ATLANTIS PRESS
Advances in Computer Science Research, volume 62
2nd Joint International Information Technology, Mechanical and Electronic Engineering Conference (JIMEC 2017)

YA-JOP.Yet Another Java Object Profiler On Art Virtual Machine

Fei Wang, Xiaohua Shi, Chao Li

State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing, China , waananikaho@gmail.com

Keywords: java object profiler; embedded systems; Android; managed languages; object lifetime information.

Abstract

In this paper, we design and implement an efficient and low overhead Java object profiler called YA-JOP on Android 6.0 and its ART virtual machine, which uses an AOT(ahead-of-time) compiler. YA-JOP records the allocation site, the class information, the object size, the birth time and death time, the last access time and the access regular pattern for every Java object. The data profiled can help the developers to detect memory leaks, find reusable objects, implement optimizations like pretenuring, etc. The profiler proposed in this paper has reasonable execution time overhead, imposes no overhead on the Java heap and does not modify any existing key data structure of the ART Virtual Machine, including the object layouts, class layouts and any others.

1 Introduction

In recent years embedded devices have become more and more popular. Mobile phones and tablets, which often use Android system, are the most popular electronic devices. The latest Android operating system uses the ART virtual machine to execute Java programs and Android applications. Managed languages such as Java, C#, Python and PHP have garbage collection to manage all the objects at runtime, which reduce the memory-related failures and improve the efficiency of the usage of memory heaps. But object-oriented Android applications still have memory problems which are difficult to be located. For example, developers may keep references to objects but don't use them, or create and destroy large number objects of the same type. Sometimes, we need a tool to know the lifetime of every object to find ways to optimize our programs.

Some researchers have tried to profile memory behaviour of programs on the ART virtual machine[15]. Developers can also use HPROF to generate a snapshot of Java heaps for Android applications [5]. However, these tools don't have the ability to provide the life cycle and access information for Java objects. Other researchers have implemented tools to profile Android system and applications in IO overhead[7], privacy leak[1], application behaviour[14], resource leak[19], energy consumption[3], etc. There are many researches and tools in other platform to track information of Java objects[2,4,6,8,9,10,13,16,17]. But these tools are not designed for the ART virtual machine and not available in Android platform. Sometimes they incur both high space and time overhead. Therefore, how to profile and optimize programs in Android is remaining a challenge.

In this paper, we present YA-JOP, an efficient Java object profiler on the ART virtual machine. Object events (e.g., object allocation, access and death) are tracked online and recorded into files. Unlike other Java virtual machines, ART virtual machine uses AOT(ahead-of-time) compilers. Based on the features of the ART virtual machine, we encode the object access event to a single bit in the bitmap mapping the heaps. YA-JOP adds instrumentation at object uses at compiling time before Android programs first running. The read/write barriers overheads and I/O overheads of our profiler are about 61% and 28% separately on average for EEMBC, SciMark and other workloads, which are reasonable. The profiled result provides object information like object lifetime, last access time and access frequency, which are valuable for finding memory leaks[2,16,18] in Java programs and many runtime optimization techniques, such as object tenuring[12], reusing[17] and compressing[8,11].

2 Profiling Object Information

In this section, we describe YA-JOP, a Java object profiler for ART virtual machine. We begin with an overview that describes how a user interacts with YA-JOP and the instrumentation work flow within YA-JOP. We then discuss the profiling rules, and describe how we record the object access event on the ART virtual machine.

2.1 Overview of YA-JOP

YA-JOP has an online object events tracker and an offline object events analyzer. Fig.1 describes the overall workflow for YA-JOP. A user provides a Java program to YA-JOP. The first is the original app binary to be instrumented by the modified compiler. Modified compiler inserts object access event recording codes in the compiled app. Second, the instrumented app is executed with the modified ART virtual machine runtime. Objects lifetime will be recorded by the modified ART runtime. While app is running, all the information of Java objects specified by our profiling rules will be recorded into files. Then, we use an offline analyzer to analyze the profiled data and give the profiled result.

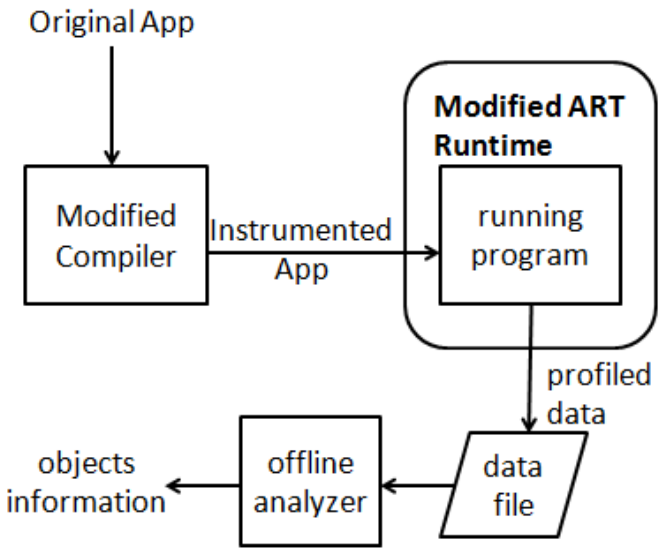

Figure 1: Overview of YA-JOP

2.2 Profiling rules

CMS (Concurrent mark-sweep garbage collector) is the most used garbage collector in the ART virtual machine. Now we only instrumented CMS garbage collector. Objects could not be moved while using CMS garbage collector. We can use a vector to present one object, as below:

< AllocS, T_0, T_d, Type, Size, Addr, T_{a0}, T_{a1}, ..., T_{ak}>

In which, T_i stands for the number of times garbage collector has been invoked. *AllocS* means that the allocation site of object *o* was *AllocS*. T_0 and T_d means that *o* was created at time T_0 and dead at time T_d. *Type* and *Size* indicate object *o*'s class type and object size. *Addr* means the initial physical address of *o* was *Addr*. T_{ai} means *o* was accessed at time T_{ai}. This vector uniquely identifies an Object for a particular program execution. For example, <0x6DA342B0, 1, 8, 0x530000D0, 32, 0x12CD73C0, 4, 5> represents that an object which had type 0x530000D0 was allocated at memory address 0x12CD73C0 from the allocation site 0x6DA342B0 in the first GC cycle (i.e., between the first GC and the second GC), and then was accessed in the fourth and fifth GC cycles. The size of this object was 32 bytes. Finally this object was dead after the eighth GC.

This vector does not exist in Java heaps. It is built from the profiled data. The profiling rules at runtime are as follows:

events	*Profiled data*
Allocating a new object	AllocS, T_0, Type, Size, Addr
Garbage collection starting	GCTIME + 1
Marking an object as dead	Addr, T_d
Accessing an object	Addr, T_{ai}
Garbage collection finish	GCFINISH

Table 1: Profiling rules.

2.3 Encoding object access event

There are researches to steal bits in object header or change the layout of the object while recording object information[2, 9,18], such as access events, allocation site. Programs in the ART virtual machine have limited resources. If we add additional fields to object, it could introduce significant overheads to the ART virtual machine. Stealing bits in objects often need using atomic instructions to guarantee the thread safety. We have tried to steal bits in the object's hash code using ldrex and strex, but this sometimes adds huge overheads and EEMBC runs 400\% slower in out tests.

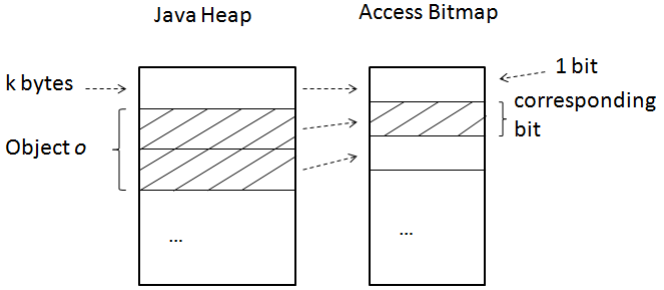

Figure 2: Bitmap Mapping.

In this paper, we present a way using a bitmap mapping Java heaps to track object access event. As in Fig.2, k bytes allocated by Java Virtual Machine correspond to one bit in the bitmap. K is the alignment of an object. When an object *o* is accessed, we set the corresponding bit to 1. If a GC occurs, the bitmap is dumped to file and all the bits are all set to 0. Only when objects mapped to the same byte are accessed in multi threads at the same time, we may lose some access information. The address of corresponding bit of object *o* can be calculated as:

$$Addr_{bit} = Addr_{bitmap} + (Addr_o - Addr_{heap}) / (8 \times kAlign) \quad (1)$$

and the location of the bit in the byte is:

$$Loc_{bit} = ((Addr_o - Addr_{heap})/ kAlign) \bmod 8 \quad (2)$$

We simply dump the bitmap and deduce objects access information from this bitmap without scanning every object in Java heap. Our recording method does not add any space overhead in Java heap and does not modify the layouts of Object. This makes our method easier to be ported to other platforms.

3 Implementation

We implement the object events tracker on top of the ART virtual machine in Android6.0. We instrument ART's concurrent mark-sweep garbage collector to track object allocation, death events. We instrument and use the Quick ahead-of-time compiler to acquire object access events.

3.1 Garbage collector instrumentation

ART virtual machine splits Java heaps into ImageSpace and AllocSpace. Java objects created by programs are allocated in AllocSpace. AllocSpace are split into ZygoteSpace, MainAllocSpace and LargeObjectSpace. ZygoteSpace is a space shared among Zygote process and all Android applications. Android apps will create objects mainly on

MainAllocSpace, of which the object alignment is 8 bytes. Primitive array objects or string objects that are larger than 3 pages can be allocated in LargeObjectSpace, of which often be used to store resources like images. We track object allocation, death events for AllocSpace and access event for MainAllocSpace.
While an object is allocated, the process will jump from compiled codes to stub codes, we simply record the value LR (link register) as the allocation site. Art method has a beginning and ending address in memory. So we can deduce the allocation function from the allocsite of an object.

3.2 Garbage collector instrumentation

ART virtual machine uses AOT compilers to compile Java programs before running them. We instrumented the Quick compiler. Before programs are actually running, modified compiler inserted the instrumentation at object access points. If an object is read or written to, we treat this object as accessed. We use bitmap to record object access events. Before every GC finishes, we dump the bitmap to a single file and clear all bits in bitmap. Similar to[8,18], we add read or write barriers when objects get accessed at the following points:

- field read/write
- array element read/write
- array length read
- method invocation
- lock acquisition and reference comparison

3.3 Offline object events analyzer

CMS is a concurrent garbage collector, objects will get accessed even garbage collector is running. We choose to dump the access bitmap when the GC finishes for tracking the complete object access events. In this situation, we will get access events for dead objects. As the access bitmap was dumped into a single file, we can simply process access events before calculating object death information in this GC cycle to avoid this problem.

4 Evaluation

4.1 Methodology

The Android source code we use is Marshmallow-6.0.0_r1. The target we build is aosp_flo-userdebug and target build type is release. The host OS environment is linux-3.16.0-71-generic-X86_64-with-Ubuntu-14.04. We perform our experiments on Nexus 7(wifi), which is a quad-core machine with a NVIDIA Tegra3(ARMv7) 1.6GHz processor and has 1GB RAM and 16GB ROM.
We use the EEMBC benchmarks, SciMark 2.0 benchmarks, CaffeineMark 3.0 benchmarks and DeltaBlue benchmark to evaluate the performance of our profiler. We execute each benchmark with a minimum possible heap size for that benchmark. Kxml and deltablue benchmarks are configured with 2MB heap, scimark.large benchmark is configured with 34MB heap, and all other benchmarks are configured with 1MB heap. We run each benchmark 5 iterations and get the average result.

4.2 Overhead

Our profiler dumps all the data into files. The only additional execution space is the bitmap which tracks object access event (Section 2.3). On ART virtual machine object's alignment is 8 bytes. The size of a bitmap is 1/64 of the corresponding Java heap. In this way, our profiling method neither add any overhead to the Java heaps nor change any key data structure of the ART virtual machine.
Unlike other Java virtual machines, the ART virtual machine uses AOT compiler. The compilation does not add any execution overhead, so all the benchmarks we ran don't need warm-up. While compiling we can do instrumentation in one-pass. The compilation time overhead of our benchmarks is low and not significant.

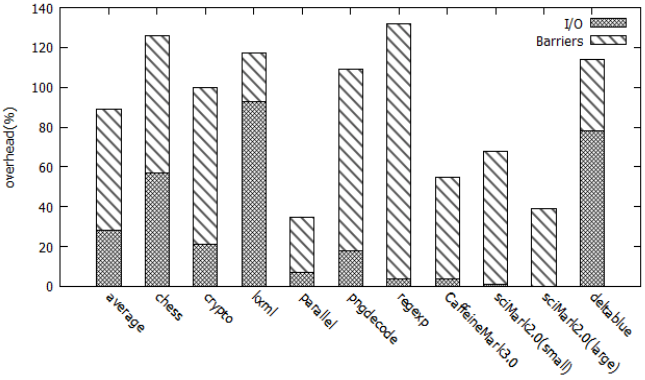

Figure 3: Runtime overheads of the profiler.

Fig.3 shows the execution overhead added by our online tracker on each benchmark. The average overhead imposed on benchmarks is about 89%. Barriers at object access have 61% overhead on average. Dumping the recorded events into files contribute 28% of the total overhead. Chess, kxml and DeltaBlue have a much higher I/O overhead than other benchmarks. One reason is that these benchmarks allocate more objects. Another reason is that we configure a minimum possible heap size. GC occurs frequently. We dumped huge object creation and death information into files.

4.3 Analysis of profiled data

Site	*No. obj*	*Per. %*	*Name*
12C10380	1833825	63.93	com.sun.mep.bench.Chess.Point
12C101C0	660625	23.03	com.sun.mep.bench.Chess.Move
6FC2F7E8	142642	4.97	java.lang.Object[]
6FC77830	52850	1.84	java.util.Vector
6FC98748	50960	1.78	byte[]
Others	…	4.4	…

Table 2: Top classes of Chess

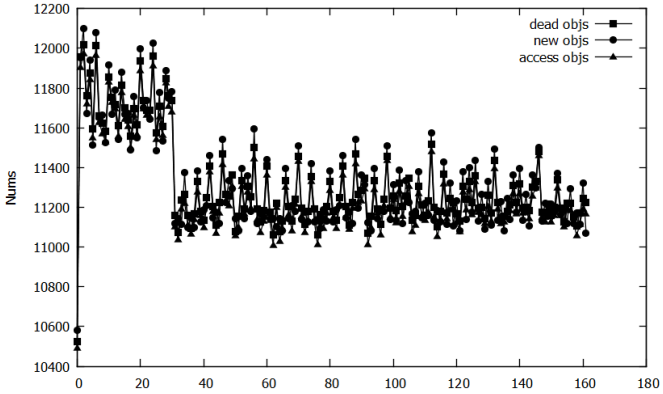

Figure 4: Allocation, collection and access numbers of *com.sun.mep.bench.Chess.Point* in the execution.

Table.2 shows the top classes of *Chess* benchmark. *Chess* is a program that simulates how chessmen move. In *Chess* more than half of the allocated objects have type *com.sun.mep.bench.Chess.Point*. A *Point* object stands for a point in the chessboard. As in Fig.4, *Point* objects are almost all collected in every GC. There may have opportunity to reuse *Point* objects in *Chess* benchmark.

5 Conclusions

In this paper, we proposed YA-JOP, an efficient profiler to profile Java objects on the ART Virtual Machine. Our profiler reports per-object source information such as the allocation site, the last access time. Our profiling method don't introduce any overhead to Java heaps, nor modifies any existing key data structure of the ART virtual machine and Android system. The total execution overheads of our profiler are reasonable. The profiled result gives us sufficient information to find memory leak and help optimizations, such like pretenuring and finding reusable objects. YA-JOP can help developers to find out memory allocation bottlenecks, identify and solve memory problems and efficiently utilize Java heaps.

Acknowledgements

This work was supported by National Natural Science Foundation of China grants No.61272166, the State Key Laboratory of Software Development Environment of China No.SKLSDE-2016ZX-08, and Huawei Research Fund No.HIRPO20140405-YB2015080015.

References

[1] Ali-Gombe, Aisha, et al. "AspectDroid: Android App Analysis System", *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pp. 145-147, (2016).

[2] Bond, Michael D., K. S. Mckinley. "Leak pruning", *Acm Sigarch Computer Architecture News*, **37**, pp. 277-288. (2009) .

[3] Couto M, Carçao T, Cunha J, et al. "Detecting anomalous energy consumption in android applications", *Brazilian Symposium on Programming Languages*, pp. 77-91, (2014).

[4] Hertz M, Blackburn S M, Moss J E B, et al. "Generating object lifetime traces with Merlin", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **28**, pp. 476-516, (2006).

[5] HPROF Viewer. 2016. [Online]. Available: https://developer.android.com/studio/profile/am-hprof.html

[6] Jprofiler. 2016. [Online]. Available: https://www.ej-technologies.com/products/jprofiler/overview.html

[7] Lim E, Lee S, Won Y. "Androtrace: framework for tracing and analyzing IOs on Android", *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, (2015).

[8] Marinov D, O'Callahan R. "Object equality profiling", *ACM SIGPLAN Notices*, **38**, pp. 313-325, (2003).

[9] Odaira R, Nakatani T. "Continuous object access profiling and optimizations to overcome the memory wall and bloat", *ACM SIGARCH Computer Architecture News*, **40**, pp. 147-158, (2012).

[10] Ricci N P, Guyer S Z, Moss J E B. "Elephant tracks: portable production of complete and precise gc traces", *ACM Sigplan Notices*, **48**, pp. 109-118, (2013).

[11] Sartor J B, Hirzel M, McKinley K S. "No bit left behind: the limits of heap data compression", *Proceedings of the 7th international symposium on Memory management*, pp. 111-120, (2008).

[12] Sewe A, Yuan D, Sinschek J, et al. "Headroom-based pretenuring: dynamically pretenuring objects that live long enough", *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pp. 29-38, (2010).

[13] Shi X, Xie J, Yu H. "Address chain: Profiling java objects without overhead in java heaps", *Asian Symposium on Programming Languages and Systems*, pp. 408-427, (2014).

[14] Silva A, Simmonds J. "BehaviorDroid: monitoring Android applications", *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pp. 19-20, (2016).

[15] Su T H, Tsai H J, Yang K H, et al. "Reconfigurable vertical profiling framework for the android runtime system", *ACM Transactions on Embedded Computing Systems (TECS)*, **13**, pp. 59, (2014).

[16] Xu G, Bond M D, Qin F, et al. "LeakChaser: helping programmers narrow down causes of memory leaks", *ACM SIGPLAN Notices*, **46**, pp. 270-282, (2011).

[17] Xu G. "Resurrector: a tunable object lifetime profiling technique for optimizing real-world programs", *ACM SIGPLAN Notices*, **48**, pp. 111-130, (2013).

[18] Yu H, Shi X, Feng W. "LeakTracer: Tracing leaks along the way", *Source Code Analysis and Manipulation (SCAM)*, pp. 181-190, (2015).

[19] Zhang H, Wu H, "Rountev A. Automated test generation for detection of leaks in Android applications", *Proceedings of the 11th International Workshop on Automation of Software Test*, pp. 64-70, (2016).