM implementations usually rely on *safepoints* for sampling stack traces. Safepoints can cause inaccuracies and have a considerable performance impact.

We present a new approach that does not use safepoints, but instead relies on the operating system to take snapshots of the stack at arbitrary points. These snapshots are then asynchronously decoded to call traces, which are merged into a calling context tree. We show that we are able to decode over 90% of the snapshots, and that our approach has very small impact on performance even at high sampling rates.

1. INTRODUCTION

Software profiling measures the execution frequency or the run time of methods during program execution. It is useful for finding bottlenecks, identifying dead code, or determining test coverage. Typically, the profiling data is not captured per method, but rather by *calling context* (or *stack trace*), which is a chain of calls from the root method to an executing method. Calling contexts are commonly merged into a calling-context tree (CCT) [1]. In contrast to a call tree, a CCT merges identical children (callees) of a node.

In general, there are two approaches for collecting calling contexts. *Instrumenting profilers* insert code snippets in methods to record calls in the CCT. This approach yields a complete CCT, but the instrumentation can introduce significant overhead and can distort other observations such as method execution times. *Sampling profilers,* on the other hand, periodically interrupt the application to take snapshots of the entire chain of calls and to merge them into the CCT. This approach requires no instrumentation and typically causes less overhead, but it can miss method invocations and thus results in an approximate CCT with only statistically significant information. Our research focuses on the sampling approach, because we strive for minimal overhead.

The Java Virtual Machine Tool Interface (JVMTI) [7] offers functions for both sampling calling contexts and instrumenting code and is supported by all common Java VM implementations. Because of its lower overhead, many monitoring tools prefer sampling over exhaustive instrumentation. However, implementations of JVMTI rely on *safepoints* for sampling, which are special locations in code where it is safe to pause the application, for example to perform garbage collection. Whenever JVMTI takes a sample, it requires all threads to run to the next safepoint, which introduces considerable overhead and makes sampling intervals irregular.

Some JIT compiler optimizations add to these problems. For example, a compiler may omit certain safepoint checks to increase performance, which can considerably increase the time to reach a safepoint. When safepoints in an inlined method are omitted, the invocation of that method never shows up in the samples. On the other hand, safepoints in tiny methods such as getters or setters can cause these methods to be overrepresented in the samples.

As a faster and more accurate alternative to JVMTI sampling, Oracle's Hotspot Java VM [8] offers an undocumented sampling mechanism that does not use safepoints. Instead, it uses a POSIX signal that is sent to Java threads to interrupt them. The signal handler can then walk the thread's stack and build a call trace. Because threads can be interrupted individually and at any point (not just at safepoints), this mechanism has a much lower performance impact.

In this paper, we present yet another approach that relies on the operating system and hardware timers to sample fixed-size fragments of Java stacks and to copy them into a buffer. An agent asynchronously reads this buffer, creates call traces from the stack fragments and adds them to a CCT. The agent uses a modified version of the OpenJDK 8 Hotspot VM to decode the stack frames. Our approach achieves very low overhead because analysis happens asynchronously and threads are interrupted only for copying the stack fragments, which can be done quite efficiently.

The main contributions of this paper are:

1. We describe a new technique for collecting call traces of Java applications. It is more efficient than JVMTI sampling and allows for higher sampling rates. Our technique also provides more accurate call traces than JVMTI and thus gives a better picture of where an application spends its time.

ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright 2014 ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2576759.

2. We describe new heuristics for analyzing stack contents at arbitrary sampling points.

3. We demonstrate the efficiency of our approach with the DaCapo benchmark suite, and the accuracy of our approach by comparing the generated calling context trees to those produced with JVMTI.

2. APPROACH

2.1 Taking Stack Samples

For sampling the stacks, we rely on the operating system to set up a timer to interrupt the application at regular intervals, and when these interrupts trigger, to copy a 16 KByte fragment of the executing thread's current stack. We chose to use the *perf* subsystem of the Linux kernel which already offers this functionality [5, 11]. Sampling a running thread with perf is enabled through a system call that takes a wealth of parameters including the sampling interval, the kinds of data to sample, and the size of the stack fragments. Once perf is enabled, it makes a buffer of collected data available via a file descriptor. This file can be mapped into user-space memory and then acts as a ring buffer of events with the requested data.

The perf subsystem even provides functionality for building call traces by following frame pointers on the stack. However, Java stacks are too complex for this mechanism and even native compilers can omit these links to increase performance, so we perform the stack analysis in user space.

2.2 Retrieving the Samples

For retrieving and processing the samples, we implemented a native *agent* that runs within the Java VM and interfaces with it using JVMTI. The agent registers for a callback when the Java application's main thread is launched. In that callback, it calls perf to start sampling stack fragments as well as the instruction pointer, stack pointer and frame pointer on that thread. The agent also enables inheritance, so sampling automatically applies to all threads the Java application launches.

Next, the agent launches a separate *reader thread* that periodically reads the buffer supplied by perf. The reader thread waits until the buffer is filled to a "watermark" that can be set in the initial system call. It then retrieves samples from the buffer and first copies each stack snapshot to a local buffer where it can be modified. Next, it scans the snapshot for addresses that point into the live stack, and adjusts them to point into the snapshot itself. The reader thread then submits the adjusted snapshot as well as the frame pointer (adjusted to point into the snapshot) and the instruction pointer to the Java VM using a JVMTI extension method that we introduced. This method finally returns a call trace that the reader thread merges into a single CCT.

In some cases, it is necessary to process the collected samples before the buffer is filled to the watermark. One of these cases is when the VM decides to unload compiled methods. Snapshots in which the instruction pointer or return addresses refer to unloaded code could otherwise no longer be transformed to call traces. Similarly, when the VM decides to unload entire classes, it disposes metadata that can then no longer be used to decode snapshots. Hence, our agent registers callbacks for the corresponding JVMTI events to process all buffered samples when one of these situations occurs. Callbacks for such events are invoked from application or VM threads, so they need to synchronize access to the buffer with the reader thread.

2.3 Analyzing the Samples

Decoding stack snapshots to call traces requires knowledge of the frame layout, the frame sizes and other VM-internal information. Because all this is readily available within the VM, we decided to implement the analysis of stack snapshots within the OpenJDK 8 Hotspot VM and to make it available to the agent via JVMTI extension methods. Using JVMTI's extension mechanism also has the advantage that agents can probe for this capability and can fall back to some other type of sampling when it is not available.

We based our implementation on that of `AsyncGetCallTrace`, an undocumented function of the Hotspot VM for walking stacks from within a POSIX signal handler. This function already handles several of the intricacies of analyzing stacks in a state where the topmost Java frame cannot be clearly identified. However, unlike `AsyncGetCallTrace`, our approach analyzes stacks asynchronously and thus cannot access the VM state at the time the sample was taken.

One case where it is difficult to walk the stack is when the sample was taken while the thread was executing native code, which the VM has no knowledge about. This occurs frequently because Java relies heavily on native calls for I/O. If the native code does not establish a proper chain of frame pointers (e.g., due to a compiler optimization), its frames cannot be walked and the topmost Java frame, where the native call occurred, cannot be determined. In such cases, `AsyncGetCallTrace` can retrieve the location of this frame from a VM-internal structure where it was recorded at the time of the call, but since we are analyzing the samples asynchronously, we have no access to this information. Therefore, we resort to a heuristic: we leave a "breadcrumb" of two words with specific "magic" values on the stack when a Java-to-native call occurs. When the top of the Java stack cannot be found, we simply scan for a breadcrumb and perform additional checks when found.

Native code invoked from Java can again call Java code via the Java Native Interface (JNI). In such cases, the stack consists of alternating sequences of Java frames and native frames. The resulting stack trace should contain all Java frames on the stack and not just the top frames, which again requires detecting Java/native boundaries. In this case however, we need not rely on breadcrumbs: when native code calls Java methods, the call leaves an entry frame on the stack with a link to the last Java frame below the native caller. Using this link, the stack walk can reliably skip the native frames.

Another problem occurs when frames from stub code are on top of the stack. Stub code refers to snippets of machine code that the Hotspot VM dynamically generates as call wrappers, compiler intrinsics and other helper code. Stub frames have no common layout and some do not even have a known frame size. When a snapshot contains a stub frame with unknown size, we merely scan the words below the stack pointer for a word that looks like a return address into Java code. When such an address is found and additional checks reassure that it is at the boundary to a Java frame, the stub frame is ignored and the call trace is created starting from the Java frame.

Occasionally, samples are also taken in a method's prologue, which is the entry code that saves the caller's frame pointer on the stack, makes the current stack pointer the new frame pointer and moves the stack pointer to the end of the new frame. These samples then have an incomplete frame on top, but this can be detected because Hotspot records where in each compiled method the prologue ends. Instead of discarding these snapshots, we use simple heuristics to find the frame below: we compare the word on top of the stack to the stack pointer and frame pointer to test if the caller's frame pointer has been pushed on the stack yet. The word below, or if the frame pointer has not be pushed, the word on top, should be a return address to Java code and the boundary to the frame below. We then start the stack walk from that frame, ignoring the incomplete frame.

Every sample contains only a fixed-size fragment of the stack. Thus, the stack walk can arrive at the end of the fragment before reaching the end of the stack. The end of the Java stack is denoted by a special entry frame. When this entry frame is not reached, we set a flag that signals to the agent that the stack trace is incomplete.

Our heuristics considerably increase the percentage of stack snapshots that can be turned into stack traces. However, there are still situations in which the topmost Java frame cannot be determined conclusively. The analysis method then simply returns an error and the agent ignores the sample.

3. EVALUATION

We evaluated our approach using the DaCapo 9.12 benchmark suite [3], which consists of open source, real-world applications with pre-defined, non-trivial workloads[1]. To compensate for the warm-up phase of the VM, we chose to run ten successive iterations of each benchmark and to use only metrics collected in the last iteration. We used a stack fragment size of 16 KB and intervals of 10ms, 1ms and 0.1ms between samples. These translate to sampling rates of at most 100, 1000 and 10000 samples per thread per second, which we refer to below. We executed 50 runs of each benchmark at each of these three sampling rates.

3.1 Success Rate

First, we measured how many collected samples could be successfully decoded to a stack trace. Table 1 shows the percentages for each benchmark at 10000 samples per second in the row labeled *Res.* On average over all benchmarks, 90.7% of all stack snapshots could be successfully processed. The *avrora* benchmark shows the lowest rate at 84.3%, while nearly all samples could be used in the *sunflow* benchmark at 96.8%. In almost all cases where the stack walk fails, the problem was in identifying the top Java frame or its caller frame. Hence, additional or better heuristics could improve this rate further and are part of our ongoing research.

3.2 Completeness of Traces

We measured how often the Java stack was larger than the fragment size, leading to incomplete stack traces. The *Inc.* row in Table 1 shows the resulting percentages by benchmark at 10000 samples per second. For ten out of the twelve benchmarks, less than 0.06% of stack traces were incomplete.

[1]We did not use the DaCapo suite's *batik* and *eclipse* benchmarks because they do not run on OpenJDK 8.

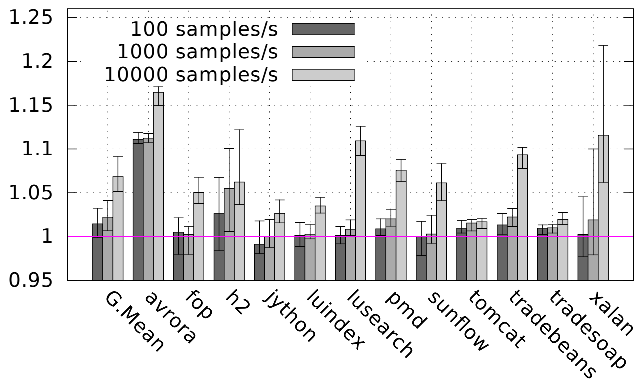

Figure 1: Sampling and analysis overhead for the DaCapo benchmarks

This shows that stack fragments with a size of 16 KB are typically sufficient to produce full stack traces.

With 4.09% incomplete stack traces, the *tomcat* benchmark seems to be an exception to this rule. However, increasing the fragment size had no substantial effect. We noticed that most of the incomplete stack traces are related to class loading and exception handling, and will examine these cases more closely to increase the completeness of our traces.

3.3 Performance Impact

Figure 1 shows the median execution time for each benchmark at the sampling rates of 100, 1000 and 10000 samples per second, normalized to the benchmark's median execution time without sampling. The error bars indicate the first and third quartiles. The *G.Mean* bars on the left show the geometric means over all benchmarks, which are 1.45%, 2.21% and 6.83% overhead at 100, 1000 and 10000 samples per second, respectively. Only the *avrora* benchmark shows consistently above 10% overhead.

For comparison, we implemented an additional agent that collects samples using JVMTI and builds a CCT from them as well. With OpenJDK 8, we determined a geometric mean overhead over all benchmarks of 5.4% at 100 samples per second, 45.2% at 1000 samples per second and 73.0% overhead at 10000 samples per second. Hence, our approach is clearly superior to JVMTI sampling, particularly at high sampling rates.

3.4 Accuracy

For assessing the accuracy of our approach, we compared the CCTs produced by our agent to those from the JVMTI agent. Our tests confirm that JVMTI frequently misrepresents how an application spends its time. An extreme example is SciMark 2.0 [9], a scientific computing benchmark that spends roughly equal times in five different computational kernels. The CCT with data from JVMTI, however, suggests that the program spends over 40% of its time in `kernel.measureLU()`, which represents just one of the five kernels, and in fact calls another method `LU.factor()` that performs the actual work. This suggests that the safepoints are placed unfavorably in SciMark so that the distribution of samples is distorted. The CCT produced with our approach distributes the execution time more evenly and attributes it to the computationally intensive methods.

	Mean	avrora	fop	h2	jython	luindex	lusearch	pmd	sunflow	tomcat	tradebeans	tradesoap	xalan
Res.	**90.7%**	84.3%	91.6%	88.7%	91.9%	96.0%	95.7%	85.8%	96.8%	87.8%	87.7%	88.3%	93.6%
Inc.	**0.41%**	0.00%	0.48%	0.01%	0.05%	0.04%	0.01%	0.06%	0.00%	4.09%	0.01%	0.03%	0.11%

Table 1: Fraction of resolved and incomplete stack traces by benchmark

3.5 Achievable Sampling Rates

Our approach allows for high sampling rates because of its low impact on performance and because taking samples is triggered by a hardware timer and is not delayed by any software mechanism. We were able to achieve sampling rates of more than 30000 samples per second. In comparison, the quickly increasing overhead of JVMTI sampling sets a much lower practical limit. Using POSIX signals with `AsyncGetCallTrace` has a practical limit as well, because rapidly successive signals are either coalesced into one signal or are all handled at once without the application running in between, producing samples of an identical state.

3.6 Redundancy of Samples

With both JVMTI sampling and `AsyncGetCallTrace`, it is not possible to determine whether a thread has been active before taking a sample. As a result, samples are often taken of unchanged states, such as when a thread is waiting. With our approach, the operating system only takes samples while a monitored thread is executing, or when a context switch occurs, which allows us to capture a thread's calling context before it enters a waiting state. This reduces the amount of collected data that needs to be processed.

4. RELATED WORK

Problems with Java stack sampling have been analyzed before. Mytkowicz et al. [6] demonstrate that four commonly used Java profilers often produce incorrect profiles due to safepoints and optimizations. They propose a profiler that pauses threads at arbitrary locations, but they choose to implement it entirely outside the VM, which restricts its use in the presence of optimizations. Binder [2] confirms the high overhead of sampling with JVMTI and presents a pure Java profiler that rewrites bytecode to maintain a shadow stack and periodically capture samples, which he claims is more accurate than JVMTI at comparable overhead.

To our knowledge, copying stack fragments for fast Java profiling has not been attempted before. Whaley [12] describes an in-VM Java profiler that samples threads at arbitrary points and avoids full stack walks by marking visited stack frames, claiming a low overhead of 2-4% at 1000 samples per second. However, the used VM performs thread scheduling itself ("green threads"), which permits certain assumptions and direct access to thread states. Green threads are uncommon in modern Java VMs because of their disadvantages in multi-processor systems. Inoue and Nakatani [4] describe a Java profiler that uses hardware events to take samples of only the executing method and the stack depth. It builds a CCT based on matching stack depths and caller information, and is reported to achieve an overhead of 2.2% at 16000 samples per second. Serrano et al. [10] present a Java profiler that uses hardware branch tracing to create partial call traces and attempt to merge them optimally into approximate CCTs, claiming to produce highly accurate CCTs at negligible overhead. Unlike our approach, both of these techniques require specific hardware and their accuracy can suffer from ambiguous callers.

5. CONCLUSIONS AND FUTURE WORK

We described an approach for Java stack trace sampling that relies on the operating system to capture stack fragments which are then asynchronously retrieved and analyzed. We further described a set of heuristics for identifying the topmost Java frame in these fragments for building stack traces from them. Our preliminary results show that these heuristics work for over 90% of the collected fragments, and that a fragment size of 16 KB is sufficient to obtain complete stack traces from more than 99.5% of samples. We further showed that the accuracy of our approach is high while its performance impact is very low even at high sampling rates, particularly when compared to JVMTI sampling.

As next steps, we plan to experiment with smaller fragments and to determine how well incomplete stack traces can be correctly matched to a CCT. We also consider using hardware performance counters instead of taking samples at fixed time intervals, which could further reduce overhead.

6. ACKNOWLEDGEMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Compuware Austria GmbH.

7. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. PLDI '97, pages 85–96, 1997.

[2] W. Binder. Portable and accurate sampling profiling for Java. *Software: Practice and Experience*, 36(6):615–650, 2006.

[3] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. OOPSLA '06, pages 169–190, Oct. 2006.

[4] H. Inoue and T. Nakatani. How a Java VM can get more from a hardware performance monitor. OOPSLA '09, pages 137–154, 2009.

[5] kernel.org. perf: Linux profiling with performance counters. `https://perf.wiki.kernel.org/`.

[6] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. PLDI '10, pages 187–197, 2010.

[7] Oracle. JVM^{TM}Tool Interface version 1.2.1. `http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html`.

[8] Oracle. OpenJDK HotSpot group. `http://openjdk.java.net/groups/hotspot/`.

[9] R. Pozo and B. Miller. SciMark 2.0. `http://math.nist.gov/scimark2/`.

[10] M. Serrano and X. Zhuang. Building approximate calling context from partial call traces. CGO '09, pages 221–230, 2009.

[11] V. Weaver. The unofficial Linux perf events web-page. `http://web.eece.maine.edu/~vweaver/projects/perf_events/`.

[12] J. Whaley. A portable sampling-based profiler for Java virtual machines. Java Grande '00, pages 78–87, 2000.