Cloud-Scale Java Profiling at Alibaba

Fangxi Yin, Denghui Dong, Chuansheng Lu, Tongbao Zhang, Sanhong Li, Jianmei Guo and Kingsum Chow
Alibaba Group, Hangzhou, China
{fangxi.yfx,denghui.ddh,chuansheng.lcs,tongbao.ztb,sanhong.lsh,jianmei.gjm,kingsum.kc}@alibaba-inc.com

ABSTRACT

On the 2017 Double 11 Global Shopping Festival, Alibaba's cloud platform achieved total sales of more than 25 billion dollars and supported peak volumes of 325,000 transactions and 256,000 payments per second. Most of the cloud-based e-commerce transactions were processed by hundreds of thousands of Java applications with above a billion lines of code. It is challenging to achieve comprehensive and efficient performance profiling for large-scale, cloud-based Java applications in production. We developed ZProfiler, a fine-grained, low-overhead Java performance profiler. ZProfiler allows developers to load a profiling agent on the fly without restarting Java virtual machines, and its profiling information also facilitates code warmup. ZProfiler is developed based on Alibaba JDK (AJDK), a customized version of OpenJDK, and it has been rolled out to Alibaba's cloud platform to support large-scale performance tuning for online critical business.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software performance**;

KEYWORDS

Java performance, cloud, profiling, overhead, code warmup

ACM Reference Format:
Fangxi Yin, Denghui Dong, Chuansheng Lu, Tongbao Zhang, Sanhong Li, Jianmei Guo and Kingsum Chow. 2018. Cloud-Scale Java Profiling at Alibaba. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3185768.3186295

1 INTRODUCTION

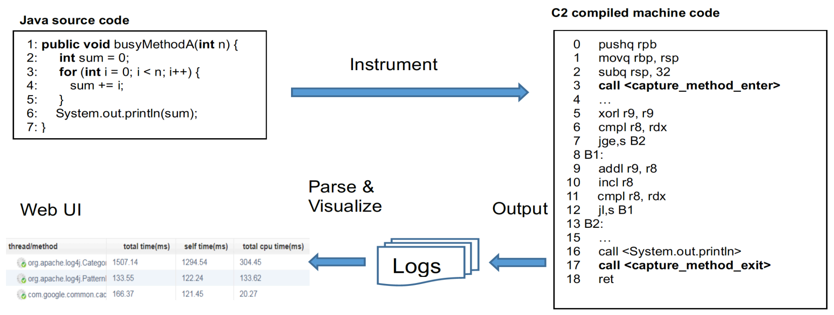
Performance profiling helps developers to diagnose performance issues and identify optimization opportunities. At Alibaba, the requirements for Java performance optimization come mainly from three aspects: First, how to execute Java code efficiently (e.g., using less time to execute a certain method)? Second, how to tune the behavior of *garbage collector* (GC) to minimize the time of *Stop-The-World* (STW)? Third, how to ensure the peak performance of Java applications for the bursty traffic over a short time period.

It is challenging to achieve comprehensive and efficient performance profiling for Java applications in production. Firstly, many existing profiling methods and tools [3], such as VisualVM [4], Dynatrace[2], and Java Interactive Profiler [6], rely on *bytecode instrumentation* to trace the execution time of each method. They usually suffer from non-negligible overheads caused by the measurement of additional code. Also, changing the size of the code changes the optimizing decisions made by the JIT. Secondly, it is usually hard for developers to diagnose the issues relevant to GC only through the straightforward analysis of GC logs [5]. For example, when a Java application is experiencing a long GC pause, it is not reliable to depend on GC logs to diagnose issues. Thirdly, a peak of 325,000 transactions per second is a huge challenge for the runtime performance of any Java program. The bursty traffic requires that the application must remain a peak performance at that time. To meet this requirement, a well-established code warmup is often expected [1].

Further, there are more particular challenges for Java profiling at cloud-scale environment. First, the profiling activites takes place frequently because of the massive quantity of JVM instances. So the profiling tools' troublshooting efficiency is critical and an intuitive profiling tool is always favored over. Second, the overhead requirement of profiling is stricter. At cloud-scale environment, even one percent overhead would increase cost significantly. To compensate the overhead, a common way is to purchase more machines.

To address the above challenges, we developed a fine-grained, low-overhead Java performance profiler, called ZProfiler, based on Alibaba JDK (AJDK). AJDK supports all the Java applications developed at Alibaba and affiliated companies (e.g., Alipay, CaiNiao). Java developers can easily load the profiling agent *on the fly* without restarting the JVM.

In summary, we make the following contributions. Firstly, different from typical Java profilers working at the levels of bytecode or JVM Tool Interface (JVMTI), ZProfiler traces the execution time and memory usage at the level of each compiled method. Secondly, ZProfiler provides fine-grained information of GC behavior with much lower overheads, which used to minimize the STW cost. Thirdly, ZProfiler has the built-in code cache analysis ability to help users to warmup code by ahead-of-time compilation, which makes application ready for bursty traffic handling. Finally, ZProfiler has been rolled out to Alibaba's cloud platform to support large-scale performance tuning for online critical business.



```
public void busyMethodA(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
      sum += i;
    }
    System.out.println(sum);
}
```

```
    pushq rpb
    movq rbp, rsp
    subq rsp, 32
    call <capture_method_enter>
    …
    xorl r9, r9
    cmpl r8, rdx
    jge,s B2
B1:
    addl r9, r8
    incl r8
    cmpl r8, rdx
    jl,s B1
B2:
    …
    call <System.out.println>
    call <capture_method_exit>
    ret
```

thread/method	total time(ms)	self time(ms)	total cpu time(ms)
org.apache.log4j.Categor	1507.14	1294.54	304.45
org.apache.log4j.Patternl	133.55	122.24	133.62
com.google.common.cac	166.37	121.45	20.27

Figure 1: Method Tracing of ZProfiler in C2 compiler

2 APPROACH AND IMPLEMENTATION

2.1 Method Tracing

Instead of instrumenting at the level of Java bytecode, ZProfiler modifies the Hotspot VM, including interpreter and C1/C2 compiler, to trace entry/exit of each method. Figure 1 illustrates how ZProfiler enables method tracing in C2 compiler.

When C2 JIT compiler compiles the application's method, ZProfiler adds a few machine code for every method, that is, adding capture instructions when compiler encounters method entry or exit. For every method enter or exit, the captured runtime functions record the onsite information, including the current CPU/wall-clock time, method information, and the allocated memory of each thread. When the compiled method is executed, the onsite information would be recorded in the profiling files. When profiling is switched off, ZProfiler uploads the profiling files, parses the profiling data, and visualizes them in a Web UI.

ZProfiler's method tracing does not impact JIT inline decisions because its instrumentation is implemented at VM level instead of bytecode level. Moreover, we added only a few extra instructions for non-inline methods, so that the profiling overheads are reduced as far as possible. Most of APM tools [2] often instrument applications at the byte code level. The byte code instrumentation often impacts JIT inline decision thereby affects the performance. A common case is, the getter/setter functions which are widely used can not be inlined and the performance loss is non-negligible.

2.2 GC Tuning

ZProfiler modifies the implementation of GC in the Hotspot VM and records the information of time consumption at each of GC root processing phases. Developers can use the recorded information to determine which phase gives rise to a longer GC pause. For example, if one acquires the information that the StringTable root processing takes a long time during a GC pause, then the developer can check String.intern use in the implementation and get clues to fix this problem. Moreover, ZProfiler provides mxbeans for monitoring the fine-grained information.

2.3 Code Warmup

ZProfiler extended jcmd utility of OpenJDK to support code cache dump, which helps them to understand if the Java application has been fully warmed up and ready for processing bursty traffic. The code cache dump provides the summary information of code cache usage, the memory usage of compiled method by class loader, and the distribution of compiled methods at each level (from level 0 to level 4) of code cache. AJDK provides public APIs to trigger compilation explicitly at *ahead-of-time*. For example, if one finds many methods get compiled at Level 2 in code cache, the developer can choose to compile some of them in advance according to business logic.

3 CONCLUSION

This paper presents our Java performance profiler ZProfiler that supports hundreds of thousands of Java applications deployed in Alibaba's cloud platform. Compared to existing Java profilers, ZProfiler instruments each method with a few machine code and traces the resource usage of each compiled method with low overheads. Simultaneously, ZProfiler provides a fine-grained view on GC behavior and code cache, which facilitates comprehensive performance diagnosis and efficient fixing of performance bugs. Furthermore, ZProfiler allows Java developers to easily load the profiling agent on the fly without restarting the JVM or code de-optimization, which is critical to guarantee the reliability and serviceability of online Java applications in production.

REFERENCES

[1] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages* 1 (OOPSLA) (2017), 52:1–52:27.
[2] Dynatrace. 2018. Deliver unrivaled digital experiences. (2018). https://www.dynatrace.com/
[3] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *Proceedings of PLDI*. 187–197.
[4] Oracle. 2014. Java VisualVM. (2014). https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/profiler.html
[5] Sun and IBM. 2002. Tool Report: GCViewer. (2002). http://www.javaperformancetuning.com/tools/gcviewer
[6] Andrew Wilcox. 2006. JIP - The Java Interactive Profiler. (2006). http://jiprof.sourceforge.net/