Profiling and Tracing Support for Java Applications

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:29. Aug. 2022
Profiling and Tracing Support for Java Applications

Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján
School of Computer Science, University of Manchester, Manchester, UK
{name.surname}@manchester.ac.uk

ABSTRACT

We demonstrate the feasibility of undertaking performance evaluations for JVMs using: (1) a hybrid JVM/OS tool, such as async-profiler, (2) OS-centric profiling and tracing tools based on Linux perf, and (3) the Extended Berkeley Packet Filter Tracing (eBPF) framework where we demonstrate the rationale behind the standard offwaketime tool, for analysing the causes of blocking latencies, and our own eBPF-based tool bcc-java, that relates changes in microarchitecture performance counter values to the execution of individual JVM and application threads at low overhead.

The relative execution time overheads of the performance tools are illustrated for the DaCapo-bach-9.12 benchmarks with OpenJDK9 on an Intel Xeon E5-2690, running Ubuntu 16.04. Whereas sampling based tools can have up to 25% slowdown using 4kHz frequency, our tool bcc-java has a geometric mean of less than 5%. Only for the avrora benchmark, bcc-java has a significant overhead (37%) due to an unusually high number of futex system calls. Finally, we provide a discussion on the recommended approaches to solve specific performance use-case scenarios.

CCS CONCEPTS
• General and reference → Performance; • Software and its engineering → Virtual machines.

ACM Reference Format:

1 INTRODUCTION

We briefly survey standard performance evaluation approaches and tools, and their limitations for Java Virtual Machines (JVMs) concerning Garbage Collection (GC) log files, heap analysis, lock contention, processor core sampling, and bytecode based instrumentation for the measurement of application specific performance metrics (Table 1 summarises tool features and capabilities). Logging involves turning on JVM flags to monitor specific JVM subsystem behavior (e.g. JIT compilation or GC). The two main approaches to profiling are tracing and sampling. Tracing instruments code to measure microarchitecture or system performance metrics. Sampling profilers repeatedly collect the stack trace of called functions/methods, that describe the code running on the processing core, on-core, at the sampled moments in time.

The Unix OS perf tool is becoming widely used as it supports both sampling and tracing. In the Java community, sampling based profiling is widely implemented with JVM Tools Interface (JVTI) agent support using GetCallTrace (e.g. JProfiler). This approach is reliant on restricting sampling to safe-points, that are inserted by the JIT compiler to support GC. Such restrictions generate bias, and can lead to incorrect performance information. DTrace was a key tool popularizing tracing on Solaris, and then ported to FreeBSD, NetBSD, Mac OS, and in 2018 to Windows. Since 2016, Linux integrated similar functionality via the eBPF framework (Kernel version 4.9). This paper demonstrate the feasibility of undertaking performance evaluations for unmodified JVMs using either, (1), a hybrid JVM/OS tool, such as async-profiler, (2), OS-centric profiling and tracing tools based on perf, or (3) our own eBPF-based tracing tool, bcc-java. The contributions of this paper are:

We measure the overheads of on-core sampling with profiling for 100Hz (1 sample every 10ms), 1kHz (every 1ms) and 4kHz (every 250µs) for the DaCapo benchmarks. We find that 1kHz sampling can be used with the async-profiler and perf/perf-map-agent with a reasonable geometric overhead of less than 5.2% and 11.2% respectively compared to normal execution.

We show how flamegraphs produced using perf combined with perf-map-agent can be used to identify where JIT based compilation has failed to inline successfully.

We report that heap allocation based profiling using the async-profile tool is low overhead, at less than 2.8%, for all benchmarks, and that flamegraphs enable developers to easily determine the main allocation sources in their programs.

We present a low overhead tracing tool bcc-java (see Section 5) with a geometric mean overhead of 3.6% for characterizing the performance of all service and application threads created by a JVM. The bcc-java tool is developed on top of Linux eBPF Compiler Collection (BCC) [12, 23] for adding tracing support to operating system kernels and applications. Performance counter measurements such as instructions executed, processor cycles and cache misses can be directly related to application thread IDs, and to VM services. The monitoring techniques in the tool have generic applicability that could be deployed to characterize any multi-threaded application.

We demonstrate how the BCC offwaketime tracing tool can be used to produce flamegraph visualizations that describe important aspects of thread blocking and wakeup execution behavior without requiring JVM modifications.

We summarize the main features of the tools async-profiler, perf, offwaketime, and our new bcc-java tool in Table 1.

Section 2 generally discusses the capabilities of log file analysis tools targeting GC and JVM related memory performance analysis. Section 3 presents Flamegraphs [13] and how to interpret...
them for JVMs. Section 4 explains the capabilities of traditional Java based profiling tools and their limitations. The benefits of AsyncGetCallTrace based profiling, and the stack fragment sampling approach of [15] are discussed. Section 5 explains the rationale behind the design of the eBPF/BCC tools and the tracing tool that we have built. Section 6 presents our experimental methodology and performance overhead analysis. Section 7 presents guidelines for the use of performance evaluation tools under different use-cases, and finally Section 8 discusses our conclusions.

2 RELATED WORK: GC & JVM ANALYSIS

JVM implementations support the logging of GC statistics by statically or dynamically turning on JVM flags. The statistics typically concern object allocations, usage of heap memory spaces, stop-the-world (STW) pauses, and total execution times for each GC invocation. STW pauses occur when all application threads are blocked during GC. Post-processing or runtime analysis of logs can be performed by tools such as [10, 9, 11, 20]. Lengauer et al. [17] presents an in-depth analysis of the GC behavior of Java benchmarks including DaCapo using G1 GC in the HotSpot JVM. Automated processing of logs can aid identification and solution of GC related performance problems, such as increasing the maximum heap size, changing the ratio of younger to older generation heap sizes, and changing heuristics for moving objects into different heap spaces. Dynamic dashboard visualizations are typically used to present information concerning the relative health and efficiency of applications in production environments.

However, GC log analysis cannot help in identifying or resolving which application code is the cause of high memory allocation and/or leaks. The JVM can provide a heap dump snapshot of the current live objects at relatively high cost. Such dumps may help developers to determine why GC execution and pause times might be higher than expected. Unfortunately heap dumps lack information on the allocation site of an object, nor can a dump help determine which thread allocated an object. Object deallocations can only be detected by comparing two subsequent dumps and finding that an object was removed. This is difficult as objects often change their heap location during GC where objects move between memory generation regions, and in heap compactation. Object allocation profiling and recording of every object using standard JVMs are typically based on expensive [24] combinations of bytecode based instrumentation, and stack traces. Consequently, many tools only record allocations above minimum sizes, and/or every n-th allocation in order to reduce overhead.

AntTracks [16] is a customized HotSpot based JVM that instruments object allocation, movement and deallocation with a low logging overhead of less than 4.68%. The recorded events are sufficiently detailed to enable the heap’s state to be reconstructed offline, for the beginning and end of every GC cycle by incrementally applying the effect of events described in trace files. Thread local allocation buffers (TLABs) are used to store event traces, with data compression to minimise trace collection and storage overheads.

Cao et al. [6] evaluated the power and performance requirements of managed runtime workload execution. Their work considered single-ISA heterogeneous asymmetric multicore processor (AMP) hardware with processor cores that are optimised for different power-performance tradeoffs. The main findings of interest are that GC accounts for 10% of processor execution cycles on average, and nearly 40% for lusearch from the DaCapo benchmarks.

Sartor et al. [21] presented techniques to analyze the scalability of managed language applications with speedup stacks. Lightweight OS kernel modules were used to monitor an application and its VM service threads scheduling behavior with an overhead of up to 1.15% but, typically under 1%. They perform experiments on one node of an Intel Xeon E5-2650L server consisting of 2 sockets, each with 20MB shared LLC, 8 physical cores, with hyper thread-enabled giving 16 logical cores on each socket, and a 64-bit 3.2.37 Linux kernel. Sufficient logical cores are present to ensure that the OS does not need to schedule out a thread other than for synchronization or I/O. A problem with this approach is that it is not expected to be portable across processor ISA architectures, and it may require recompilation for each new kernel release. For their experiments, they used JikesRVM [4] with the DaCapo lusearch, pmdb, sunflow and xalan benchmarks. JikesRVM was modified to record the JVM application, compilation and garbage collection thread type process identifiers (PID) to enable their kernel modules to attribute microarchitectural performance counter measurements to specific JVM thread types. The speedup stack visualization helps developers to determine if optimization efforts that focus on application, or JVM runtime services are likely to improve performance. Our bcc–java tool development was motivated by [21].

Hofer et al. [14] modified OpenJDK8u45 to observe and trace events related to lock contention with a mean runtime overhead of 7.8%. The call-stacks of a thread blocking on a lock, and the thread holding the lock causing the blocking were recorded. They also presented a tool for users to identify locking bottlenecks in their code by analysing the traced events. They identified that many call-stacks that use locks are actually identical, therefore it is only necessary to maintain a hash set of known call-stacks.

The BCC ftrace tool performs a similar task to [14], except that it uses eBPF’s OS level tracing of kernel events associated with thread scheduling to determine: (1) the call-stack of a blocked thread, (2) its blocked time duration, and (3) the call-stack of the waking thread, which causes the scheduling state of the blocked thread to change to runnable. The call-stack of the blocking thread does not change whilst it is off-core and blocked. Hence, its call-stack need only be recorded when it meets some selection criteria, such as belonging to a process that we are interested in, and if its blocking duration is within a specified range. This enables efficient and selective tracing of call-stack blocking and wakeup behavior due to thread interactions.

3 FLAMEGRAPH VISUALIZATIONS

Brendan Gregg [13] has shown that scalable vector graphic (SVG) visualizations of flamegraphs, see Figure 1, can aid identification of performance critical code from sampled stack traces. Stack traces can be related to processing core time, or to the change in a quantity, such as last level cache misses, from a hardware counter. In this way it is possible to produce flamegraphs that characterize the microarchitectural behavior of an application’s execution. SVGs enable flamegraphs to be viewed, zoomed in/out, and searched for text inside web browsers. The percentage contribution of a
method is easily determined by searching for its name, whereupon each instance of the method in the graph is highlighted and its overall contribution to the total number of samples is displayed at the bottom right hand corner. This can be especially important for determining the relative contribution of methods having many different call-sites in an application.

In a flamegraph, any stack traces having identical callers are merged, then any non-identical child callee nodes in the collected traces appear as a new control flow path. Control flow paths are visualized by presenting the unique names of methods inside rectangular blocks that are typically organized lexicographically in order from left to right. Divergence in a control flow path is indicated by more than one rectangular block being stacked on top of another, such as for the blocks stacked on top of GangWorker::loop in our Figure 1. The topmost frame in a flamegraph is the method that was executing when a call-stack was sampled. The topmost methods will be identified correctly as long as any sampling skid \(^1\) does not cause an incorrect method to be attributed. Note how the use of different colours can be used to distinguish between inlined Java, kernel, JIT compiled Java, C++, and native/library code. Interpreted Java methods are only labelled as Interpreter, and cannot be identified with \texttt{perf}/\texttt{perf-map-agent}. However, call-stacks and flamegraphs produced with \texttt{async-profiler} can identify methods that undergo interpretation. It is a fairly trivial matter, using options for \texttt{async-profiler} and \texttt{perf} to generate flamegraphs where thread PIDs are also captured with call-stacks, this can sometimes be beneficial, but it can quickly become confusing if an application contains many threads.

\textit{Identifying Inlined Methods}. In Figure 1, an inlined method appears as a teal/blue frame above a green Java JIT compiled method, further inlined methods appear as stacks of teal/blue frames. Inlining information accuracy is improved through the use of \texttt{-XX:+DebgugNonSafePoints}. Note that flamegraph columns where a teal/blue rectangle/method is directly below green, indicate a place where the JIT compiled and inlined code has called a method that was chosen not to be inlined. Note that on-core flamegraphs, only demonstrate stacks for threads that were executing when sampling of call stacks occurred. They neglect the importance of off-core time when threads are blocked.

\textit{Heap/Allocation Flamegraphs}. Figure 2 is an heap allocation flamegraph created using \texttt{async-profiler}, here the width of a call stack is proportional to the amount of heap memory allocation. The topmost call-stack element represents the allocated data type, i.e. an array, such as char [ ], or an object such as \texttt{StandardFilter}. The profiler features TLAB-driven sampling that relies on callbacks (see \texttt{AllocTracer} in the OpenJDK/HotSpot sources) to receive notifications: i), when an object is allocated inside a newly created TLAB (annotated as \texttt{INSIDE} or aqua in Figure 2), and ii), when an object is allocated on a slow path outside a TLAB (annotated as \texttt{OUTSIDE} or in orange in Figure 2), for example when its size exceeds that of the TLAB. This means not every allocation is counted, but only allocations every N kB, where N is the average size of TLAB. This makes heap sampling very cheap and suitable for production, as in practice it will often reflect the top allocation sources, but the collected data may be incomplete.

\textit{Offwaketime Flamegraphs}. The blocking latency associated with off-cpu time can be analysed with \texttt{offwaketime} flamegraphs. Figure 3 presents cropped illustrative output of the BCC/eBPF tool \texttt{offwaketime} for the \texttt{avrora} benchmark for blocking of thread node-0. The bottom stacks, colored blue, are off-core stacks that indicate the sequence of method calls leading to a thread blocking, and these are read from the bottom up leading towards the point where they blocked, that is separated from the waking thread stack by a grey block that is labelled \texttt{--}. The waking thread stacks, colored aqua, have their call-stack frames listed in reverse order, and are read from the top down, from application code, down to the method that caused the blocked thread’s scheduling state to change. The tool \texttt{offwaketime} achieves similar information to \cite{14} for determining the blocking and wakeup behavior without requiring a modified JVM. A current limitation is that one wakeup stack may not be sufficient to explain all the sources of blocking latency. For example, when the “waker-thread” was itself blocked on another thread, potentially leading to a chain of blocking and wakeup stacks.

4 JAVA BASED PROFILING

Sampling profilers repeatedly collect snapshots of the call-stack of functions/methods, that describe the on-core execution context of a thread running on a processing core, at the sampled moments in time. The thread execution time of specific methods is statistically related to the number of times the method appears in all stack traces - as long as the samples are collected fairly, and that all points in a program run are selected randomly to remain statistically independent. If many samples are collected over time, and the code stack-traces are highly correlated, then we can safely assume that threads in a program are spending most of their time executing code described by the most heavily correlated call-stack. Unfortunately, profilers that directly rely on the \textit{Java Virtual Machine Tool}
<table>
<thead>
<tr>
<th>Tool</th>
<th>async-profiler</th>
<th>perf</th>
<th>bcc-java</th>
<th>offwaketime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native applications</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JIT compiled code support</td>
<td>Automatic</td>
<td>requires perf-map-agent</td>
<td>Not appropriate</td>
<td>requires perf-map-agent</td>
</tr>
<tr>
<td>JVM Restrictions</td>
<td>OpenJDK/HotSpot AsyncGetCallTrace TLAB overflow callbacks /DK 7u40</td>
<td>JDK 8u60 onwards</td>
<td>No, but for thread names JDK9 needed</td>
<td>JDK8u60 onwards</td>
</tr>
<tr>
<td>Interpreted methods</td>
<td>Identified</td>
<td>Seen as Interpreter</td>
<td>Not appropriate</td>
<td>Seen as Interpreter</td>
</tr>
<tr>
<td>Identify inline Java methods</td>
<td>No</td>
<td></td>
<td>-XX+DebugNonSafePoints</td>
<td>No</td>
</tr>
<tr>
<td>JDK -XX Flags +PreserveFramePointer +DebugNonSafePoints</td>
<td>None</td>
<td>Yes</td>
<td>Required</td>
<td>No, Yes</td>
</tr>
<tr>
<td>Methodology</td>
<td>Sampled call-stack</td>
<td>Sampled call-stack</td>
<td>eBPF tracing instrumentation</td>
<td>eBPF tracing blocked time captures call-stacks</td>
</tr>
<tr>
<td>PC stack sampling skid</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Heap profiling/overhead</td>
<td>TLAB based, low overhead</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Identify hot-methods</td>
<td>Some JVM stubs/intrinsics are invisible</td>
<td>Full-stack, stubs &amp; intrinsics are visible</td>
<td>Thread-based (No)</td>
<td>Blocked-time (No)</td>
</tr>
<tr>
<td>Microarchitecture counters</td>
<td>Sample call-stacks</td>
<td>Sample call-stacks</td>
<td>Measures changes associated with threads</td>
<td>Measures off-core time</td>
</tr>
<tr>
<td>Flamegraph visualisation</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Relate blocked thread call-stacks to wakeup call-stacks</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes - with selective filtering on blocking time</td>
</tr>
</tbody>
</table>

**Table 1: Summary of features and functionality of profiling and tracing tools used in experiments.**

---

Figure 2: Annotated flamegraph, async-profiler for lusearch benchmark for heap memory allocations.

Figure 3: Cropped, annotated offwaketime flamegraph (restricted to thread node-0 blocking) for avrora benchmark.

Interface (JVTMI) suffer from the problem of safe-point bias [18]. The sampled data is skewed towards the times when threads have voluntarily yielded the processing core at safe-points where the JVTMI’s GetStackTrace can directly obtain call-stacks. One might incorrectly come to the conclusion that the actual hot code should lie somewhere between the sampled call stack of the “hot” safe-point and the potential call-stack of the immediately previous safe-point. Yet, this is not the case because there are significant, variable and lengthy delays between when an JVTMI agent signals an JVM to sample a stack trace and the operation occurring. The JVMTI request is first added to the VM thread’s work queue, and the VM thread must process any prior pending tasks first. Figure 1 demonstrates significant VM_Thread activity (20% of call stacks) related to GC.

Mytkowicz et al. demonstrated, in [18] that four different commercial and open-source profilers often produced demonstrably different results for the percentage overhead of, and the actual hottest method. The overhead of profiling varied between 1.1x and 1.5x for the different profilers on sequential benchmarks chosen from DaCapo, and different profilers caused significant perturbation...
Figure 4: Normalized execution time overheads for unprofiled (normal) bcc-java (BCC), async-profiler on-core with (ASY) and heap allocation with (ALL), perf (PMAP) summarised for lusearch, pmd, sunflow, xalan benchmarks. g.mean refer to all 11 benchmarks studied.

The honest-profiler [3] and async-profiler [2, 19] are recent examples that have extended and improved the proof of concept ideas of [18]. We do not discuss the honest-profiler due to space limitations. The async-profiler employs a hybrid approach for its stack trace collection based on i), AsyncGetCallTrace to obtain information on a thread’s Java frames, without the requirement to be at a safe-point, and ii), perf to provide information on native code and kernel frames. The perf_event API is used to configure stack samples to be placed into a memory buffer, and a signal is delivered when a sample has been taken. Its signal handler then calls AsyncGetCallTrace to capture the Java stack that it then merges with the perf stack trace of native and kernel information.

The main benefits of this approach over a purely perf/eBPF approach is that it can be used to profile applications running on older JVM releases that do not support -XX:+PreserveFramepointer that is necessary for OS tool based stack walking of Java thread stacks. The main pros and cons of async-profiler are: – AsyncGetCallTrace is an internal Oracle/OpenJDK API that is not guaranteed to be available in other JVM implementations. – Optimised compiler inserted stub code that enables efficient implementation of compiler intrinsics, and performance critical JVM operations such as crc32 and System.arraycopy will generally be invisible to AsyncGetCallTrace because the JVM does not record the necessary metadata for many such stubs.

---

Execution overhead of this flag was less than 4% in our experiments.
Off-core thread stacks that are blocked or sleeping are not collected by AsyncGetCallTrace. AsyncGetCallTrace will produce some failed stack traces where it is unable to safely walk a stack without causing an invalid memory access. There were very few failed stack traces in our experiments, perf$ also suffers from this unavoidable problem.

Does not require generating a map file to map Java code addresses to method names, if one uses perf_event directly then one traditionally uses perf-map-agent for this purpose.

Identifies the methods in interpreter frames, perf-map-agent merely identifies that an interpreter frame is present.

Allocation profiling can be performed to identify the sites where the largest amount of heap memory is allocated. In our experiments the overhead of allocation profiling is always below 2.8%. Allocation profiling does not impact on escape analysis or JIT optimizations.

The async-profileler tool can collect stack traces from Java methods, native calls, JVM code and kernel functions (only possible in processing core profiling mode). Both async-profileler and perf approaches support reading performance counters supported by perf_events such as: last-level cache/data TLB load misses, L1-dcache misses, branch misses, page faults, and instructions retired. The perf approach collects callstacks and has access to perf_events, and uses the perf-map-agent JVMTI agent to generate a mapping of code addresses to Java symbol names. The main benefits of the perf-based approach over async-profileler are that: (1) the JVM flag -XX: +DebugNonSafepoints enables inlining decisions to be accurately visualized in a flamegraph, and that (2) JVM generated stubs, and performance critical operations (e.g. JVM intrinsics), can be observed that are invisible to async-profileler.

No commercial features are required by async-profileler, as it is completely based on open-source technologies and works with OpenJDK from version 7u40 onwards, where the TLAB callbacks appeared. Note, the heap allocation functionality of async-profileler requires libjvm.so JVM debug symbols.

Stack Fragment Sampling (SFS). SFS, as described in [15], has less overhead than AsyncGetCallTrace and JVMTI based methods, this is achieved using a modified HotSpot JVM. SFS typically occurs only when a thread is active and on-core. A thread is paused to copy a fragment of its full stack to a buffer. Copied fragments are asynchronously retrieved from the buffer and decoded to full stack traces. Redundant samples are not captured giving reduced storage overheads.

Their work is of low overhead, but they encounter significant problems when enabling the collection of information on waiting periods, i.e. when a thread is blocked for I/O or to acquire a lock. The number of lost samples increases from a maximum of 2% for pmd, to more than 50% for the benchmarks actors, avrora, and scalabletest in their experiments. Further, SFS overheadal approaches are likely to be lower on their quad-core experimental setup in comparison to our 8 core node. The performance slowdown of their techniques was approximately 1.15x (i.e. 15% slower) for avrora when waiting time was not sampled for a 10kHz sampling rate. They found i) that their heuristic rules to enable stack walking work for more than 90% of sampled stacks, even when execution is in native or VM code, and ii), that a fragment size of 32KB can obtain complete stack traces.

5 THE BCC-JAVA TOOL

The Berkeley Packet Filter (BPF) [22] enabled instrumentation code to be attached to a network socket by injecting bytecode (describing an BPF program) from userspace to the kernel. JIT compilation and verification checks are performed to ensure no security or kernel crashes can occur. The development of extended BPF (eBPF) has (i) improved JIT compilation, (ii) added many more probes to Linux, and (iii) enabled user-level statically defined trace probe points for application programs. Probes are points where dynamic instrumentation can be attached. Each probe has a specific set of variables that are used to describe the probe’s events. The instrumented code can perform different processing based on the values of the variables of a probe.

We use the BCC toolkit that provides a set of Python scripting examples demonstrating how to, i), launch BCC tools, ii), exploit BPF features for instrumentation, and iii), how to efficiently perform I/O to obtain profiling/tracing results.

The main capabilities of BCC, and eBPF are summarized in [12]. Instrumented code can read hardware performance counters, and update arbitrary data structures to collect and process information. A Maps data abstraction enables information to be processed and aggregated in kernel, before sharing with user-space when required.

We have developed a low-overhead tracing tool that records microarchitectural and execution time characteristics associated with the multi-threaded execution of Java programs. The tool currently records a thread’s name, its start/exit time, cycles, instructions, and time spent on-core in execution. We additionally record the number of scheduling quanta given to each thread, the off-core waiting time, and the total number of sys_futex system calls. The initial JVM process is created by executing the JDK java command; hereafter referred to as the java-process. Our tool uses instrumentation of: - java-process creation and termination by instrumenting kernel methods for sched_process_exec, sched_process_fork and sched_process_exit. The PID of the initial java-process that is created is stored in a hashmap. - The creation of new threads that are children of the initial PID are tracked using a kernel probe attached to the sys_clone method. - Kernel probes are attached to the entry and exit of the sys_futex system call. We record for each thread, the number of futex calls, the total elapsed time whilst blocked, and performance counter values when a thread is blocked/unblocked.

Thread scheduling, namely sched_swi tch is instrumented to collect time and performance counter information concerning the allocation of a thread to a processing core. The counter values and elapsed time are updated at the end of a thread’s quantum, or when the thread enters the sys_futex system call.

On termination of the java-process, the tool stores per-thread details of accumulated performance counter values, total wait time, total calls to sys_futex, and elapsed time. OpenJDK9 uses a default thread naming scheme that allows to distinguish between different types of threads, i.e. application and service threads related to JIT compilation, GC, and other JVM services. Thus, measurements are linked to specific JVM subsystems.
6 EXPERIMENTS – TOOLS OVERHEADS

The experimental objective is to quantify the execution time overheads of perf, async-profiler and bcc-java tools using the established set of DaCapo [5] JVM performance evaluation benchmarks. The overheads of running the different performance tools are normalized against the execution times of ELEVEN unmodified (avrora, fop, h2, jython, luindex, luserase, pmd, sunflow, tradebeans, tradesoap, xalan) DaCapo-bach-9.12 benchmarks that execute successfully under OpenJDK9 with G1 GC using default application problem and heap sizes.

The experimental machine is a two-socket Intel Xeon E5-2690, with 8 physical cores per socket (hyper-threading is disabled), 20MB LLC cache, and a total of 384GB DRAM main memory, with Linux kernel 4.13.0-36-generic. All cores are set to 2.9GHz frequency via a user-space governor. No other users are present on the system, a single ssh connection is used to initiate and monitor experiments that only access local disks. NUMA memory effects are avoided by executing applications only on cores belonging to one socket, as in [21]. We used bcc-tools version 0.5.0-1, and OpenJDK (build 9-internal-0-2016-04-14-19524-6.builddd.src, mixed mode).

Each benchmark is executed for 20 iterations and the 19th iteration’s time is recorded. This is repeated 30 times in order to avoid variability due to JIT recompilation and startup costs. The 19th iteration execution time measurements are normalized to the average of each benchmark under normal execution without profiling over its 30 runs. Figure 4 depicts the median execution times, in the form of box plots, normalized to average execution time without sampling. On each box, the central mark denotes the median, and the bottom and top edges indicate the 25th and 75th percentiles, respectively. The box spans the interquartile range of the sampled data. The whiskers extend to the most extreme data points not considered outliers, which are not shown. However, the whiskers extend to up to 1.5x of the interquartile range below and above the bottom and top edges, respectively. This corresponds to approximately ±2.7σ or 99.3% coverage if the data were normally distributed. The g.mean labels in Figure 4 show the geometric means over all ELEVEN benchmarks; e.g., g.mean-ASY-100Hz is the geometric mean over all benchmarks of async-profiler on-core profiling with 100Hz sampling. These are accompanied by error bars that delimit 95% confidence intervals.

The experimental configurations are: (1) baseline (no profiling tools used), (2) async-profiler with ALL allocation and ASY processing core profiling, (3) perf with perf-map-agent (notated as PMAP), and (4) bcc-java based tracing (notated as BCC) where we measure the instructions and cycles executed along with accumulated counts of execution/waiting time and the start and end times of individual threads. For on-core async-profiler and perf we used 100Hz, 1kHz and 4kHz sampling frequencies. Note, 1kHz perf sampling was not feasible as kernel warning messages appeared concerning interrupts and frequency throttling.

For clarity, Figure 4 only plots sampling frequencies of 100Hz and 1kHz for a small set of individual benchmarks (luserase, pmd, sunflow and xalan), whilst geometric plots refer to execution of all ELEVEN benchmarks with 100Hz and 4kHz sampling, in this way we can see the relatively higher overheads of async-profiler (1.26x ASY-4kHz) in comparison to perf (1.16x PMAP-4kHz) at higher sampling rates (perf overhead is substantially less than async-profiler for 4kHz sample rates for all benchmarks other than avrora). This is despite the unusually high overhead of perf on avrora that is more than 2x even at a low 100Hz. However on a laptop class machine with an i7-7500U processor having 2 physical processing cores (4-hyperthreaded) @2.7GHz, the overhead of the same experiment was 3.8%. We cannot definitively explain why there is a significant overhead even for 100Hz sampling, but we suspect that processor chipset issues, and kernel version issues such as default perf buffer sizes may be relevant.

Our bcc-java tool has very low-overhead apart from avrora where its average overhead is 1.37x. bcc-java text output enabled us to determine that the Java application threads were spending circa 80% of their total execution time blocked off-core, largely because of repeated calls to sys_futex when attempting to acquire locks. bcc-java has relatively high overhead on avrora because instrumentation measuring time, cycles and instructions executed, along with the code to update the in-kernel eBPF maps, occurs with relatively high frequency, that we calculated from bcc-java tracing to occur on average every 9863 cycles and 12658 instructions executed, by examining bcc-java tracing information for one iteration of one application thread on avrora.

7 PERFORMANCE USE-CASE GUIDELINES

The next paragraphs present use-cases and the suggested guidelines for how to identify them.

A) Identifying the call-stack context, load-imbalance, and functions where significant execution time is spent. Browser views of on-core flamegraphs produced via perf, or using async-profiler are appropriate. The method with the most significant execution time equates to finding the top-most widest call-stack tower. Note that some functions may appear in many different call-stacks, because they are called from many different call-sites, and this may cumulatively add up to be a significant proportion of the total number of stack-samples, and hence execution time. Searching for a function name in the browser view of an SVG will determine the total % appearances of the function in all call-stack samples. perf can see many JVM intrinsics, compiler stubs and inlining decisions that are mostly invisible to async-profiler. Load-imbalance can be identified by using flamegraphs that propagate thread PIDs to the call-stack. Unequal width call-stacks concerning different thread PIDs will signify unequal on-core load-imbalance across threads. Load-imbalance can also be measured by examining thread start/stop and blocked execution times that is traced by the bcc-java tool.

B) Identifying if GC time, and memory allocation issues are significant Core-profiled flamegraphs can be used to identify if GC related methods are significant, whereas allocation based flamegraphs with async-profiler can determine the source code methods where significant memory allocation occurs, and whether allocations were from inside or outside of a TLAB. Manually restructuring the code to reduce the volume of memory allocated, and applying standard GC log file analysis tools to change and optimize specific GC algorithm parameters are recommended.

C) Identifying JIT compilation method inlining decisions that may be critical for performance. We recommend, using perf, with -XX:+DebugNonSafePoints enabled. Flamegraph columns where
a teal/blue rectangle is placed directly below a green rectangle indicates a call site where JIT compiled and inlined code has called a method that was chosen not to be inlined. JIT compilation logging can be selectively enabled for the affected methods. A tool such as JITWatch [1] may help to determine the reasons why inlining failed. Careful refactoring of large methods into multiple smaller ones may enable inlining decisions that were previously precluded.

D) Identifying the causes of blocked thread execution states, for example, lock-contention, I/O, and a lack of processing resources. The bcc-java tool can be used to identify on a per-thread basis if a thread spends a significant amount of time blocked, and, also if it relates to sys_futex lock acquisition/release. The eBPF/BCC tool offwaketime can be used to identify the specific call-stacks that contribute to long blocking times, and also the call stacks that cause the blocked threads to be woken up in the kernel. This achieves similar functionality to [14] but without any JVM modifications. Note, the offwaketime tool uses tracing so the actual blocked execution time is measured. Command line arguments can be used to filter out very short and extremely long thread blocking times such as when a server is lightly loaded or inactive.

8 CONCLUSIONS

This paper examines the performance analysis capabilities of async-profiler, perf/perf-map-agent and the potential of eBPF-based tools such as our bcc-java, and offwaketime. Flamegraph visualizations of memory allocation (GC) and processing core profiling can easily show which Java methods and native or OS code are significant, and that even JIT compilation inlining decisions can be captured. Sampling at 1kHz (1000 samples per second) offers reasonable overhead and 100Hz sampling rates are suitable for production use, except for avro/a with perf. Although not presented in the paper, the async-profiler and the perf/perf-map-agent approaches can both be used to attribute microarchitectural performance counter events to flamegraphs. However, PC sampling skid (affecting both async-profiler and the perf/perf-map-agent) will vary depending on the specific performance counter event and processor implementation. We have found that sampling-based profiler flamegraph visualizations provide a useful tool to investigate and to characterize Java application and GC performance. In contrast, our tool bcc-java does not suffer from the PC-sampling skid problem as it is a tracing based tool. The overhead of bcc-java is small (geomean 3.6%) despite capturing microarchitectural performance counters on thread scheduling activities.

REFERENCES