ABSTRACT

LI, BOLUN. Identifying Performance Inefficiencies via Object-centric Profiling for Java Programs Running on JVM and Android Runtime. (Under the direction of Xu Liu).

Java is the “go-to” programming language choice for developing scalable enterprise cloud applications. In such systems, even a few percent CPU time savings can offer a significant competitive advantage and cost savings. Although performance tools abound for Java, those that focus on the data locality in the memory hierarchy are rare.

My thesis present three lightweight object-centric profiling tools that target on data locality for Java programs. (1) DJXPerf, a memory profiler for Java, which associates memory-hierarchy performance metrics (e.g., cache/TLB misses) with Java objects. DJXPerf uses statistical sampling of hardware performance monitoring counters to attribute metrics to not only source code locations but also Java objects. DJXPerf presents Java object allocation contexts combined with their usage contexts and presents them ordered by the poor locality behaviors. DJXPerf’s performance measurement, object attribution, and presentation techniques guide optimizing object allocation, layout, and access patterns. (2) OJXPerf, a sampling-based profiler, which probabilistically identifies identical objects. OJXPerf employs hardware performance monitoring units (PMU) in conjunction with hardware debug registers to sample and compare field values of different objects of the same type allocated at the same calling context but potentially accessed at different program points. The result is a lightweight measurement — a combination of object allocation contexts and usage contexts ordered by duplication frequency. (3) DroidPerf, a memory profiler for Android applications running on Android Runtime (ART). On ART, managed languages, such as Java and Kotlin, employ various abstractions, runtime support, ahead-of-time (AOT) compilation, and garbage collection (GC), which hide important execution details from the plain source code. DroidPerf pinpoints specific objects in Android programs that exhibit poor locality, guiding locality optimization on Android unique locality issues, such as excessively using Android unoptimized data structure, keeping many Android persistent services running, etc.

All of the tools presented in this thesis require no modifications to hardware, OS, Java virtual machine, ART, or application source code, which makes them attractive to use in production. Guided by DJXPerf, OJXPerf and DroidPerf, we study and optimize a number of Java and Android programs, including well-known benchmarks and real-world popular applications, and demonstrate significant performance gains.
Identifying Performance Inefficiencies via Object-centric Profiling for Java Programs
Running on JVM and Android Runtime

by
Bolun Li

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2022

APPROVED BY:

Xipeng Shen

Guoliang Jin

Huiyang Zhou

Xu Liu
Chair of Advisory Committee
BIOGRAPHY

Bolun Li is a Ph.D. candidate in the Computer Science Department at North Carolina State University. He received his B.S. and M.S. in Computer Science from Sichuan University and Rice University respectively. In 2017, he started his Ph.D. study in Computer Science. His research interest resides in programming languages, program analysis as well as software engineering, with an emphasis on providing tools support for analyzing and optimizing software inefficiencies. Bolun has been actively writing research works in major venues in both software engineering and high-performance/parallel computing areas, such as ICSE, ASPLOS, PPoPP, CGO, and OOPSLA. He was the receipt of NCSU Computer Science Outstanding Graduate Fellowship Award in 2022.
ACKNOWLEDGEMENTS

This dissertation is written with the support and help of many individuals. I would like to thank all of them.

First and foremost, I would like to thank my advisor, Dr. Xu Liu, for his help, encouragement, and patience throughout my doctoral study. His technical advice, as well as his enthusiasm for research, were essential to the completion of this dissertation. I could not have imagined having a better advisor and mentor for my Ph.D. life.

I would like to thank the rest of my thesis committees—Dr. Xipeng Shen, Dr. Guoliang Jin—and the graduate school representative—Dr. Huiyang Zhou—for their time and efforts in scheduling my final defense exam. Their insightful comments and helpful suggestions are essential for completing this dissertation.

I would also like to thank Dr. Yan Yan for his guidance and help during the great internships in two Summers at the Meta Platform and for leading me in working on diverse, exciting projects.

My sincere thanks also go to all the members of the research group, past and present, and friends for their stimulating discussions, constructive suggestions, generous assistance, and effective teamwork, such as Dr. Pengfei Su, Dr. Milind Chabbi, Dr. Shuyin Jiao, Qidong Zhao, Dr. Hao Xu, Yueming Hao, Dr. Qingsen Wang, Dr. Guoqiang Zhang, Zhuoran Liu, Dr. Yushi Chang, and Yuxuan Liu.

Last but not least, I would like to thank my parents—Ang Li and Rong Li—for supporting me spiritually throughout my life. You are always my greatest support and motivation. I literally cannot achieve anything without you.
# TABLE OF CONTENTS

List of Tables ........................................................................................................ vii

List of Figures ........................................................................................................ viii

Chapter 1 INTRODUCTION ..................................................................................... 1
  1.1 Motivations ........................................................................................................ 1
  1.2 Contribution Highlights ................................................................................... 2
    1.2.1 Memory Inefficiencies Detection for Java Programs ............................... 2
    1.2.2 Object Replica Detection for Java Programs ........................................... 4
    1.2.3 Android Unique Memory Inefficiencies Detection ................................. 5
  1.3 Organization ..................................................................................................... 6

Chapter 2 Background ............................................................................................. 7
  2.1 ASM Framework ............................................................................................... 7
  2.2 Java Virtual Machine Tool Interface (JVMTI) ................................................ 7
  2.3 Hardware Performance Monitoring Unit (PMU) ............................................ 8
  2.4 Hardware Debug Register ............................................................................... 8
  2.5 Android Runtime (ART) .................................................................................. 8
  2.6 ART Tooling Interface (ART TI) ..................................................................... 9

Chapter 3 DJXPerf: Identifying Memory Inefficiencies via Object-centric
Profiling for Java Programs .................................................................................. 10
  3.1 Introduction ...................................................................................................... 10
    3.1.1 Motivating Examples .............................................................................. 13
    3.1.2 Paper Contributions .............................................................................. 15
  3.2 Related Work ................................................................................................... 15
    3.2.1 Data Locality Analysis in Java .................................................................. 16
    3.2.2 Java Profilers Based on PMUs ................................................................ 17
    3.2.3 NUMA Optimization for Java ................................................................. 17
  3.3 Methodology ................................................................................................... 18
    3.3.1 Java and JVMTI Agents .......................................................................... 18
    3.3.2 Object-centric Attribution ...................................................................... 19
    3.3.3 NUMA Locality Analysis ...................................................................... 19
    3.3.4 Calling Context Determination ............................................................... 20
    3.3.5 Interfering with Garbage Collection ...................................................... 20
  3.4 Implementation ................................................................................................. 21
    3.4.1 Online Collector ..................................................................................... 21
    3.4.2 Offline Analyzer ..................................................................................... 23
    3.4.3 Discussions ............................................................................................. 24
  3.5 Evaluation ......................................................................................................... 24
  3.6 Case Studies ..................................................................................................... 26
Chapter 4 OJXPerf: Featherlight Object Replica Detection for Java Programs

4.1 Introduction ............................................................. 41
  4.1.1 Observation ..................................................... 44
  4.1.2 Contribution Summary ........................................ 45
4.2 Related Work .......................................................... 46
  4.2.1 Software Approaches ......................................... 46
  4.2.2 Hardware Approaches ......................................... 47
4.3 Methodology ............................................................ 48
4.4 Implementation ........................................................ 51
  4.4.1 Online Data Collector ......................................... 51
  4.4.2 Offline Data Analyzer and GUI .............................. 53
  4.4.3 Discussions ...................................................... 53
4.5 Theoretical Analysis .................................................. 54
4.6 Evaluation ............................................................. 57
  4.6.1 False Positives and Negatives .............................. 59
  4.6.2 Overhead Measurement ....................................... 60
4.7 Case Studies .......................................................... 61
  4.7.1 Soot ............................................................... 61
  4.7.2 Eclipse DeepLearning4J – SameDiff ...................... 62
  4.7.3 Eclipse DeepLearning4J – AlphaGo Zero ................. 63
  4.7.4 FindBugs-3.0.1 .................................................. 64
  4.7.5 fj-kmeans ........................................................ 65
  4.7.6 JGFSerialBench ................................................ 66
  4.7.7 RoaringBitmap ................................................ 66
4.8 Summary ............................................................... 67
Chapter 5  DroidPerf: Profiling Memory Objects on Android Devices . 69

5.1 Introduction ................................................................. 69
  5.1.1 Motivating Examples .................................................. 71
  5.1.2 Contribution Summary ............................................... 72
5.2 Categorizing Memory Inefficiencies .................................. 73
  5.2.1 Inefficiency due to Memory Bloat .................................. 73
  5.2.2 Inefficiency due to Long Running Services ...................... 75
  5.2.3 Inefficiency due to Misuse of Android APIs ..................... 76
  5.2.4 Inefficiency due to Runtime Configuration Changes ............ 77
  5.2.5 Inefficiency due to Unoptimized Data Structures .............. 78
  5.2.6 Inefficiency due to Duplicated Objects ......................... 78
5.3 Methodology ............................................................... 79
  5.3.1 ART TI Agent .......................................................... 80
  5.3.2 Object-centric Attribution ......................................... 81
5.4 Implementation ........................................................... 82
  5.4.1 Online Collector ...................................................... 85
  5.4.2 Offline Analyzer ..................................................... 85
5.5 Evaluation ..................................................................... 86
5.6 Case Studies ............................................................... 87
  5.6.1 DNS66 ..................................................................... 87
  5.6.2 RxAndroid ............................................................... 88
  5.6.3 Stream Chat ................................................................ 89
  5.6.4 MediaPicker ............................................................. 90
  5.6.5 Foxy Droid ............................................................... 90
5.7 Related Work ............................................................... 91
  5.7.1 HotSpot Analysis ....................................................... 91
  5.7.2 Inefficiency Analysis .................................................. 92
  5.7.3 Lightweight tools ....................................................... 93
5.8 Summary ....................................................................... 94

Chapter 6  Conclusions and Future Directions .......................... 95
References ......................................................................... 97
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overview of performance optimization guided by DJXPerf.</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Optimizing insignificant objects yields little speedups.</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparing OJXPerf with other state-of-the-art inefficiency analysis tools/approaches.</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Overview of performance optimization guided by OJXPerf.</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Accuracy for OJXPERF’s replica detection.</td>
<td>61</td>
</tr>
<tr>
<td>5.1</td>
<td>Measuring behaviors of popular Android apps.</td>
<td>74</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of performance optimization guided by DROIDPerf.</td>
<td>86</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Code-centric vs. object-centric profiling. $O$ denotes an object and $I$ denotes a memory access instruction; tuple $(O_m, I_n)$ denotes that instruction $I_n$ accesses object $O_m$.</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>The workflow of DJXPerf.</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>DJXPerf’s runtime and memory overheads in the unit of times ($\times$) on various benchmarks.</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>The top-down object-centric GUI view of ObjectLayout shows a problematic object’s allocation site in source code, and its allocation call path with its all access call paths.</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Percentage of replicated objects over all the objects in various Java applications.</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Watchpoint scheme for object replica detection. $Off_1$ ($Off_2$) presents memory offset with value $V_1$ ($V_2$) for Object $O_1$ (Object $O_3$). When a watchpoint trap of memory access happens at offset $Off_1$ ($Off_2$), OJXPerf compares their corresponding values $V_1$ and $V_1'$ ($V_2$ and $V_2'$).</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Workflow of object-level redundancy detection.</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>OJXPerf’s runtime and memory overheads in the unit of times ($\times$) on various benchmarks.</td>
<td>60</td>
</tr>
<tr>
<td>4.5</td>
<td>The Flame Graphs GUI of Soot shows a problematic object $st$ with its accesses in full call stacks.</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of DROIDPerf’s profiling methodology.</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of DROIDPerf in the system stack.</td>
<td>83</td>
</tr>
<tr>
<td>5.3</td>
<td>DROIDPerf’s runtime and memory overheads in the unit of times ($\times$) on various benchmarks.</td>
<td>84</td>
</tr>
<tr>
<td>5.4</td>
<td>The object-centric GUI view of DNS66 shows two problematic objects’ allocation sites in DROIDPerf’s Flame Graphs GUI in VSCode.</td>
<td>89</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

1.1 Motivations

Inefficiencies are present in every aspect of software, no matter whether they are mobile applications, desktop applications, or large-scale parallel applications running on supercomputers. Software packages have become increasingly complex at the code level, comprised of a large amount of source code, multi-layer abstractions, and deep call chains. Without careful design and monitoring, software developers can easily introduce performance inefficiencies embedded deep in the code bases that go unnoticed or remain challenging to diagnose, leading to performance degradation. Hardware evolution outpaces the performance optimization of software at the resource level, leading to resource wastage and energy dissipation in emerging architectures. Even worse, due to Java Virtual Machine (JVM), a step removed from the underlying hardware makes JVM-based managed languages suffer from worse performance than native languages.

Performance profiling tools abound in the Java community to aid software developers in understanding program behavior. Profiling for execution hotspots and inefficiencies are the most popular ones. Hotspot analysis profilers [Cohen et al. 2018; ej-technologies GmbH. 2018; GmbH. 2018; Linux 2015; Oracle 2016, 2017, 2018; Pangin 2018] identify code regions that consume a large portion of resources disregarding whether these resources are being used productively; instead, tool users need to make a judgment call. Inefficiency analysis profilers [Della Toffola et al. 2015b; Eizenberg et al. 2016; F. Sweeney et al. 2004; Hauswirth et al. 2004; Khanh et al. 2013; Nistor et al. 2013; Song et al. 2017; Su et al. 2019a; Xu 2012; Xu et al. 2009a, 2010b] identify code regions that waste resources instead of consuming resources. Distinguished from hotspot analysis tools, inefficiency
analysis can guide users to concentrate on code regions involved in some inefficiency.

A Java object is a combination of properties and methods working on the properties. Software developers primarily think of Java code in terms of objects which form mental boundaries of functionality. It is natural that when developers investigate performance inefficiencies, they often tend to analyze the code at the object-level granularity. A recent line of inefficiency analysis tools have demonstrated that redundancies are a significant software inefficiency indicator in both native languages [Chabbi et al. 2012; Su et al. 2019b; Wen et al. 2017a, 2018] and managed languages [Della Toffola et al. 2015a; Dhok et al. 2016; Nistor et al. 2013; Song et al. 2017; Su et al. 2019a] such as Java. However, they are missing a critical piece — the object-level inefficiencies that happen across objects in Java programs. Our observation is that various kinds of inefficiencies are due to improper use or allocate the Java objects.

To overcome this critical missing piece, we propose DJXPERF, OJXPERF and DROIDPERF, three lightweight sampling-based performance analysis profilers with novel profiling techniques to pinpoint object-level inefficiencies on JVM and ART.

1.2 Contribution Highlights

1.2.1 Memory Inefficiencies Detection for Java Programs

In modern systems, compute is “free” but memory accesses cost dearly. Long-latency memory accesses are a major cause of execution stalls on modern general-purpose CPUs. CPU’s memory hierarchy (different level of caches) offers a means to reduce average memory access latency by staging data into caches and accessing repeatedly before evicting. Access patterns that reuse previously fetched data are said to exhibit good data locality.

There are traditionally two types of data locality: spatial and temporal. An access pattern exhibits spatial locality when it accesses a memory location and then accesses nearby locations soon afterward. An access pattern exhibits temporal locality when it accesses the same memory location multiple times. Programs that do not exploit these features are said to lack spatial or temporal locality. Besides the traditional locality problems, garbage collected languages such as Java expose another unique locality issue — memory bloat [Xu 2012]. Memory bloat occurs by allocating (and initializing) many objects whose lifetimes do not overlap. For example, allocating objects in a loop where the lifetime of the object is only the scope of the loop body. Since the garbage collection happens sometime later in the future, the memory consumption spikes, which results in
a higher memory footprint and results in suboptimal cache utilization. Memory bloat can be seen as a case of lack of both spatial and temporal locality since accessing a large number of independent objects results in accessing disparate cache lines (lack of spatial locality) and little or no reusing of a previously accessed cache line(s) (lack of temporal locality).

Beyond the traditional caches, data locality can exist in the main memory. The non-uniform memory access (NUMA) becomes a mainstream shared-memory architecture, integrating multiple processors (as sockets or NUMA domains) with their own memory controllers. A socket may access memory attached to another socket known as remote accesses; In contrast, a socket accessing the memory attached to itself is known as local accesses. Typically remote accesses incur higher latency and lower bandwidth over local accesses, known as poor NUMA locality. Co-locating data with computation can exploit NUMA locality.

Since developers in native languages have explicit knowledge and understanding of objects and their lifetimes, they pay more attention to objects and their locality; Java developers, on the other hand, lack the precise knowledge of object lifetimes and their influence on locality. Surprisingly, Java lacks any object-centric performance attribution tool, pinpointing objects subject to serious latency problems. Tools such as Linux perf [Linux 2015] and Intel VTune [Intel 2019] exploit hardware performance monitoring and attribute cache miss metrics to code, such as Java methods, loops, or source locations, but they do not attribute metrics to Java objects. A few tools [Mitchell et al. 2010; Xu 2011; Xu et al. 2010a] instrument Java byte code to identify problematic objects. However, these tools suffer from high overhead and lack real execution performance metrics available from the hardware. Oftentimes, the optimization lacking the quantification from the underlying hardware can yield trivial or negative speedups [Liu et al. 2013b].

We propose DJXPerf, a lightweight object-centric profiler for Java programs. DJXPerf complements existing Java profilers by collecting memory-related performance metrics from hardware PMUs and attributing them to Java objects. DJXPerf provides unique insights into Java’s memory locality issues. DJXPerf works on off-the-shelf Java virtual machine (JVM) with minimal Java byte code instrumentation, which typically incurs 8.5% runtime and 6% memory overhead. In well-known Java benchmarks and real-world applications, we demonstrate that DJXPerf identifies data locality inefficiencies and obtains significant speedups after eliminating such inefficiencies. Guided by DJXPerf, we are able to obtain significant speedups by improving data locality in various Java programs.
1.2.2 Object Replica Detection for Java Programs

Prior approaches to memory bloat have focused on identifying objects that outlive their life span. Few studies have, however, looked into whether and to what extent myriad objects of the same type are identical. A quantitative assessment of identical objects with code-level attribution can assist developers in refactoring code to eliminate object bloat, and favor reuse of existing object(s). The result is reduced memory pressure, reduced allocation and garbage collection, enhanced data locality, and reduced re-computation, all of which result in superior performance.

In this work, we define two objects are replicas if their contents are identical. Two objects are shallow replicas if their fields are bitwise identical; and deep replicas if the transitive closure of the constituent objects and their respective fields values are bitwise identical. There is a temporal aspect to replica objects: one may regard two objects as replicas either because they were identical at the time (or a window) of observation or because they were identical for their entire lifetime. Two objects are mutable replicas if they are identical to each other, but they may independently evolve during their lifetimes; whereas two objects are immutable replicas if they are identical to each other and are also immutable during their lifetimes.

There are two performance dimensions to replica objects: total memory consumption and total number of memory accesses. On the one hand, an object may be large in size and replicated only a few times, and such replicas still contribute to the overall memory bloat; also, an object may be small in size but replicated many times, which also contributes to memory bloat; both these cases are worth optimizing to remove the replicas. On the other hand, an object may be small in size and replicated only a few times, but the program may be accessing these few replicated objects a lot of times; although this situation is not memory bloat, it can still have a significant consequence to the overall performance since it increases the memory footprint at the CPU cache level, squanders potential memory reuse [Allen et al. 2001], and often results in redundant re-computations [Nguyen et al. 2013; Wen et al. 2017b].

We propose OJXPerf, a lightweight sampling-based profiler, which probabilistically identifies identical objects. OJXPerf employs hardware performance monitoring units (PMU) in conjunction with hardware debug registers to sample and compare field values of different objects of the same type allocated at the same calling context but potentially accessed at different program points. The result is a lightweight measurement — a combination of object allocation contexts and usage contexts ordered by duplication...
frequency. This class of duplicated objects is relatively easier to optimize. OJXPERF in- 
curs 9% runtime and 6% memory overheads on average. We empirically show the benefit 
of OJXPERF by using its profiles to instruct us to optimize a number of Java programs, 
including well-known benchmarks and real-world applications. The results show a notice-
able reduction in memory usage (up to 11%) and a significant speedup (up to 25%).

1.2.3 Android Unique Memory Inefficiencies Detection

The free and open-source Android platform has established itself as the dominant mobile 
operating system on the market. Android-based mobile phones and tablets are the most 
popular electronic devices. Java and Kotlin are the de-facto programming languages for 
Android apps, which are run in Android Runtime (ART) virtual machine. ART employs 
garbage collectors (GC) to manage memory objects allocated on heap, which effectively 
avoid many memory errors, such as memory leaks.

However, ART is not a panacea for memory inefficiencies, which are rooted at the 
growing speed gap between CPU and memory in Android ecosystems. Memory ineffi-
ciences not only slow down the app execution, but also drain the battery quickly. One 
the one hand, apps with poor data locality can result in excessive cache misses, increas-
ing the average memory access latency. On the other hand, ART can introduce memory 
bloats Xu [2012] due to the delay of GC for reclaiming unused data objects, which in-
crease the memory footprint. Thus, it is urgent for Android apps to maximize memory 
efficiencies by enhancing data locality and avoiding memory bloats.

There are few studies about data locality issues happened on ART because of the 
unique challenges in the abstraction introduced by the managed runtime on ART. First, 
ahead-of-time (AOT) compilation and interpretation on ART separate program source 
code from its execution behaviors. Second, further preventing understanding of memory 
performance of Android programs is the managed memory runtime system, which in-
cludes virtual machines and garbage collection. Third, unlike many Java profilers that 
have the Java Virtual Machine Tool Interface (JVMTI) supported, JVMTI on ART is par-
tially supported. Hence, some essential functions for profiling tools are not available. For 
example, the JVMTI compiled method load and unload callbacks (sent when a method 
is compiled and loaded/unloaded into memory by the VM) cannot be used on ART. 
And lastly, ART does not have the ASYNC unwind facility (the AsyncGetCallTrace 
API provided by OpenJDK/oracle JVMs to facilitate non-safepoint collection of stack 
traces), which means profiler developers are not able to get the call stack inside a signal
handler.

Given these unique challenges on ART, the existing tool, Simpleperf [sim 2022], adapts PMU-based approaches to monitor hardware events (e.g., cpu cycles, cache hit/miss) for running Android applications. However, without object-level information, Simpleperf has difficulties in providing actionable optimization guidance for object inefficiencies. Some other tools such as Android Profiler [and 2021] and Perfetto [per 2021], they are able to have the hotspot analysis (e.g., record a memory trace with high usage, capture a heap dump). However, these tools lack of performance statistics collected from hardware. Optimizing without quantifying the underlying hardware can often only yield trivial or negative speedups.

We propose DROIDPERF, an object-centric memory profiler for ART, which associates memory-hierarchy performance problems collected from hardware to objects in Android programs. DROIDPERF identifies six types of inefficiencies that lead to locality issues in Android applications: high memory usage in loops, long persistent services running, improper use of Android APIs, excessively use of Android unoptimized data structure, runtime configuration changes, and duplicated objects. These locality issues are a result of traditional spatial/temporal data locality as well as memory bloat. DROIDPERF incurs 14% runtime overhead and 8% memory overhead on average. In order to use DROIDPERF in production, DROIDPERF does not require any changes to hardware, OS, Dalvik virtual machine, ART, or Android application source code. Guided by DROIDPERF, we optimize a number of popular Android applications, yielding significant performance gains.

1.3 Organization

The dissertation is organized as follows. Chapter 2 covers the background knowledge needed to comprehend the technical details. Chapters 3, 4, and 5 depict the methodology and implementation of DJXPERF, OJXPERF, and DROIDPERF, as well as evaluate their overhead and effectiveness by applying them on a number of benchmarks and real-world popular applications. Our conclusions are presented in Chapter 6, along with several possible future directions.
Chapter 2

Background

The purpose of this chapter is to provide some background knowledge that will aid in understanding the technical details in the subsequent chapters.

2.1 ASM Framework

ASM [Bruneton et al. 2019] is a Java byte code manipulation and analysis framework. ASM can modify existing classes or dynamically generate classes, supporting custom complex transformations and code analysis tools. ASM focuses on performance, with an emphasis on the low overhead, which makes it suitable for dynamic analysis. ASM can instrument object allocation (e.g., \texttt{new}) and capture the object information, such as allocation size and context.

2.2 Java Virtual Machine Tool Interface (JVMTI)

JVMTI [Oracle 2007] is a native programming interface of the JVM, which supports developing debuggers/profilers (aka JVMTI agents) in C/C++ based native languages to inspect JVM internals. JVMTI provides a number of event callbacks to capture JVM start and end, thread creation and destruction, method loading and unloading, garbage collection epochs, to name a few. User-defined functions are subscribed to these callbacks and invoked when the associated events happen. Additionally, JVMTI maintains a variety of JVM internal states, such as the map from the machine code of each JITted method to byte code and source code, and the call path for any given point during the execution.
Tools based on JVMTI can query these states at any time. JVMTI is available in off-the-shelf Oracle HotSpot JVM [Oracle 2019].

2.3 Hardware Performance Monitoring Unit (PMU)

Modern CPUs expose programmable registers (aka PMU) that count various hardware events such as memory loads, stores, and CPU cycles. These registers can be configured in a sampling mode: when a threshold number of hardware events elapse, PMUs trigger an overflow interrupt. A profiler can capture the interrupt by taking a sample and attribute the metrics collected along with the sample to the execution context. PMUs are per CPU core and virtualized by the operating system (OS) for each thread.

Intel offers Precise Event-Based Sampling (PEBS) [Intel 2010] in SandyBridge and following generations. PEBS provides the effective address (EA) at the time of the sample when the sample is for a memory load or store instruction. PEBS also reports memory-related metrics of the sampled loads/stores such as cache misses, TLB misses, memory access latency. This facility is often referred to as address sampling — a building block of DJXPerf. AMD Instruction-Based Sampling [Drongowski 2010] and PowerPC Marked Events [Srinivas et al. 2011] offer similar capabilities.

2.4 Hardware Debug Register

Modern x86 processors provide debug facilities for developers in debugging code and monitoring system behaviors. Such debug support is accessed using hardware debug registers. Hardware debug registers [E. McLear et al. 1982; Scott Johnson 1982] enable trapping the CPU execution for debugging when the program counter (PC) reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). Hardware debug registers allow programmers to selectively enable various debug conditions associated with a set of four debug addresses because our current x86 processors have four debug registers.

2.5 Android Runtime (ART)

Android Runtime (ART) [clo 2014] is an application runtime environment used by the Android operating system. Replacing Dalvik, the process virtual machine originally used
by Android, ART translates of the application’s bytecode into native instructions that are later executed by the device’s runtime environment. In ART, entire applications are compiled into native machine code before installation by using ahead-of-time (AOT) compilation, which eliminates the Dalvik’s interpretation and trace-based JIT compilation. JIT compilation will not only steal resources away from the running program, it will also need battery power, and it will need that every time the program runs, whereas an AOT compiler will only need to spend that power budget once, when the app is installed. Additionally, ART provides faster execution of applications, improved memory allocation and garbage collection (GC) mechanisms, new applications debugging features, and more accurate high-level profiling of applications [art 2022a; how 2022].

2.6 ART Tooling Interface (ART TI)

In Android 8.0 and higher, the ART Tooling Interface (ART TI) [art 2022b] exposes certain runtime internals, and enables profilers and debuggers to influence the runtime behavior of applications. This can be used to implement state-of-the-art performance tools that are provided for implementing native agents on other platforms. Runtime internals are exposed to agents that have been loaded into the runtime process. These communicate with the ART through direct calls and callbacks. Different orthogonal-profiling concerns can be separated using the runtime’s support for multiple agents. Agents may be either supplied at runtime start (when dalvikvm or app_process are invoked), or attached to an already-running process. The Java Virtual Machine Tool Interface (JVMTI) is partially implemented as a runtime plugin in ART TI.
Chapter 3

DJXPerf: Identifying Memory Inefficiencies via Object-centric Profiling for Java Programs

3.1 Introduction

Java is the “go-to” programming language choice for developing scalable enterprise cloud applications [Baum 2016, 2018; Google 2021; Humble 2021; Lozinski 2019; Rocha 2019, 2020]. Performance is critical in such Java programs running on a distributed system comprised of thousands of hosts; on such systems, saving CPU time even by a few percentages offers both competitive advantages (lower latency) and cost savings. Tools abound in Java for “hotspot” performance analysis that can bubble-up code contexts where the execution spends the most time and are also used by developers to tune the performance. However, once such “low hanging targets” are optimized, identifying further optimization opportunities is not easy. Java, like other managed languages, employs various abstractions, runtime support, just-in-time compilation, and garbage collection, which hide important execution details from the plain source code.

In modern computer systems, computing is “free”, but memory accesses cost dearly. Long-latency memory accesses are a significant cause of execution stalls in modern general-purpose CPUs. CPU’s memory hierarchy (different levels of caches) offers a means to reduce average memory access latency by staging data into caches and repeatedly accessing before evicting. Access patterns that reuse previously fetched data are said to exhibit good data locality. There are traditionally two types of data locality: spatial and
temporal. An access pattern exhibits spatial locality when it accesses a memory location and then accesses nearby locations soon afterward. An access pattern exhibits temporal locality when it accesses the same memory multiple times. Programs that do not exploit these features lack spatial or temporal locality.

Maintaining the locality of references in a CPU’s memory hierarchy is well-known and mastered to achieve high performance in natively compiled code such as C and C++. Besides the traditional locality problems, garbage collected languages such as Java expose another unique locality issue — memory bloat [Xu 2012]. Memory bloat occurs by allocating (and initializing) many objects that are not reclaimed immediately after their last usage. For instance, allocating objects in a loop where the object’s lifetime is only the scope of the loop body. Since the garbage collection happens sometime later, the memory consumption spikes, resulting in a higher memory footprint and suboptimal cache utilization. Memory bloat can be seen as a lack of both spatial and temporal locality because accessing a large number of different objects results in accessing disparate cache lines and little or no reusing of a previously accessed cache line(s).

Beyond the traditional caches, data locality can exist in the main memory. The non-uniform memory access (NUMA) becomes a mainstream shared-memory architecture, integrating multiple processors (as sockets or NUMA domains) with their own memory controllers. A socket may access memory attached to another socket known as remote accesses; In contrast, a socket accessing the memory attached to itself is known as local accesses. Typically remote accesses incur higher latency and lower bandwidth over local accesses, known as poor NUMA locality. Co-locating data with computation can exploit NUMA locality.

Exploring data locality in native languages has been under investigation for decades. There exist a number of tools [Berg et al. 2005; Beyls et al. 2006, 2009; Liu et al. 2011, 2013b, 2014a, 2013c, 2014b, 2015; Zhong et al. 2009] to measure the data locality using various metrics. Software metrics such as reuse distances [Ding et al. 2003; Rane et al. 2012; Shen et al. 2007; Zhong et al. 2008, 2009], memory footprints [Bastoul et al. 2003; Kumar et al. 1998], and cache miss ratio curves [Beyls et al. 2009; Çaşçaval et al. 2003], which are derived from memory access traces, quantify the locality independent of architectures. In contrast, hardware metrics, such as cache/TLB misses, collected with hardware performance monitoring units (PMU) during program execution, quantify data locality on a given architecture. Attributing PMU metrics to source code is a straightforward way to demonstrate code regions that incur high access latencies; we call it code-centric profiling.
Figure 3.1: Code-centric vs. object-centric profiling. $O$ denotes an object and $I$ denotes a memory access instruction; tuple $(O_m, I_n)$ denotes that instruction $I_n$ accesses object $O_m$.

In object-oriented systems, delivering profiles centered around objects is highly desirable. An object may have access to it scattered in many places, and each code location may contribute a small fraction to the overall data-access problem. Bridging the memory-hierarchy latency metrics with runtime data objects requires more involved measurement and attribution techniques, which is the focus of our work; we call it object-centric profiling. Note that object-centric profiling not only shows the objects subject to high aggregate access latencies but also pinpoints code locations ordered by their contribution to the overall latency to the object.

Figure 3.1 illustrates the difference between code-centric profiling and object-centric profiling. The code-centric profiling associates the cache miss metric with memory accesses, showing that access $I_c$ accounts for the most cache misses during program execution. In contrast, object-centric profiling aggregates the cache miss metric from different accesses that touch the same object to present a unified view. Guided by object-centric profiling, we find that object $O_1$ accounts for the most cache misses. However, its accesses are scattered across multiple instructions, and each individual access is less significant than the access to object $O_3$. Thus, instead of checking individual accesses, one can apply various optimization to the allocation of the $O_1$ object or its data layout.

Collecting object-centric profiles for Java has unique challenges posed by managed runtimes. First, just-in-time compilation and interpretation used in JVM disjoin the pro-
gram source code and its execution behaviors. Second, automatic runtime memory management (i.e., garbage collection) further impedes understanding memory performance of Java and other JVM languages, e.g., Scala [Piquerez et al. 2019] and Clojure [Hickey 2019].

Since developers in native languages have explicit knowledge and understanding of objects and their lifetimes, they pay more attention to objects and their locality; Java developers, on the other hand, lack the precise knowledge of object lifetimes and their influence on locality. Surprisingly, Java lacks any object-centric performance attribution tool, pinpointing objects subject to serious latency problems. Tools such as Linux perf [Linux 2015] and Intel VTune [Intel 2019] exploit hardware performance monitoring and attribute cache miss metrics to code, such as Java methods, loops, or source locations, but they do not attribute metrics to Java objects. As shown in Figure 3.1, these tools only attribute metrics to problematic code lines; without object-level information, they cannot tell which objects are problematic and deserve optimization. A few tools [Marinov et al. 2003; Mitchell et al. 2010; Xu 2011; Xu et al. 2010a] instrument Java byte code to identify problematic objects. However, these tools suffer from high overhead and lack real execution performance metrics available from the hardware. Oftentimes, the optimization lacking the quantification from the underlying hardware can yield trivial or negative speedups [Liu et al. 2013b].

In this thesis, we describe DJXPerf, a lightweight object-centric profiler for Java programs. DJXPerf complements existing Java profilers by collecting memory-related performance metrics from hardware PMUs and attributing them to Java objects. DJXPerf provides unique insights into Java’s memory locality issues. In the rest of this section, we show two motivating examples, the contributions of this work.

### 3.1.1 Motivating Examples

This section motivates the importance of combining object-level information and PMU metrics for locality optimization. Listings 3.1 and 3.2 show two problematic code snippets suffering from memory bloat, which are respectively from batik and lusearch, both from Dacapo-9.12 [M. Blackburn et al. 2006]. We run them with the default large inputs using 48 threads.

In Listing 3.1, the object allocation site at line 11 creates an array of float objects in the method `makeRoom`, which is part of the class `ExtendedGeneralPath`. This allocation site is repeatedly invoked 2478 times, resulting in memory bloat. For optimization, one
public class ExtendedGeneralPath {
    public append() {
        while (...) {
            makeRoom(n);
            ...
        }
    }
    private void makeRoom(int numValues) {
        ...
        if (newSize > values.length) {
            ...
            float [] nvals = new float[nlen];
            System.arraycopy(values, 0, nvals, 0, numVals);
            ...
        }
    }
}

Listing 3.1: The code snippet from Dacapo 9.12 batik. Optimizing memory bloat by moving the object allocation site at line 11 outside of the loop yields a nontrivial speedup \((1.15 \pm 0.03)\times\) to the entire program.

public TopDocs search(Weight weight, Filter filter, final int nDocs) {
    ...
    for (...) {
        TopDocCollector collector = new TopDocCollector(nDocs);
        search(weight, filter, collector);
        ...
    }
}

Listing 3.2: The code snippet from Dacapo 9.12 lusearch. Optimizing memory bloat by moving the object allocation site at line 4 outside of the loop does not bring any speedup to the entire program.

can move the array allocation outside of the loop that encloses the method makeRoom and replace it with a static object array, aka the singleton pattern. This optimization addresses the memory bloat and yields a \((1.15 \pm 0.03)\times\) speedup.

In Listing 3.2, the memory bloat occurs at line 4, which repeatedly allocates the object collector 15179 times. This object is passed as an input parameter to the method search and is used in many places in that method. One can also apply the singleton pattern by hoisting the allocation site outside the loop. Unfortunately, while this optimization addresses the memory bloat, it does not bring any noticeable speedup.

The study of these two example code snippets reveals that basing the optimization only on allocation frequency (or the metrics derived from the allocation frequency [Xu 2012]) does not necessarily yield performance benefits. This motivates the need for the extra locality metrics associated with the object allocation site, which we call as object-centric profiling. To be concrete, DJXPERF measures L1 cache misses\(^1\) with PMU on

\(^1\)we can measure myriad events, e.g., L3 cache misses, TLB misses, etc.
individual memory access instances and aggregates the measurement of memory accesses to the object’s allocation site. For example, DJXPerf reports that accessing the `nvals` array object shown in Listing 3.1 accounts for 21% of total cache misses, while accessing the `collector` objects in Listing 3.2 accounts for less than 1% of total cache misses only, which explains the different speedups obtained from the locality optimization. The strength of DJXPerf is its ability to aggregate myriad accesses to the same object, scattered all over the program, back to the same object. We emphasize that object-centric analysis does not do away with the code-centric aspect; underneath each object allocation site $C$, DJXPerf provides the ability to disaggregate the code contexts contributing towards $C$’s overall locality loss. Thus, object-centric analysis with the locality metrics associated with the object allocation sites is desired to determine whether locality optimization can yield significant speedups.

### 3.1.2 Paper Contributions

This paper makes the following contributions:

- We give the first systematic study on data locality issues in Java programs via characterizing their root causes and optimization strategies.

- We show the design and implementation of DJXPerf, an object-centric profiler that guides data locality optimization in Java programs.

- DJXPerf works on off-the-shelf Java virtual machine (JVM) with minimal Java byte code instrumentation, which typically incurs 8.5% runtime and 6% memory overhead.

- In well-known Java benchmarks and real-world applications, we demonstrate that DJXPerf identifies data locality inefficiencies and obtains significant speedups after eliminating such inefficiencies. Some of our optimization patches have been upstreamed to the code repositories.

### 3.2 Related Work

There are numerous Java performance tools assisting developers in understanding their program behaviors, such as profiling for execution hotspots [Cohen et al. 2018; ej-technologies GmbH. 2018; GmbH. 2018; Linux 2015; Oracle 2018; Pangin 2018] in CPU cycles or heap usage, and pinpointing redundant computation [Della Toffola et al. 2015a; Dhok
et al. 2016; Nistor et al. 2013; Song et al. 2017]. These tools target orthogonal problems to DJXPerf, which particularly focuses on data locality. Furthermore, there are many tools [Buck et al. 2004; Lachaize et al. 2012; Liu et al. 2011, 2014a, 2013c, 2015; McCurdy et al. 2010; Roy et al. 2016, 2018] pinpointing poor locality issues in native code via OS timers or PMU-based sampling techniques, or code instrumentation. Unlike DJXPerf, these tools do not work for Java applications. In this section, we only review Java profiling techniques that are related to data locality and PMUs.

### 3.2.1 Data Locality Analysis in Java

Most existing Java profilers focus on memory bloat, which is one of the locality issues. [Mitchell et al. 2007] design a mechanism to track data structures that suffer from excessive amounts of memory. Their follow-up work [Mitchell et al. 2009, 2010] summarizes memory usage to uncover the costs of design decisions, which provides intuitive guidance for code improvement.

[Xu et al. 2009a] develop copy profiling that detects data copies and suggests removal of allocation and propagation of useless objects. Their follow-up work [Xu et al. 2010b] presents a technique that combines static and dynamic analyses to identify underutilized and overpopulated containers. They also develop a dynamic technique [Xu 2012] to highlight data structures that can be reused to avoid frequent object allocations.

Nguyen and Xu [Khanh et al. 2013] develop Cachetor, a value profiler for Java. Cachetor identifies operations that keep generating identical data values and suggests memoizing the invariant values for future usage. [Yan et al. 2012] track object propagation by monitoring object allocation, copy, and reference operations; by constructing a propagation graph, one can identify never-used or rarely-used object allocations. [Dufour et al. 2007, 2008] analyze the use and shape of temporary data structures based on a blended escape analysis to find excessive memory usage. JOLT [Shankar et al. 2008] uses dynamic analysis to identify object churn and performs function inlining.

There are few studies in measuring traditional data locality in Java programs. [Gu et al. 2009] develop ViRDA, which is perhaps the most related to DJXPerf. They collect memory access traces and compute reuse distances to quantify the temporal and spatial data locality in Java programs.

While these existing efforts can identify some locality issues in Java, they predominantly suffer from two limitations. First, they employ fine-grained byte code instrumentation, which incurs high overhead. For example, the work [Khanh et al. 2013; Xu et al.
2009a] can incur 30-200× runtime overhead. Second, they do not collect performance data from the real execution provided by the hardware; instead, they employ cache simulators. Without the information of the underlying hardware, optimization efforts may be misguided, as shown in Section 3.1.1.

DJXPERF addresses these limitations by introducing an object-centric profiling technique, which is based on lightweight data collection from hardware PMUs. DJXPERF is not a replacement for existing tools; it provides complementary information to save non-fruitful optimization efforts.

### 3.2.2 Java Profilers Based on PMUs

[F. Sweeney et al. 2004] develop a system to help interpret results obtained from PMUs when analyzing a Java application’s behavior. [Cuthbertson et al. 2009] map the instruction IP address based hardware event information to the JIT server components. Goldshtein [Goldshtein 2018] monitors CPU bottlenecks, system I/O load, and GC with perf in production JVM environments. [Hauswirth et al. 2004] introduce vertical profiling, adding software performance monitors (SPMs) to observe the behavior in the layers (VM, OS, and libraries) above the hardware. [Georges et al. 2004] measure the execution time for each method invocation with PMUs and study method-level phase behaviors in Java applications. [Lau et al. 2006] guide inline decisions in a dynamic compiler with the direct measure of CPU cycles. [Eizenberg et al. 2016] utilize PMUs to identify false sharing in Java programs.

Unlike these existing approaches, DJXPERF leverages PMUs to identify data locality in Java programs. Its usage of lightweight PMU measurement for object-centric analysis is unique among all existing Java profilers.

### 3.2.3 NUMA Optimization for Java

[Gidra et al. 2013] study the scalability of throughput-oriented GCs and propose to map pages and balance GC threads across sockets. [Gidra et al. 2015] propose a local mode to prevent GC threads from stealing references in remote sockets. [Carpen-Amarie et al. 2015] show the optimization of three mainstream GCs in OpenJDK for NUMA. [M.Tikir et al. 2005] propose NUMA-aware heap configurations for Java server applications to improve the memory performance during the GC phase. [Raghavendra et al. 2006] propose a dynamic compiler scheme for splitting the Java code buffer on a NUMA machine.
These approaches optimize NUMA locality via adapting GC configurations. In contrast, DJXPERF, without modifying default GC, pinpoints objects that incur remote accesses to guide optimization of object layouts and memory accesses.

3.3 Methodology

DJXPERF includes two agents: a Java agent and a JVMTI agent. The Java agent adds lightweight byte code instrumentation to capture object allocation information during execution, such as the allocation context and address range of objects. The JVMTI agent subscribes to Java thread creation callbacks to enable PMU to sample memory accesses. When PMU interrupts a thread with a sampled address, DJXPERF associates the address seen in the sample with the Java object enclosing that address. In the rest of this section, we elaborate on each agent and discuss their interactions.

3.3.1 Java and JVMTI Agents

Capturing Object Addresses via A Java Agent  DJXPERF leverages a Java agent to capture object allocation. The Java agent is based on the ASM framework [Bruneton et al. 2019]. The Java agent scans Java byte code and instruments four object allocation routines — new, newarray, anewarray, and multianewarray. The Java agent inserts pre- and post-allocation hooks to intercept each object allocation and returns the object information (e.g., object pointer, type, and size) via user-defined callbacks. Upon each allocation callback, we follow an existing technique [Jrebel 2013] to obtain the memory range allocated for each Java object.

Generating Memory Access Samples via JVMTI Agent  DJXPERF leverages a JVMTI agent to sample and collect memory accesses. With the help of JVMTI, DJXPERF can intercept Java thread start, where DJXPERF configures PMUs to sample precise events for cache misses (e.g., MEM_LOAD_UOPS_RETIRED:L1_MISS), TLB misses (e.g., DTLB_LOAD_MISSES), or memory access latency (e.g., MEM_TRANS_RETIRED:LOAD_-LATENCY). DJXPERF also installs a signal handler to process PMU samples. On thread termination, DJXPERF stops PMUs and produces a profile for each thread. Besides controlling PMUs, DJXPERF also utilizes the JVMTI agent to capture the calling contexts for both PMU samples and object allocations (see Section 3.3.4).
3.3.2 Object-centric Attribution

Identifying Objects  Java objects are allocated on the heap. How to represent an object to a developer is a challenging question. We adopt a simple and perhaps the most intuitive approach that developers can identify with the call path leading to the object allocation. We represent a call path where an object $O$ was allocated with $\mathcal{P}(O)$. An application may create multiple objects via a single allocation site, for example, in a loop. In our approach, all such objects are represented by a single call path and become indistinguishable from one another; we accept this trade-off since objects allocated at the same call path are likely to exhibit similar behavior. DJXPERF’s Java agent captures each allocation and invokes the JVMTI agent to obtain the allocation call path. As these allocation instances share the same call path, DJXPERF associates the PMU metrics for any of those objects with the same call path.

Attributing PMU Samples to Objects  DJXPERF utilizes an interval splay tree [Dominic Sleator et al. 1985] to maintain the memory ranges allocated for all the monitored objects. A splay tree is a binary search tree with the additional property that recently accessed elements are quick to access again. Like self-balancing binary search trees, a splay tree performs basic operations such as insertion, look-up and removal in $O(\log n)$ amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even performing better than $O(\log n)$ for sufficiently non-random patterns.

On each PMU sample, DJXPERF uses the effective address $M$ presented by the PMU to look up into the splay tree. The lookup for $M$ returns the object $O$ whose memory range encloses the sampled address. DJXPERF then attributes any associated PMU metric related with the sample to $\mathcal{P}(O)$ — the object’s allocation call path.

3.3.3 NUMA Locality Analysis

Upon each PMU sample, DJXPERF, with the help of move_pages system call, identifies the location (i.e., socket $N_1$) of the memory page that encloses the effective address $M$ collected from PMU. DJXPERF then obtains the CPU ID running the current thread via perf_events and the socket ($N_2$) that this CPU belongs to. If $N_1$ and $N_2$ are distinct, DJXPERF reports a remote memory access and accumulates it to the corresponding object.
3.3.4 Calling Context Determination

We associate an object allocation with the full calling context (aka call path) leading to its allocation. To obtain the full call path, DJXPerf does not set an upper bound on the length of the call path. The full call path helps distinguish allocations by the same routine called from different code contexts. The alternative, a flat profile, would be unable to distinguish, for example, an allocation in a common library routine called from two different user code locations.

3.3.5 Interfering with Garbage Collection

The garbage collector (GC) complicates object-centric attribution because GC implicitly reclaims memory of unused objects and moves objects for compact memory layouts. The trigger of the GC thread is determined by JVM, which is transparent to Java applications. The scenario of GC reclamation is simple. Whenever a memory interval is allocated for a new object and that memory interval overlaps with any existing memory intervals in the splay tree, the DJXPerf can just remove the overlapped memory intervals from the splay tree and add the newly allocated object to the splay tree. However, the situation becomes complicated for the object movement. Ignoring object movement by GC, DJXPerf may yield incorrect object attribution. For example, if we assume GC moves an object to a new memory location, then any subsequent PMU samples of the new address will be unavailable for us to map via the original mapping maintained in the splay tree.

Thus, handling object movement by GC is necessary for DJXPerf. Unfortunately, JVM exposes limited information about GC via JVMTI: JVMTI only provides hooks to register callbacks on GC start and end, with no insight into individual object behavior (i.e., object movement). We offer a general solution to handle all kinds of GC (e.g., Parallel GC, G1 GC, etc.) in the off-the-shelf JVM.

Solution for Object Movement by GC Our solution is based on an important observation from the source code of OpenJDK: GC moves objects using the memmove function. Thus, DJXPerf overloads memmove to obtain the source and destination of every moved object and update the memory ranges associated with this object in the splay tree described in Section 3.3.2. However, updating the splay tree upon each memmove invocation is costly. Instead, DJXPerf creates a relocation map for each thread to record the moved objects (e.g., source as the key, destination memory addresses, and size as the value). DJXPerf updates the objects in the map in a batch at the end of each GC.
invocation.

To capture every GC invocation, DJXPERF utilizes JVM interface **MXBean**. DJXPERF registers GC invocation callbacks via the **GARBAGE_COLLECTION_NOTIFICATION** event. Upon each GC completion, an **MXBean** instance (i.e., **GarbageCollectorMXBean**) emits a callback; DJXPERF captures this callback and updates all the newly moved objects in the relocation map to the splay tree. The relocation map is reset after the update.

It is worth noting that DJXPERF may not always capture all the object allocations because its attach mode may omit some allocations (see Section 3.4.1). In this case, DJXPERF directly inserts the new memory intervals for the moved objects.

### 3.4 Implementation

The implementation of DJXPERF consists of 15K lines of code in multiple languages — 50.4% C++, 22.6% Java, 14.9% C, 6.8% Python, and 5.3% Starlark&Shell scripts. DJXPERF is a user-space tool with no need for any privileged system permissions. DJXPERF requires no modification to hardware, OS, JVM, or monitored applications, which makes DJXPERF applicable to production environment. Figure 3.2 shows the workflow of DJXPERF, which consists of an online data collector and an offline data analyzer. There are two ways to enable the collector. If we need to profile Java source code as soon as the JVM starts up, we can launch DJXPERF as an agent by passing JVM options. If the JVM is already started, we can attach DJXPERF to this running JVM. The collector gleans the measurement via the Java and JVMTI agents and generates a profile file per thread. The analyzer then aggregates the files from different threads, sorts the metrics, and highlights the problematic objects for investigation.

The implementation challenges include maintaining a low measurement overhead and scaling the analysis to many threads. In the rest of this section, we discuss how DJXPERF addresses these challenges.

#### 3.4.1 Online Collector

DJXPERF supports two modes to monitor a Java program. On the one hand, DJXPERF can monitor the end-to-end execution of a program by launching the tool together with the program. On the other hand, DJXPERF can attach and detach to any running Java program to collect the object-centric profile for a while. This is particularly useful to monitor long-running programs such as web servers or microservices.
DJXPerf accepts any memory-related PMU precise events. In our implementation, DJXPerf presets the event as L1 cache misses \(^2\). We empirically choose a sampling period to ensure DJXPerf is able to collect 20-200 samples per second per thread, which has a good trade-off between runtime overhead and statistical accuracy [Tallent 2010]. Additionally, such a sampling period means that GC will be called much less often than the sample triggers, so GC will not degrade profiling accuracy. To minimize the thread synchronization, DJXPerf has each thread collect PMU samples independently and maintains the calling contexts of PMU samples in a compact calling context tree (CCT) [Arnold et al. 1999], which merges all the common prefixes of given calling contexts. The only shared data structure between threads is the splay tree for objects because an object allocated by a thread can be accessed by other threads. To guarantee the thread-safety of DJXPerf, we use a spin lock to ensure the integrity of the splay tree across threads.

\(^2\)\texttt{MEM\_LOAD\_UOPS\_RETIRED:LI\_MISS}
Figure 3.3: DJXPERF’s runtime and memory overheads in the unit of times (×) on various benchmarks.

3.4.2 Offline Analyzer

To generate compact profiles, which is important for scalability, DJXPERF’s offline analyzer merges profiles from different threads. Object-centric profiles, organized as a CCT per thread, are amenable to coalescing. The analyzer merges CCTs in a top-down way — from call path root to leaf. Call paths for individual objects and their memory accesses can be merged recursively across threads. If object allocation call paths are the same, they have coalesced even if they are from different threads. All memory accesses with their call paths to the same objects are merged as well. Metrics are also summed up when merging CCT nodes. Typically, DJXPERF’s analyzer requires less than one minute to merge all profiles in our experiments. DJXPERF provides a Python-based GUI to visualize the
profiles for intuitive analysis.

### 3.4.3 Discussions

**DJXPerf** is a sampling-based dynamic profiling tool. False negatives can happen because the sampling strategy only identifies statistically significant performance issues and ignores some insignificant ones. Additionally, the sampling rate should be appropriately chosen. A high sampling rate brings high overhead, and a low sampling rate obtains insufficient samples, resulting in over- or under-estimation. Nevertheless, there is sufficient evidence to show that random sampling via PMUs is superior to biased sampling [Mytkowicz et al. 2010]. Like other dynamic profilers, **DJXPerf**’s optimization guidance is input dependent. We recommend using typical program inputs for representative profiles. We ensure the optimization is correct across different inputs. Additionally, **DJXPerf** pinpoints objects potentially for optimization, but users need to determine and apply the optimization. False positives can happen as the problematic objects may have no simple solution to fix.

Based on examining the source code from OpenJDK 8 to OpenJDK 17, we observed that `memmove` function is used by GC to move objects. As there is no real technical difference between OpenJDK and JDK, our solution that overloading `memmove` function for dealing with object movement works on mainstream JVMs. However, we admit there are customized JDKs; for example, Twitter has its own JDK. As a result, our solution may not work as expected since these customized JDKs may modify the GC implementation according to their own preferences. It is also our goal to encourage JVM developers to expose more GC related hooks, which can be used to provide more insights into how objects are behaving during the GC phase.

Finally, for some optimizations in memory bloat issues, we observe the maximum allocation size and allocate the largest possible object. A future request for a larger object is checked by our optimization to see if the current size of the object is sufficient for the application. If not, our optimization automatically reallocates a new larger object so that there is no OOM (OutOfMemory) exception.

### 3.5 Evaluation

We evaluate **DJXPerf** on a 56-core, 4-socket Intel Xeon E7-4830 v4 CPU with a clock speed of 2.0GHz running Linux 5.11. The memory hierarchy consists of a private 32KB
Table 3.1: Overview of performance optimization guided by DJXPerf.

<table>
<thead>
<tr>
<th>Real-world Applications</th>
<th>Problematic Code</th>
<th>Type</th>
<th>Optimization</th>
<th>WS (+)</th>
<th>WCH (%)</th>
<th>WMPKI (%)</th>
<th>WGC (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindBugs 3.0.1</td>
<td>LoadOfKnownNullValue.java (120) ClassPerfUsingASM.java (643)</td>
<td>Memory burst (excessive memory usage in nested loops)</td>
<td>Move problematic allocation sites out of loop and reset them upon request</td>
<td>1.15±0.02</td>
<td>32.6%</td>
<td>14.6%</td>
<td>16.6%</td>
</tr>
<tr>
<td>CoorAscent.java (218)</td>
<td>MergeSorter.java (137, 138)</td>
<td>Memory burst (excessive memory usage in nested loops)</td>
<td>Move problematic allocation sites out of loop and reset them upon request</td>
<td>1.23±0.04</td>
<td>70.8%</td>
<td>49.7%</td>
<td>20.0%</td>
</tr>
<tr>
<td>Apache SAMOA 0.3.0</td>
<td>Hash2.java (313)</td>
<td>Memory burst (excessive memory usage in nested loops)</td>
<td>Move problematic allocation sites out of loop and reset them upon request</td>
<td>1.09±0.02</td>
<td>18.6%</td>
<td>8.5%</td>
<td>13.2%</td>
</tr>
<tr>
<td>Commons Collections 4.2</td>
<td>AbstractHashedMap.java (151)</td>
<td>Memory burst (excessive memory usage in nested loops)</td>
<td>Move problematic allocation sites out of loop and reset them upon request</td>
<td>1.1±0.02</td>
<td>21.3%</td>
<td>9.7%</td>
<td>10.7%</td>
</tr>
<tr>
<td>Renaissance 0.10. scale-ons-bench7</td>
<td>AccessHistory.scale (619)</td>
<td>Problematic data with high L1 cache misses</td>
<td>Enlarge the initial allocation size to avoid frequent allocation</td>
<td>1.12±0.03</td>
<td>20.4%</td>
<td>12.9%</td>
<td>9.0%</td>
</tr>
</tbody>
</table>

WS: Whole-program speedup after optimization. WCH: Whole-program L1 cache hit increase. WMPKI: Whole-program L1 cache misses per kilo-instruction reduction. WGC: Whole-program number of GC reduction.

DJXPerf works for any Oracle JDK version with the JVMTI support. In this evaluation, we run all applications in Oracle Hotspot JVM with JDK 17.0.3.1. We set the initial Java heap size as 256MB and the maximum Java heap size as 1GB. The Garbage-First Garbage Collector (G1 GC) was chosen for this evaluation because G1 GC has been set by default since JDK 9 and is also a long-term-support (LTS) GC algorithm.

L1 cache, a private 256KB L2 cache, a shared 35MB L3 cache, and 256GB main memory.

DJXPerf works for any Oracle JDK version with the JVMTI support. In this evaluation, we run all applications in Oracle Hotspot JVM with JDK 17.0.3.1. We set the initial Java heap size as 256MB and the maximum Java heap size as 1GB. The Garbage-First Garbage Collector (G1 GC) was chosen for this evaluation because G1 GC has been set by default since JDK 9 and is also a long-term-support (LTS) GC algorithm.

Overhead Analysis The runtime overhead (memory overhead) is the ratio of the runtime (maximum RSS memory usage) under monitoring with DJXPerf to the runtime (maximum RSS memory usage) of the corresponding native execution. RSS is the Resident Set Size and is used to show how much memory a process is consuming in physical RAM. Due to the fact that DJXPerf is developed in the native language C++ which is out of the JVM, we are not able to analyze memory overhead based on JVM memory usage. DJXPerf works in the context of the target JVM process. DJXPerf does not consume memory on its own, but the fact of using DJXPerf may add to the RSS of the Java process being profiled. Thus, the maximum RSS memory usage is a desirable metric that can be used to measure how much extra memory overhead (e.g., constructing the splay tree) is introduced by DJXPerf. Note that the amount of extra memory overhead depends on many factors which are mostly application-specific, e.g., number of classes.
and methods, depth of stack traces, number of threads, CPU utilization, etc. A typical memory overhead is usually less than 100MB.

Figure 3.3 quantifies the overhead when DJXPerf is enabled at a sampling period of 5M in both runtime and memory. 5M sampling period means DJXPerf collects 200 samples per second per thread. Figure 3.3 shows that we apply DJXPerf to a number of well-known Java benchmark suites, such as the most recent JVM parallel benchmarks suite Renaissance [Prokopec et al. 2019], Dacapo 9.12 [Blackburn et al. 2018], and SPECjvm2008 [SPEC 2008]. We run all benchmarks with all the 48 threads if applicable. The on-disk memory size of DJXPerf is as small as 536KB. We run every benchmark 30 times and compute the average and error bar. The error bars indicate inconsistent results across different runs. This is reasonable because of the non-deterministic behavior of GC and just-in-time (JIT) compilation. The non-deterministic behavior of GC also leads to the maximum RSS memory usage for some benchmarks with DJXPerf enabled being less than the native run.

Some Renaissance and Dacapo benchmarks have higher time overhead (larger than 30%) because they usually invoke too many allocation site callbacks (e.g., more than 400 million times for mnemonics, par-mnemonics, scrabble, akka-uct, db-shootout, dec-tree, and neo4j-analytics). From Figure 3.3, we can see that DJXPerf typically incurs 8.5% runtime and 6% memory overhead.

3.6 Case Studies

DJXPerf’s low overhead allows us to collect object-centric profiles from a variety of Java and Scala applications, such as the Renaissance benchmark suite [Prokopec et al. 2019], Java Grande Benchmark suite [Bull et al. 2001], Ranklib [Saladi 2013], Cache2k [Wike et al. 2013], Apache SAMOА [Apache 2017], Commons Collections [Apache 2019], NPB [NASA 2018], Apache Druid [Apache 2020], to name a few. We run these applications (sampling period is 5M) with the default inputs released or the inputs that we can find to our best knowledge. To fully utilize CPU resources, we run each parallel application to saturate all CPU cores if not specified.

DJXPerf can pinpoint many previously not reported data locality issues and guide optimization choices. To guarantee correctness, we ensure the optimized code passes the application validation tests. When optimizing memory bloat, our strategy is to allocate a single object instance and reuse it with no need to create new instances. The single
object is declared as a static thread-local object if the Java application is a multi-threaded program. Therefore, our optimization guarantees thread-safety and the ability of multiple threads of the application to access their own copy of this hoisted single object.

To avoid system noises, we run each application 30 times and use a 95% confidence interval for the geometric mean speedup to report the performance improvement, according to the prior approach [Su et al. 2019a]. Guided by DJXPerf, Table 3.1 presents an overview of the performance improvements on several real-world Java applications. All the optimizations shown in Table 3.1 are newly found by DJXPerf.

For case studies that suffer from memory bloat in Table 3.1, our optimization is to move problematic allocation sites out of loop and avoid frequent allocations. With this optimization, we can replace many short-lived objects that have non-overlapping lifetimes with a single long-lived object. Compared with the unoptimized code, we still have the same number of objects that can be used at any given time at the source code level. At the memory level, however, many short-lived objects continue to consume memory (because GC does not immediately claim them) even after they are no longer accessible. Our optimization can reduce the JVM heap memory usage. Assume there are a sequence of objects \{O_1, O_2, O_3, O_4\}, which have the same allocation context and are sorted by the allocation timestamp during program execution. Also, assume the sizes of these objects are \{10, 20, 30, 40\}, respectively. When the object \(O_4\) is allocated, Objects \{O_1, O_2, O_3\} cannot be accessed, but they are still consuming memory. As of now, the total size taken by objects \{O_1, O_2, O_3, O_4\} is 100 in unoptimized code. In contrast, if we allocate one object \(O\) whose size is 40 and reuse it every time, then the total size is 40.

In the remaining section, we elaborate on the analysis and optimization of several applications under the guidance of DJXPerf in Section 3.6.1 - 3.6.14. It is worth noting that Section 3.6.12 - 3.6.14 are related to NUMA locality. To perform the optimization guided by DJXPerf, we leverage Java Native Interface (JNI) to access the native libnuma API to control page placement. Section 3.6.15 shows some studies on optimizing insignificant objects, which illustrate the unique usefulness of DJXPerf over other tools. We have submitted our optimization patches and some of them got upstreamed (i.e., ObjectLayout [Tene et al. 2018] and FindBugs [Pugh et al. 2015]).

We also collected code-centric profiles using the Linux perf utility for each case study to compare with DJXPerf’s profiles. Code-centric profiles cannot detect any of the case studies in this section. In several cases, we found it arduous to tie data accesses segregated over many code locations back to the object allocation sites in code-centric profiles. There are several reasons for this:
• Code-centric profiling can only pinpoint hotspots without indicating the root causes of the hotspot that is due to a memory hierarchy problem. In contrast, object-centric profiling clearly identifies the root causes (aka data objects) of memory inefficiencies.

• In code-centric analysis, at the same code location, multiple objects may be used and it would be non-trivial to pinpoint or quantify the problem to a specific object.

• The same code context can be incurring high cache misses for some objects and not for others, which is invisible in code-centric analysis but visible in object-centric analysis.

DJXPerf eases developer’s task by showing top data objects subject to locality problems along with an ordered, hierarchical view of code locations contributing to the locality problem.

3.6.1 ObjectLayout 1.0.5

ObjectLayout [Tene et al. 2018] is a layout-optimized Java data structure package, which provides efficient data structure implementation. We run ObjectLayout with the SA-HashMap input released with the package. DJXPerf reports four problematic objects, which account for 84% of L1 cache misses in the entire program. Figure 3.4 shows the snapshot of DJXPerf’s GUI for intuitive analysis. The top pane of the GUI shows the Java source code; the bottom left pane shows the object allocation (in red) and accesses (in blue) in their full call paths; and the bottom right pane shows the metrics (e.g., L1 cache misses, object allocation times in this example).

The GUI in Figure 3.4 shows one problematic object, which is at line 292 of method allocateInternalStorage in class AbstractStructuredArrayBase. The allocateInternalStorage method is repeatedly invoked in a loop when a new instance is created by the newInstance method. The bottom left pane of the GUI shows one problematic object allocation in the full call path, which is allocated 217 times in a loop. There are multiple sampled accesses associated with this allocation site, and they account for 30.4% L1 cache misses of the entire program. To save the space, we only show one access in its call path in blue (method getNode in class SAHashMap) that accounts for most of the cache misses. For the other accesses, we only show their call paths rooted at java.lang.Thread.run; with no object-centric profiling, these accesses are separately presented with no aggregate view.

We further investigate the source code and find that the problematic object is array intAddressableElements. The reason for memory bloat is due to inappropriate memory
Figure 3.4: The top-down object-centric GUI view of ObjectLayout shows a problematic object’s allocation site in source code, and its allocation call path with its all access call paths.
Listing 3.3: DaCapo 9.12 lusearch: The hotspot object parser (line 119) which suffers from memory bloat problem.

management. ObjectLayout treats input objects as many small partitions, and allocates array `intAddressableElements` repeatedly in every partition. Since the life cycles of different `intAddressableElements` instances do not overlap, which means using a singleton pattern of this object (i.e., allocating a single object instance and reusing it without creating more instances) is safe and avoids memory bloat. To apply the singleton pattern, we hoist `intAddressableElements` allocation out of `allocateInternalStorage` method, which is thread-safe. We optimize three other problematic objects with similar methods. Our optimization increases L1 cache hits by 76.1%, and the JVM heap memory usage is reduced by 16.6%, yielding a $(1.42 \pm 0.05)\times$ increase in throughput. Our optimization has been merged to the repository.

### 3.6.2 DaCapo 9.12 lusearch

lusearch uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible [Blackburn et al. 2018]. DJXPERF reports a problematic object—the `QueryParser` object—allocated at line 119 in method `parse` of class `QueryParser`, shown in Listing 3.3, which accounts for 36.4% of L1 cache misses. In order to parse a query string into a query object, the allocation site creates a new `QueryParser` object whenever a new query string is generated. As different instances of `QueryParser` object have no overlap in their lifetimes and are never needed simultaneously in the program, we hoist this allocation out of the method `parse` and declare it as a static thread-local object. When a `QueryParser` object is needed again, its content is reset and reused. This optimization reduces the JVM heap memory usage from 84.2MB to 78.5MB (6.7%), and yields a $(1.15 \pm 0.02)\times$ speedup to the entire program execution.

### 3.6.3 FindBugs 3.0.1

FindBugs is a program to find bugs in Java programs. It looks for instances of “bug pattern” — code instances that are likely to be errors [Pugh et al. 2015]. We run FindBugs...
```java
public void setAppClassList(List<ClassDescriptor> appClassCollection) {
    for (ClassDescriptor appClass : allClassDescriptors) {
        XClass xclass = currentXFactory().getXClass(appClass);
        // getXClass method will call to parse method below
    }
}

public void parse(ClassInfo.Builder builder) {
    char[] buf = new char[1024];
    // parse method will call to analyzeMethod method below
}
```

Listing 3.4: The problematic source code highlighted by DJXPERF in FindBugs.

```java
private void analyzeApplication() throws InterruptedException {
    for (Detector2 detector : detectorList) {
        detector.visitClass(classDescriptor);
        // visitClass method will call to analyzeMethod method below
    }
}

private void analyzeMethod(ClassContext classContext, Method method) {
    IdentityHashMap<InstructionHandle, Object> sometimesGood = new IdentityHashMap<InstructionHandle, Object>();
    // analyzeMethod method will call to analyzeApplication method below
}
```

Listing 3.5: The source code highlighted by DJXPERF shows the problematic object `sometimesGood` (allocated at line 120) suffering from poor locality.

on Java chart library 1.0.19 version as input. DJXPERF reports two objects, `buf` and `IdentityHashMap`, that account for 32% of L1 cache misses in the entire program as shown in Listings 3.4 and 3.5.

The memory bloat comes from the algorithm of data-flow analysis used in Findbugs. Findbugs divides a data-flow graph into many tiny-sized blocks and creates objects `buf` and `IdentityHashMap` for every block instead of declaring them as a global object for the entire graph. Consequently, the two problematic objects `buf` and `IdentityHashMap` are both repeatedly allocated in loops with no overlap in lifecycles across different instances. We apply the singleton pattern by hoisting the two object allocations out of the loops to avoid memory bloat. These optimizations reduce JVM heap memory usage from 1.8GB to 0.9GB, yielding a \((1.1 \pm 0.02)\times\) speedup to the entire program. Our optimization has been accepted to the repository.

### 3.6.4 Renaissance 0.10: scala-stm-bench7

Scala-stm-bench7 is a Renaissance [Prokopec et al. 2019] benchmark, which runs stm-bench7 code using ScalaSTM [Stanford 2017] for parallelism. It is written in Scala. We run scala-stm-bench7 using the default 60 repetitions. DJXPERF pinpoints a problem-
private def grow() {
  _wCapacity *= 2
  if (_wCapacity > _wDispatch.length) {
    ▶ _wDispatch = new Array[Int](_wCapacity)
  }
}

Listing 3.6: DJXPERF pinpoints _wDispatch object suffering from high cache misses in scala-stm-bench7.

atic object _wDispatch as shown in Listing 3.6. This object accounts for 25% of L1 cache misses. Updating DJXPERF’s splay tree causes a 6% runtime overhead.

With further investigation, the memory bloat is due to the inappropriate initial size set for object _wDispatch. The method grow is called frequently to adjust the capacity of _wDispatch array and create a new one. Such frequent invocation of grow is because the initial size of _wDispatch array is only 8. For optimization, we increase the initial size of _wDispatch array to be 512, which reduces array creation and copy by 79% and reduces the JVM heap memory usage by 7.6%. This optimization yields a (1.12 ± 0.03)× speedup to the entire program.

3.6.5 Ranklib 2.3

RankLib [Saladi 2013] is a library for comparing different machine learning ranking algorithms. We run RankLib with Million Query track of TREC 2008 (MQ2008). DJXPERF pinpoints three problematic objects, which account for 84.2% L1 cache misses of the entire program. The first problematic object allocation was reported by DJXPERF at line 218 in method rank of class CoorAscent, as shown in listing 3.7. From Listing 3.7 we can find that the array score (line 215) is repeatedly created in a nested loop of learn method. As different instances of score have no overlap in their lifetimes and are never needed simultaneously in the program, it is thread-safe to hoist this allocation out of the loop and declare it as a static object. It is worth noting that our optimization fixes the length of the array score. However, the code later uses score’s length to sort its elements. To ensure correctness, we modify the sort method, which accepts the real length of array score as its argument.

The other two problematic objects are allocated in method sort of class MergeSorter, as shown at lines 220 and 221 in Listing 3.7. We hoist the two array objects idx and tmp out of the loop that encloses sort and refer to them as static objects, which reduces the JVM memory usage from 860MB to 605MB. This optimization yields a (1.27 ± 0.04)×
public void learn() {
...
for (int s = 0; s < sign.length; s++) {
    for (int j = 0; j < numIter; j++) {
        double score = scorer.score(rank(samples));
...
    }
}
public RankList rank(RankList rl) {
    double[] score = new double[rl.size()];
    int[] idx = MergeSorter.sort(score, 0, rl.size(), false);
...
}
public static int[] sort(double[] list, int begin, int end, boolean asc) {
...
int[] idx = new int[len];
int[] tmp = new int[len];
...
return idx;
}

Listing 3.7: The source code related to the problematic object allocation and usage in Ranklib.

private void rehash() {
    Entry<K,V>[] src = entries;
    int i, sl = src.length, n = sl * 2, _mask = n - 1, idx;
    Entry<K,V>[] tab = new Entry[n];
...
entries = tab;
calcMaxFill();
}

Listing 3.8: cache2k: The hotspot object tab (line 313) which suffers from memory bloat problem.

speedup to the entire program execution.

3.6.6 Cache2k 1.2.0

Cache2k [Wike et al. 2013] provides an in-memory object cache implementation for Java applications. We run Cache2k with provided Cache2k JMH benchmarks as input. DJXPERF reports a problematic object allocated at line 313 in method rehash of class tt Hash2 as shown in Listing 3.8. This object—an array of Entry objects that contain key-value pairs—accounts for 85.6% of L1 cache misses. Further investigation shows that the code repeatedly creates Entry object to rehash the entries. Because the Entry object never escapes to the heap, each thread needs only one instance of this data structure at any point during the execution. To address the problem, we create a static Entry array that maintains one Entry object per thread. Also, each time an Entry object is needed, we reset this Entry array. This optimization reduces the JVM memory usage by 5.2% and yields a (1.09 ± 0.02)× speedup.
Listing 3.9: Apache SAMOA: The hotspot object instance (line 165) which suffers from memory bloat problem.

### 3.6.7 Apache SAMOA 0.5.0

Apache SAMOA [Apache 2017] (Scalable Advanced Massive Online Analysis) is a platform for mining big data streams. We run Apache SAMOA with the covtypeNorm.arff dataset as its input. DJXPerf reports a problematic object—`Instance`—allocated at line 165 in method `readInstanceDense` of class `ArffLoader`. This object accounts for 26% of L1 cache misses. With code investigation as shown in Listing 3.9, we find that this `Instance` object is repeatedly formed (line 165) every time the program reads a dense instance from a file, and different instances of this object do not overlap in their life intervals. Thus, we hoist this object outside of the loop and put it into a static location to avoid repeated allocation and. This optimization reduces the JVM memory usage by 14.3% and yields a \((1.1 \pm 0.03)\times\) speedup to the entire program execution.

### 3.6.8 Apache Commons Collections 4.2

Apache Commons Collections [Apache 2019] include many powerful data structures that accelerate the development of the most significant Java applications. We run it using its provided commons collections4 map tests as input. DJXPerf reports a problematic object—the `HashEntry` array—in the constructor of class `AbstractHashedMap`, which is used as a map to store key-value entries. Line 5 of listing 3.10 shows this object allocation, which accounts for 22% of L1 cache misses. Like other cases, different instances of this object have disjoint life intervals, so we change this object to a static one. When using a new HashEntry object, we clear and reuse the static object to avoid repeated allocation upon each iteration. The optimization reduces the JVM memory usage by 3.1% and yields a \((1.07 \pm 0.01)\times\) speedup to the entire program.
protected AbstractHashedMap(int initialCapacity, final float loadFactor) {
    
    this.loadFactor = loadFactor;
    this.threshold = calculateThreshold(initialCapacity, loadFactor);
    
    this.data = new HashEntry[initialCapacity];

    }

Listing 3.10: Apache Commons Collections: The hotspot object data (line 5) which suffers from memory bloat problem.

protected void transform_internal (double data[], int direction) {
    for (int bit = 0, dual = 1; bit < logn; bit++, dual *= 2) {
        for (int a = 1; a < dual; a++) {
            for (int b = 0; b < n; b += 2 * dual) {
                int i = 2*(b + a);
                int j = 2*(b + a + dual);
                double z1_real = data[j];
                double z1_imag = data[j+1];
                ...data[j] = data[i] - wd_real;
                ...data[j+1] = data[i+1] - wd_imag;
                ...
            }
        }
    }
}

Listing 3.11: DJXPERF identifies the data array with poor locality in SPECjvm2008: Scimark.fft.large.

3.6.9 SPECjvm2008: Scimark.fft.large

Scimark [SPEC 2004] is a composite Java benchmark measuring the performance of numerical codes occurring in scientific applications. Scimark.fft refers to a fast Fourier transform (FFT) implementation. We run Scimark.fft with its large input released with the benchmark. DJXPERF reports the object data, which is an array, suffering the most from cache miss (85.3% of L1 cache misses). The most problematic accesses are at lines 171, 172, 174, and 175 in method transform_internal of class FFT, as shown in Listing 3.11.

From the code listing, we can see that array data is accessed in a 3-level loop nest. The innermost loop index b increases by 2*dual every iteration, and dual is also doubled every iteration of the outer-most loop. Thus, the accessing stride for data array is large, resulting in the poor spatial locality. For optimization, we interchange loops a (line 167) and b (line 168) to reduce the stride. This optimization increases L1 cache hits by 81%, yielding a \((2.31 \pm 0.07)\times\) speedup.
Listed 3.12: DJXPERF pinpoints arrays \texttt{pathValue} and \texttt{operandPath} (accessed at line 14) suffering from high cache misses in JGFMonteCarloBench.

### 3.6.10 Java Grande 2.0: JGFMonteCarloBench

JGFMonteCarloBench uses Monte Carlo techniques to price products derived from the price of an underlying asset [Bull et al. 2001]. We run JGFMonteCarloBench using input size B. DJXPERF reports objects—arrays \texttt{pathValue} and \texttt{operandPath}—suffer the most from cache misses: they account for 80% of L1 cache misses. Listing 3.12 shows the problematic code (line 14), which iterates every element in arrays \texttt{pathValue} and \texttt{operandPath} in the \texttt{inc\_pathValue} method. We find the \texttt{inc\_pathValue} method is called \texttt{nRunsMC} times in a loop (line 4) in method \texttt{processResults} of class \texttt{AppDemo}. The streaming access pattern on the two arrays across iterations shows up as poor temporal locality. We tile the loops by blocking arrays \texttt{pathValue} and \texttt{operandPath} to ensure the accesses to the two arrays are in the cache across the loop at line 4. This optimization reduces L1 cache misses by 16.7%, yielding a \((1.07 \pm 0.02)\times\) speedup to the entire program execution.

### 3.6.11 Java Grande 2.0: JGFMolDynBench

JGFMolDynBench is a simple N-body code modeling the behavior of N argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions [Bull et al. 2001]. We run JGFMolDynBench with input size B. DJXPERF reports the array \texttt{md\_one} accounts for 83.8% of L1 cache misses and the problematic data accesses are highlighted in the method \texttt{force} of the class \texttt{md}, as shown at lines 4, 5, and 6 of Listing 3.13. The method \texttt{force} is called multiple times in a loop (not shown). The loop in method \texttt{force} (line 3) has a streaming access pattern on array...
public void force(double side, double rcoff, int mdsize) {
...
for (int i=0; i<mdsize; i++) {
  ▶ xx = xi - md.one[i].xcoord;
  ▶ yy = yi - md.one[i].ycoord;
  ▶ zz = zi - md.one[i].zcoord;
...
}

Listing 3.13: JGFMolDynBench: The hotspot array md.one (line 4, 5 and 6) which suffer from high cache miss.

md.one, which shows up as poor temporal locality. We tile the loop for a better temporal locality, which reduces L1 cache misses by 53.2% and yields a (1.24 ± 0.05) × speedup on the entire program.

3.6.12 Eclipse Collections

Eclipse Collections is a comprehensive Java collections library, which enables productivity and performance by delivering an expressive and efficient set of APIs and types [Eclipse-Foundation 2019]. We run eclipse collections using CollectTest as input. After profiling eclipse collections, DJXPERF reports an object that suffers from NUMA remote accesses, the integer array result, which is allocated at line 237 in method toArray of class Interval and accessed at line 245 in method batchFastListCollect of class InternalArrayIterate with a high percentage of NUMA remote accesses (73.4%).

By investigating the source code as shown in Listing 3.14, the program separates a number of operations into intervals using each operation’s index and stores these intervals in the array result. The program then passes result to batchFastListCollect method to process each interval. The large amount of remote accesses is because the master thread allocates and initializes of result, resulting in the array placed in one NUMA domain, but all threads from other NUMA domains access it. As a result, all threads compete for the bandwidth of a single NUMA domain. To avoid the contention, we optimized the program by co-locating result with computation via libnuma APIs. This optimization reduces remote accesses by 41% and increases the throughput (operations per second) by (1.14 ± 0.03) ×.

3.6.13 NPB: SP

NPB is the NAS Parallel Benchmarks, which are a set of programs designed to help evaluate the performance of parallel supercomputers [NASA 2018]. Problem sizes in NPB
Listing 3.14: DJXPERF pinpoints the array array suffering from NUMA remote accesses in Eclipse Collections.

are predefined and indicated as different classes, and we run SP using Class C (largest input of standard test) as input. After profiling SP, DJXPERF reports an object, array rhs, which is allocated at line 155 in constructor method SPBase and accessed in the computational method step of class XSolver, class YSolver and class ZSolver, as shown in Listing 3.15.

Based on the call path analysis, the program first comes to the constructor method SPBase to allocate rhs that will be used in the computation process by the master thread. And then, rhs is initialized in the init method in class XSolver, class YSolver and class ZSolver by the master thread respectively. After the preparation work that allocates and initializes for object rhs in the master thread, the method setupThreads is invoked to create and start many worker threads to do the computations in step method. Because the object rhs is allocated and initialized by the master thread, so all worker threads access it remotely. We address this problem by initializing rhs by leveraging the Linux “first touch” policy. Allocating and initializing rhs in parallel allocates parts of it near each thread using it. The optimization reduces remote accesses by 35% and yields a (1.11 ± 0.01) × speedup to the entire program.

3.6.14 Apache Druid

Apache Druid is a high performance real-time analytics database designed for workflows where fast queries and ingest matter [Apache 2020]. We run Apache Druid using BitmapIterationBenchmark as input. DJXPERF reports an array of BitSet object, bitmap, which is initialized in constructor method WrappedImmutableBitSetBitmap of class WrappedImmutableBitSetBitmap and accessed at line 120 in method next in the
Listing 3.15: DJXPERF pinpoints the \texttt{rhs} array suffering from NUMA remote access in NPB SP.

same class. Listing 3.16 shows the problematic object is the \texttt{bitmap} (accessed at line 120), of which more than half of total memory accesses are remote accesses. With further investigation, we find that the \texttt{bitmap} object is allocated in one NUMA domain but accessed by threads from other NUMA domains. To address this problem, we let each thread initializes their own portion of \texttt{bitmap} to ensure the data are co-located with the computation via Linux first touch policy. With this optimization, we reduce remote accesses by 47\% and increase the throughput by \((1.73 \pm 0.06) \times \).

Listing 3.16: DJXPERF pinpoints the BitSet object \texttt{bitmap} suffering from NUMA remote accesses in Apache Druid.
3.6.15 Attempts to Optimization for Insignificant Objects

To demonstrate the importance of PMU metrics (i.e., cache misses in our experiments) associated with the objects, we show a number of studies on attempting to optimize insignificant objects in Table 3.2. All these code bases have the memory bloat problem: repeatedly allocate objects many times, and different instances have no overlap in their life intervals. The table shows the location of problematic object allocations, the number of object instances, the associated cache miss metrics, and the speedups after optimization. Our studies show that these optimizations yield negligible speedups, emphasizing that these objects account for very few cache misses. Thus, DJXPERF’s object-centric analysis is helpful to filter out insignificant objects for non-fruitful optimization.

### Table 3.2: Optimizing insignificant objects yields little speedups.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Problematic Code</th>
<th>Allocation times</th>
<th>L1 Cache Misses (%)</th>
<th>WS (×)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB 3.0 SP</td>
<td>SP.java (2086)</td>
<td>400</td>
<td>&lt;1%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Dacapo 2006 chart</td>
<td>Datasets.java (397, 408)</td>
<td>3760</td>
<td>&lt;1%</td>
<td>1%</td>
</tr>
<tr>
<td>Dacapo 2006 antlr</td>
<td>Preprocessor.java (564)</td>
<td>2840</td>
<td>0%</td>
<td>1%</td>
</tr>
<tr>
<td>Dacapo 2006 luindex</td>
<td>DocumentWriter.java (206)</td>
<td>3035</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Dacapo 9.12 lsearch</td>
<td>IndexSearcher.java (498)</td>
<td>15179</td>
<td>&lt;1%</td>
<td>0%</td>
</tr>
<tr>
<td>Dacapo 9.12 lsearch-fix</td>
<td>FastCharStream.java (54)</td>
<td>225060</td>
<td>&lt;1%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Dacapo 9.12 batik</td>
<td>ExtendedGeneralPath.java (743)</td>
<td>2470</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>SPECjbb2000</td>
<td>StockLevelTransaction.java (173)</td>
<td>116376</td>
<td>&lt;1%</td>
<td>1%</td>
</tr>
<tr>
<td>JGFMonteCarloBench 2.0</td>
<td>RatePath.java (296)</td>
<td>60000</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

WS: Whole-program speedup after optimization

3.7 Summary

In this work, we design and implement DJXPERF, a lightweight Java profiler that first performs object-centric analysis to identify data locality issues in Java applications. DJXPERF leverages the lightweight Java byte code instrumentation and the hardware PMU available in commodity CPU processors. DJXPERF works for off-the-shelf Linux OS and Oracle Hotspot JVM, as well as unmodified Java applications. DJXPERF incurs low overhead, typically 8.5% in runtime and 6% in memory. These features make DJXPERF applicable to the production environment. DJXPERF is able to identify a number of locality issues in real-world Java applications. Such locality issues arise due to traditional spatial/temporal data locality, memory bloat, as well as NUMA locality. Guided by DJXPERF, we are able to perform optimization, which yields nontrivial speedups.
Chapter 4

OJXPerf: Featherlight Object Replica Detection for Java Programs

4.1 Introduction

Memory bloat is a common problem in managed languages, such as Java and C#. The problem is particularly severe in large, production software, which employs layers of abstractions, third-party libraries, and evolves over time into complex systems not comprehendible by any single developer. Furthermore, these programs run for a long time (several months at a time), giving them an opportunity to grow their memory footprint and become a source of major problems in production environments often shared by several other programs.

An object that is not reclaimed by the garbage collector (GC) but neither read nor written any more is considered to be “leaked”. A memory leak happens in managed languages because useless objects remain unreclaimed by the GC because of unnecessary references to them. Additionally, memory spikes occur in managed languages because of accumulated objects that are yet to be garbage collected. Memory bloat (whether due to leaks or GC) results in high memory pressure and poor performance. A lot of prior work exits to detect object leaks [Fang et al. 2015; Nguyen et al. 2013; Xu 2013; Xu et al. 2009b, 2011a, 2014, 2010c, 2012; Yan et al. 2014] and improve GC [Adl-Tabatabai et al. 2004; Chen et al. 2006; Huang et al. 2004; M. Chilimbi et al. 1998; Shuf et al. 2002; Yang et al. 2020].

However, there is another cause of memory bloat and inefficiency that has hardly been studied — replica objects — which is the focus of this work. Two objects are...
replicas if their contents are identical. Two objects are shallow replicas if their fields are bitwise identical; and deep replicas if the transitive closure of the constituent objects and their respective fields values are bitwise identical. When two objects are bitwise identical (shallow replicas), their transitive closures are also the same. However, when two objects are not bitwise identical and the difference occurs on one or more fields that are object references, it becomes necessary to chase those references to disprove that the contents of those objects are not identical. After chasing all object references, if we can prove that their contents are identical, then such two objects are deep replicas.

There is a temporal aspect to replica objects: one may regard two objects as replicas either because they were identical at the time (or a window) of observation or for their entire lifetime. Two objects are mutable replicas if they are identical to each other, but they may independently evolve during their lifetimes; whereas two objects are immutable replicas if they are identical to each other and are also immutable during their lifetimes.

There are two performance dimensions to replica objects: total memory consumption and total number of memory accesses. On the one hand, an object may be large in size and replicated only a few times, and such replicas still contribute to the overall memory bloat; also, an object may be small in size but replicated many times, which also contributes to memory bloat; both these cases are worth optimizing to remove the replicas. On the other hand, an object may be small in size and replicated only a few times, but the program may be accessing these few replicated objects a lot of times; although this situation is not memory bloat, it can still have a significant consequence to the overall performance since it increases the memory footprint at the CPU cache level, squanders potential memory reuse [Allen et al. 2001], and often results in redundant re-computations [Nguyen et al. 2013; Wen et al. 2017b].

Having described the landscape of object replication (deep vs. shallow, some time window vs. full lifespan, mutable vs. immutable, and memory size vs. access counts), we now scope this problem to a tractable subset driven by pragmatic tool-development factors. First, instrumenting every allocation and memory access to identify object replicas leads to excessive runtime overheads; we seek for a lightweight tool that can collect profiles in production rather than in test-only environments; we guarantee the analysis accuracy with the theoretical bounds of a sampling technique we use. Second, deep replica comparison is unachievable without running something analogous to the garbage collector, which can introduce high overheads and require runtime modifications, making it less adaptable; hence we restrict our tool to only shallow object comparison. Third, if two objects are not replicas for their entire life span, they are not easy to optimize,
Table 1

<table>
<thead>
<tr>
<th>Library</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>agrona</td>
<td>5.4%</td>
</tr>
<tr>
<td>akarnokd-misc</td>
<td>8.3%</td>
</tr>
<tr>
<td>apache SAMOA</td>
<td>21.1%</td>
</tr>
<tr>
<td>cache2k</td>
<td>17.7%</td>
</tr>
<tr>
<td>cactoos</td>
<td>5.6%</td>
</tr>
<tr>
<td>deeplearning4j</td>
<td>12.8%</td>
</tr>
<tr>
<td>fastutil</td>
<td>0%</td>
</tr>
<tr>
<td>findbugs</td>
<td>26.3%</td>
</tr>
<tr>
<td>graphchi-java</td>
<td>11.1%</td>
</tr>
<tr>
<td>groovy</td>
<td>10.5%</td>
</tr>
<tr>
<td>hppc</td>
<td>3.3%</td>
</tr>
<tr>
<td>jctools</td>
<td>4.3%</td>
</tr>
<tr>
<td>jfreechart</td>
<td>9.3%</td>
</tr>
<tr>
<td>JGFSerialBench</td>
<td>15.9%</td>
</tr>
<tr>
<td>mallet</td>
<td>0%</td>
</tr>
<tr>
<td>parquet mr</td>
<td>22.4%</td>
</tr>
<tr>
<td>pdfbox</td>
<td>2.8%</td>
</tr>
<tr>
<td>ranklib</td>
<td>13.9%</td>
</tr>
<tr>
<td>reactive-grpc</td>
<td>10%</td>
</tr>
<tr>
<td>roaringbitmap</td>
<td>19.5%</td>
</tr>
<tr>
<td>rxjava</td>
<td>6.2%</td>
</tr>
<tr>
<td>soot</td>
<td>28.5%</td>
</tr>
<tr>
<td>avrora</td>
<td>0%</td>
</tr>
<tr>
<td>batik</td>
<td>3.8%</td>
</tr>
<tr>
<td>eclipse</td>
<td>20.9%</td>
</tr>
<tr>
<td>h2</td>
<td>5.4%</td>
</tr>
<tr>
<td>iython</td>
<td>0%</td>
</tr>
<tr>
<td>luindex</td>
<td>12.3%</td>
</tr>
<tr>
<td>lusearch</td>
<td>15.4%</td>
</tr>
<tr>
<td>lusearch-fix</td>
<td>13.5%</td>
</tr>
<tr>
<td>tradebeans</td>
<td>5.5%</td>
</tr>
<tr>
<td>tradesoap</td>
<td>9.1%</td>
</tr>
<tr>
<td>sunflow</td>
<td>1.1%</td>
</tr>
<tr>
<td>xalan</td>
<td>0%</td>
</tr>
<tr>
<td>akka-uct</td>
<td>0%</td>
</tr>
<tr>
<td>als</td>
<td>3.2%</td>
</tr>
<tr>
<td>chi-square</td>
<td>8.4%</td>
</tr>
<tr>
<td>db-shootout</td>
<td>12.5%</td>
</tr>
<tr>
<td>dec-tree</td>
<td>0%</td>
</tr>
<tr>
<td>dotty</td>
<td>9.2%</td>
</tr>
<tr>
<td>finagle-chirper</td>
<td>3.3%</td>
</tr>
<tr>
<td>finagle-http</td>
<td>3.8%</td>
</tr>
<tr>
<td>fj-kmeans</td>
<td>17.2%</td>
</tr>
<tr>
<td>future-genetic</td>
<td>6.5%</td>
</tr>
<tr>
<td>gauss-mix</td>
<td>15.8%</td>
</tr>
<tr>
<td>log-regression</td>
<td>20.3%</td>
</tr>
<tr>
<td>mnemonics</td>
<td>0%</td>
</tr>
<tr>
<td>movie-lens</td>
<td>11.2%</td>
</tr>
<tr>
<td>naive-bayes</td>
<td>3.4%</td>
</tr>
<tr>
<td>neo4j-analytics</td>
<td>0%</td>
</tr>
<tr>
<td>page-rank</td>
<td>18.9%</td>
</tr>
<tr>
<td>par-mnemonics</td>
<td>0%</td>
</tr>
<tr>
<td>philosophers</td>
<td>10.7%</td>
</tr>
<tr>
<td>reactors</td>
<td>2.8%</td>
</tr>
<tr>
<td>rx-scrabble</td>
<td>6.1%</td>
</tr>
<tr>
<td>scala-doku</td>
<td>1.9%</td>
</tr>
<tr>
<td>scala-kmeans</td>
<td>17.1%</td>
</tr>
<tr>
<td>scala-stm-bench7</td>
<td>3.7%</td>
</tr>
<tr>
<td>scrabble</td>
<td>3.1%</td>
</tr>
</tbody>
</table>

Figure 4.1: Percentage of replicated objects over all the objects in various Java applications.

Listing 4.1: Object replicas in Parquet MR. The replica object `bytes` is allocated on line 93, initialized on line 94 and used on line 96.

```java
private void readNext() {
    ...
    currentBuffer = new int[currentCount];
    byte[] bytes = new byte[numGroups * bitWidth];
    new DataInputStream(in).readFully(bytes);
    for (int valueIndex = 0, byteIndex = 0; ...; ...) {
        packer.unpack8Values(bytes, byteIndex, currentBuffer, valueIndex);
    }
}
```

and hence we consider only those objects that are replicas for their entire duration. We do not enforce objects to be immutable for replica detection. Finally, our tool regards two or more objects as replicas only if they were allocated in the same calling context; the observation drives this restriction that it is significantly easy to refactor such code to optimize compared with optimizing replica objects allocated from myriad code locations. We emphasize that we want to be able to monitor replicas and prioritize them by their access frequency.

OJXPERF, developed to meet these factors, monitors object allocations and accesses at runtime via statistical sampling. The key differentiating aspect of OJXPERF, compared to a large class of existing profilers, is its ability to detect object replicas with minimal byte code instrumentation and no prior knowledge of programs makes it applicable in the production environment. A thorough evaluation of several real-world applications shows that pinpointing object replicas offers new avenues to understanding performance
losses; aggregating replica objects into one or a few objects reduces the memory footprint, eliminates redundant computations, and enhances performance.

4.1.1 Observation

With the help of OJXPerf, Figure 4.1 quantifies the ratio of object replicas over the total number of objects in several real applications listed at [Kull 2020] and two popular Java benchmark suites—Dacapo 9.12 [Blackburn et al. 2018] and Renaissance [Prokopec et al. 2019], showing that replicated objects are pervasive in modern Java software packages. Based on many case studies investigated in this thesis, we observe that object replication is the symptom of the following kinds of inefficiencies.

**Input-sensitive Inefficiencies.** Repeatedly using the same input to instantiate a Java class shows up as repeatedly creating objects with the same content. Listing 4.1 shows a problematic method readNext from Parquet-MR [Twitter 2019], which contains the Java implementation of the Parquet format. This method is invoked in a loop, and in each invocation, it creates a new object bytes, shown on line 93, and initializes this object via input stream “in”, as shown on line 94. We run Parquet-MR using Parquet-Column as its input. The Parquet-Column input is a columnar storage format for Hadoop; this format provides efficient storage and encoding of data. As line 94 in Listing 1 shows, each time the readNext method is invoked, it creates a copy of input contents (variable “in”), and uses this copy to initialize many objects “bytes” (object replication). None of the existing profilers, such as JXPerf [Su et al. 2019a] and LDoctor [Song et al. 2017], can identify such object replicas since they are designed only to recognize the redundancies happening at the same memory location. Instead, objects bytes are allocated in disjoint memory regions.

**Algorithmic Inefficiencies.** Suboptimal choices of an algorithm often show up as object duplications. As a practical example, Findbugs [Pugh et al. 2015] divides a graph into tiny-size blocks and creates an object for each block instead of creating a single object for the whole graph. Consequently, most of the created objects have the same content due to good value locality among adjacent blocks.

**Data Structural Inefficiencies.** Like suboptimal algorithms, poor data structures can easily introduce object replicas as well. For instance, in matrix multiplication, sparse
matrices with a dense format can yield a high proportion of objects with the same content.

The major lessons that can be learned from this observation section:

• Object replicas are not uncommon in real Java applications.

• Sampling-based measurement based on hardware counters and debug registers can provide good insights and incur significant low overhead.

• Developing OJXPerf that efficiently interacts with off-the-shelf JVM and Linux OS in the production environment requires careful design.

• The call path of object allocation and source code attribution in a GUI are particularly useful for users to identify actionable optimization.

4.1.2 Contribution Summary

In this thesis, OJXPerf makes the following contributions:

• Develops a novel object-centric profiling technique. It provides rich information to guide optimizing object replicas in JVM-based programs, such as Java and Scala.

• Employs PMU in conjunction with hardware debug registers and minimal byte code instrumentation, which typically incurs 9% runtime and 6% memory overheads.

• Quantifies the theoretical lower and upper bounds of replication ratios of OJXPerf’s statistical approach.

• Applies to unmodified Java (and languages based on JVM, e.g., Scala) applications, the off-the-shelf Java virtual machine, and Linux operating system, running on commodity CPU processors, which can be directly deployed in the production environment.

• Provides intuitive optimization guidance for developers. We evaluate OJXPerf with popular Java benchmarks, such as Dacapo [M. Blackburn et al. 2006], NPB [NASA 2018], Grande [Bull et al. 2001], SPECjvm2008 [SPEC 2008], and the most recent Renaissance [Prokopec et al. 2019] and more than 20 real-world applications. Guided by OJXPerf, we are able to obtain significant speedups by eliminating object replicas in various Java programs. We have upstreamed some of the patches to the software repositories.
Table 4.1: Comparing OJXPerf with other state-of-the-art inefficiency analysis tools/ap-proaches.

<table>
<thead>
<tr>
<th>Tools</th>
<th>Redundancy detection</th>
<th>Cross-object analysis</th>
<th>Probabilistic analysis</th>
<th>Sampling with hardware support</th>
<th>Runtime overhead**</th>
</tr>
</thead>
<tbody>
<tr>
<td>OJXPerf</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>9%</td>
</tr>
<tr>
<td>OEP</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
<td>✓</td>
<td>202X</td>
</tr>
<tr>
<td>Cachetor</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>133X</td>
</tr>
<tr>
<td>Toddler</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>16X</td>
</tr>
<tr>
<td>JOLT</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>3%</td>
</tr>
<tr>
<td>Memoizelt</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>202X</td>
</tr>
<tr>
<td>Ldoctor</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>5%</td>
</tr>
</tbody>
</table>

*OEP identifies mergeability of live objects while OJXPerf pinpoints object replicas regardless of their liveness.
**We obtain the average overhead described in the paper of these tools.

4.2 Related Work

Performance profiling techniques abound in the Java community, which fall into two categories: hardware and software approaches. Each category can be further classified into hotspot and inefficiency analyses. We also compare the related Java tools in table 4.1.

4.2.1 Software Approaches

**Hotspot Analysis.** Netbeans Profiler [Oracle 2016], JProfiler [ej-technologies GmbH. 2018], YourKit [GmbH. 2018], VisualVM [Oracle 2018], and Oracle Developer Studio Performance Analyzer [Oracle 2017] are hotspot analysis profilers, which identify execution hotspots in CPU time or memory usage. They typically introduce negligible overhead by leveraging OS timers as the sampling engines to deliver periodic samples. The hotspot analysis is indispensable but fails to tell whether a resource is being used in a productive manner and contributes to a program’s overall efficiencies. A hotspot does not need to be an inefficient code region and vice versa. Hence, a heavy burden is on users to make a judgment on whether the reported hotspots are actionable.

et al. 2017] applies a static-dynamic analysis to reduce Toddler’s overhead. However, it identifies inefficiencies within a small number of suspicious loops instead of the entire program. [Xu et al. 2009a] introduce copy profiling that optimizes data copies to remove the objects that carry copied values, and the method calls that allocate and populate these objects. Their follow-up work [Xu et al. 2010b] develops practical static and dynamic analyses that identify inefficiently-used containers, such as overpopulated containers and underutilized containers. They also present a run-time technique [Xu 2012] to identify reusable data structures to avoid frequent object allocations. OEP [Marinov et al. 2003] identifies mergeability among live objects, which requires the measurement of object reachability. In contrast, OJXPerf analyzes objects allocated in the same call path regardless of their liveness. OEP leverages bytecode instrumentation, which incurs orders of magnitude of overhead compared to OJXPerf.

OJXPerf is a profiler but applies a hardware approach to address a different inefficiency problem — object replication.

4.2.2 Hardware Approaches

There are many hardware-assisted profilers. In this thesis, we review only PMU- or debug register-assisted Java profilers.

Hotspot Analysis. Linux Perf [Linux 2015], Async-profiler [Pangin 2018], and Oprofile [Cohen et al. 2018] employ PMU as the sampling engines to deliver periodic samples. PMU-based hotspot profilers offer slightly better intuition than the OS timer-based ones since they can classify hotspots according to various forms of performance metrics collected from PMU, such as instruction numbers, cache misses, bandwidth, and many others. However, they are not panaceas; users still have to distinguish inefficient hotspots from efficient ones manually.

Inefficiency Analysis. [F. Sweeney et al. 2004] develop a system that provides a graphical interface to alleviate the difficulty in interpreting PMU results. [Hauswirth et al. 2004] present vertical profiling that captures and correlates performance problems across multiple execution layers (application, VM, OS, and hardware). [Georges et al. 2004] study methods exhibit similar and dissimilar behaviors by measuring the execution time for each method invocation using PMU. [Lau et al. 2006] present a technique that allows a VM to determine whether an optimization improved or degraded by measuring
CPU cycles. Remix [Eizenberg et al. 2016] employs PMU to identify inter-thread false sharing on the fly. JXPerf [Su et al. 2019a] detects redundant memory operations by using PMU to sample memory locations accessed by a program and using debug registers to monitor subsequent accesses to the same location.

Orthogonal to the aforementioned inefficiency analysis profilers, OJXPerf addresses a different inefficiency problem with a different usage of PMU and debug registers. To the best of our knowledge, OJXPerf is the first lightweight sampling-based profiler to pinpoint object replicas in Java.

4.3 Methodology

As previously alluded, we restrict the definition of object replicas to those allocated in the same calling context.

Definition. Object Replicas: $O_1$ and $O_2$ are two objects that have the same allocation context. If the contents of $O_1$ and $O_2$ are identical, $O_1$ is a replica of $O_2$ and $\langle O_1, O_2 \rangle$ is an object replication pair.

A straightforward detection approach is monitoring every allocation context and comparing all fields of all object instances created at that allocation context at any use points. However, performing such whole-heap object tracing can introduce a prohibitively high overhead (70 $\sim$ 300× slowdown reported in [Hertz et al. 2006]).

Instead of exhaustive duplication detection, OJXPerf takes advantage of sampling to perform lightweight replica detection. We neither compare all objects allocated in the same context nor compare all fields when comparing two objects. Instead, our algorithm chooses random fields (offsets in terms of memory locations) at random points.

Example 1 shows an object replica detection example that keeps allocating an object $O$ inside a while loop. As the while loop iterates 4 times, the program allocates a sequence of objects $\{O_1, O_2, O_3, O_4\}$, which have the same allocation context and are sorted by the allocation timestamp during program execution. First, in each loop iteration, we intercept the allocation of the object on line 3. This interception offers two pieces of information: 1) the calling context of allocation, and 2) the memory address range occupied by each object. We maintain this information for future use. For all objects allocated on line 3 in this example, the context is the same but the object addresses can be different1.

\(^{1}\)two objects of different size, e.g., arrays allocated in the same context are easily ruled out of duplication due to size difference
Second, when the program accesses an object (use point), e.g., lines 5 and 7, we can obtain the effective address of the access and map the address to the object it belongs to; furthermore, we can easily derive the relative offset of the access from the start address of the object; this relative offset guides us where to monitor another object allocated in the same context. Third, when the program accesses an object, we can read the contents of the location accessed.

---

**Example 1:** Example of Object Replica Detection.

```plaintext
1    i = 0;
2    while i < 4 do
3        allocate an object O;
4        initialize O;
5        use O;    //use O for the first time
6        update O;
7        use O;    //use O for the second time
8        i++;
9    end
```

Given the nature of sampling, let’s assume that memory-access samples occur on line 5 in iteration 1 of the loop (while accessing object \(O_1\)) and line 7 in iteration 3 (while accessing object \(O_3\)), as shown in Figure 4.2. Assume, the sample in iteration 1 for \(O_1\) on line 5 happens at the relative offset \(Off_1\) from the beginning of the object and the value at \(Off_1\) is \(V_1\); OJXPERF remembers the triple — line 5, \(Off_1\), and \(V_1\) — for future use. For the next allocation, \(O_2\), we probabilistically skip and do nothing. When \(O_3\) starts to be accessed, we decide to monitor its contents at offset \(Off_1\), and hence arm a watchpoint to trap on access to offset \(Off_1\) from the beginning of \(O_3\). This watchpoint traps when the program accesses \(O_3\) on line 5. Let the contents at \(O_3 + Off_1\) be \(V'_1\) when the trap happens. We compare \(V_1\) and \(V'_1\) and if they are the same, \(O_1\) and \(O_3\) contribute towards the number of equivalent objects allocated in context line 3; otherwise they contribute towards non-equivalent objects allocated in context line 3.

The next sample happens on line 7, for the same object \(O_3\) at offset \(Off_2\) in iteration 3 of the loop. Let the value at \(Off_2\) for \(O_3\) be \(V_2\). We remember the triple — line 7, \(Off_2\), and \(V_2\) — for future use. When \(O_4\) starts to be accessed in iteration 4 of the loop, we arm a watchpoint at address \(Off_2\) from the beginning of \(O_4\). This watchpoint traps when the
Figure 4.2: Watchpoint scheme for object replica detection. $Off_1$ ($Off_2$) presents memory offset with value $V_1$ ($V_2$) for Object $O_1$ (Object $O_3$). When a watchpoint trap of memory access happens at offset $Off_1$ ($Off_2$), OJXPerf compares their corresponding values $V_1$ and $V_1'$ ($V_2$ and $V_2'$).

program accesses $O_4$ on line 7. Let the value at the trapped location be $V_2'$. As before, depending on whether $V_2$ and $V_2'$ are the same or not, they contribute towards equivalent or non-equivalent objects allocated in context line 3. Figure 4.2 shows that $V_1$ equals to $V_1'$ (the blue star), which means that the values stored in an offset $Off_1$ of $O_1$ and $O_3$ are the same. Also, the red star in Figure 4.2 shows that the values stored in an offset $Off_2$ of $O_3$ and $O_4$ are different ($V_2$ doesn’t equal to $V_2'$), which means that $O_3$ and $O_4$ must be two objects that have different contents.

As the program continues, OJXPerf performs the same redundancy checks for other samples taken from objects \{$O_1, O_2, O_3, O_4$\}. Finally, if most of the comparisons (> 60%, obtained from our experiments) report identical values among all detection pairs, we believe objects \{$O_1, O_2, O_3, O_4$\} suffer from object replicas with a high probability, which is quantified with our theoretical analysis in Section 4.5.
4.4 Implementation

OJXPerf is a user-space tool with no need for any privileged system permission. OJXPerf requires no modification to hardware, OS, JVM, and monitoring applications, making it applicable to the production environment. Conceptually, OJXPerf consists of two components: data-centric analysis and duplication detection. These two components are implemented within two agents: a Java agent and a JVMTI agent. The Java agent instruments Java byte code execution to obtain each object’s memory interval and allocation context. The JVMTI agent subscribes to Java thread creation to enable PMU. Upon each PMU sample, OJXPerf obtains the effective address of the monitored memory access and associates it with the Java object enclosing this address. Moreover, to identify object replicas, the JVMTI agent programs the debug registers to subscribe to watchpoints.

4.4.1 Online Data Collector

Implementing Sampling with PMU. The JVMTI agent leverages PMU to sample memory accesses. It subscribes to MEM_UOPS_RETIRED:ALL_LOAD, a PMU precise event to sample memory loads. We empirically choose a sampling period to ensure OJXPerf can collect 20-200 samples per second per thread, which yields a fair tradeoff between runtime overhead and statistical accuracy [Tallent 2010]. Moreover, the JVMTI agent captures the calling contexts for both PMU samples and object allocations. To minimize synchronization, each thread collects PMU samples independently and maintains a thread-local compact calling context tree (CCT) [Arnold et al. 1999], which stores the calling contexts of PMU samples and merges all the common prefixes of given calling contexts.

Examining Object Contents with Watchpoints. OJXPerf leverages debug registers to set up watchpoints, which traps the program execution when the designated memory addresses are accessed. Assume $O_1$ and $O_2$ are two distinct objects that have the same allocation context and $O_1$ is created prior to $O_2$. Moreover, $O_1$ and $O_2$ have the same accessing context, then $O_1$ and $O_2$ form a pair $(O_1, O_2)$ as object replicas. OJXPerf uses queues $Q_1$ and $Q_2$ to store samples taken from objects $O_1$ and $O_2$, respectively. Upon a sample taken from $O_1$, OJXPerf uses a tuple $(Off, V)$ to represent it and adds this tuple to queue $Q_1$, as shown in Figure 4.3a. $Off$ is the offset between the sampled address (i.e., the address of the PMU sample) and the starting address of $O_1$, and $V$ is the value
(a) The workflow of object replica detection: collecting PMU samples taken from object $O_1$.

(b) The workflow of object replica detection: setting up watchpoint at object $O_2$.

Figure 4.3: Workflow of object-level redundancy detection.
stored at the sampled memory address. Upon a sample taken from $O_2$, OJXPerf not only adds a tuple $(\text{Off}_m, V_m)$ to queue $Q_2$, but also retrieves a sample $(\langle \text{Off}_n, V_n \rangle)$ from queue $Q_1$ and uses a debug register to set up a watchpoint at the offset $\text{Off}_n$ of $O_2$, as shown in Figure 4.3b. OJXPerf compares values at the the same offset $\text{Off}_n$ of $O_1$ and $O_2$ when the watchpoint is triggered. Watchpoint can be removed when it is triggered, and watchpoints are used for a single access.

**Limited Number of Debug Registers.** Hardware offers only a small number of debug registers, which becomes a limitation if the PMU delivers a new sample, but all watchpoints are armed with addresses obtained from prior samples. OJXPerf employs a reservoir sampling strategy [S. Vitter 1985], which uniformly chooses between old and new samples with no bias. The basic idea of reservoir sampling is to assign a probability to each debug register and perform a replacement policy based on the probability. Prior work [Wang et al. 2019; Wen et al. 2018] has shown that reservoir sampling guarantees the fairness of the measurement with a limited number of debug registers.

4.4.2 Offline Data Analyzer and GUI

To generate a compact profile, which is essential for analyzing a large-scale execution, the offline data analyzer merges profiles from different threads. Object allocation call paths coalesce across threads in a top-down way if they are identical. All memory accesses with their call paths to the same objects are merged as well. Metrics are also summed up when call paths coalesce. The offline procedure typically takes less than one minute in our experiments. Furthermore, OJXPerf integrates its analysis visualization in Microsoft Visual Studio Code, which is shown in Figure 4.5.

4.4.3 Discussions

As a sampling-based approach, OJXPerf may introduce false positives and false negatives, which are elaborated in Section 4.6.1. Our theoretical analysis to be described in the next section bounds the analysis accuracy.
4.5 Theoretical Analysis

Since OJXPERF does not exhaustively check every field of an object for replication due to the sampling, we compute the lower and upper bounds of the analysis to quantify the replication factor.

Definition. Replication Factor (RF) $\theta$: For a set of objects that suffer from object replication, the replication factor $\theta$ is the probability of last accessed object to be bit-wise same as the current accessed object. We define $\theta$ as the ratio of the number of times that an object accessed is equivalent to another object accessed previously, to the total accesses of this set of objects.

$$\theta = \frac{\text{num equivalent access times}(\text{ObjectO})}{\text{num equivalent + num different access times}(\text{ObjectO})} \quad (4.1)$$

Assume $O_1$ and $O_2$ are a pair of object under object replica detection. If all memory locations sampled from $O_2$ have the same values as the corresponding locations in $O_1$ ($O_2^\text{offset} = O_1^\text{offset}$), it is possible that $O_2$ and $O_1$ are two objects that have the same contents ($O_2 \equiv O_1$) or the different contents ($O_2 \not\equiv O_1$). Here, we have three scenarios and each with a specific probability:

- $O_2^\text{offset} = O_1^\text{offset}$ and $O_2 \equiv O_1$, the probability is $A$;
- $O_2^\text{offset} = O_1^\text{offset}$ and $O_2 \not\equiv O_1$, the probability is $B$;
- $O_2^\text{offset} \neq O_1^\text{offset}$, so $O_2 \not\equiv O_1$, the probability is $C$.

Obviously, we have $A + B + C = 1$.

Then, the $\theta$ can be rewritten using $A, B, C$ as:

$$\theta = \frac{A + B}{A + B + C} = A + B \quad (4.2)$$

Furthermore, we define $\alpha$ as the probability of $O_2^\text{offset} = O_1^\text{offset}$ when $O_2 \not\equiv O_1$. Then $\alpha$ can be denoted as:

$$\alpha = \frac{B}{B + C} \geq \frac{B}{A + B + C} = B \quad (4.3)$$
Combine Equation (4.2) and Inequality (4.3), we have:

\[ A = \theta - B \geq \theta - \alpha \]  

(4.4)

Assume there are \( X \) objects \( \{O_1, O_2, ..., O_x\} \) belonging to the same calling context. These \( X \) objects are divided into \( N \) groups. Inside each group, the objects are identical with each other. Every group contains \( X_n \) objects \((1 \leq n \leq N, \sum_{n=1}^{N} X_n = X)\), and \( X_1, X_2, ..., X_N \) are sorted in an ascending order by group size. Based on these \( X \) objects, there are \( {X \choose 2} \) object pairs. Among these \( {X \choose 2} \) object pairs, there will be \( \sum_{n=1}^{N} \left( \frac{X_n}{2} \right) \) identical object pairs. Considering we can estimate identical object pairs ratio by \( \frac{A}{A+B+C} = A \) and Inequality (4.4), we can state:

\[ \sum_{n=1}^{N} \left( \frac{X_n}{2} \right) \geq A \geq \theta - \alpha \]  

(4.5)

Then, \( \sum_{n=1}^{N} \left( \frac{X_n}{2} \right) \) can be derived as:

\[ \sum_{n=1}^{N} \left( \frac{X_n}{2} \right) = \frac{\sum_{n=1}^{N} X_n(X_n - 1)}{X(X - 1)} < \frac{\sum_{n=1}^{N} X_n^2}{X^2} \]  

(4.6)

Focusing on \( \sum_{n=1}^{N} \left( \frac{X_n}{X} \right)^2 \), we can reconstruct it as:

\[ \sum_{n=1}^{N} \left( \frac{X_n}{X} \right)^2 = \sum_{n=1}^{N-1} \left( \frac{X_n}{X} \right)^2 + \frac{X_N}{X} * \frac{X_N}{X} \]

\[ = \sum_{n=1}^{N-1} \left( \frac{X_n}{X} \right)^2 + (1 - \sum_{n=1}^{N-1} \frac{X_n}{X}) * \frac{X_N}{X} \]  

(4.7)

Since \( X_N > X_{N-1} > X_{N-2} > ... > X_1 \), we have

\[ \sum_{n=1}^{N-1} \frac{X_n}{X} \left( \frac{X_N - X_n}{X} \right) > 0 \]  

(4.8)

Furthermore, combining (4.5), (4.6), (4.7), (4.8), we then obtain

\[ \frac{X_N}{X} \geq \theta - \alpha + \sum_{n=1}^{N-1} \frac{X_n}{X} \left( \frac{X_N - X_n}{X} \right) > \theta - \alpha, \]

Because \( \frac{X_N}{X} \) represents the largest identical objects group size ratio, we know that this ratio is lower bounded by \( \theta - \alpha \). Next we show the upper bound of \( \frac{X_N}{X} \).
Based on equation 4.5, we have:

\[ \sum_{n=1}^{N} \binom{X_n}{2} = A > \frac{\binom{X_N}{2}}{\binom{X}{2}} \] (4.9)

Focusing on \[ \frac{\binom{X_N}{2}}{\binom{X}{2}} \], we have:

\[ \frac{\binom{X_N}{2}}{\binom{X}{2}} = \frac{X_N(X_N - 1)}{X(X - 1)} = \frac{X_N}{X} \left( \frac{X_N}{X} + \frac{X_N - 1}{X - 1} - \frac{X_N}{X} \right) = \frac{X_N}{X} \left( \frac{X_N}{X} - \frac{X - X_N}{X(X - 1)} \right) \] (4.10)

We also have:

\[ \frac{X - X_N}{X(X - 1)} = \frac{1}{X - 1} - \frac{1}{X - 1} * \frac{X_N}{X} \] (4.11)

We then denote \[ \frac{X_N}{X} = s \] and \[ \frac{1}{X - 1} = t \], equation 4.10 can be rewritten as:

\[ \frac{\binom{X_N}{2}}{\binom{X}{2}} = s(s - t + st) \] (4.12)

Combining with equation 4.9, we have this in-equation:

\[ A > s(s - t + st) = (t + 1)s^2 - st > s^2 - st \] (4.13)

Solving this in-equation, we then have:

\[ s < \frac{t + \sqrt{t^2 + 4A}}{2} \] (4.14)

Focusing on \( A \) in equation 4.2 and 4.3, we have:

\[ A = \theta - B = \theta - \alpha * (B + C) = \theta - \alpha * (1 - A) \] (4.15)

Solving it, we then have:

\[ A = \frac{\theta - \alpha}{1 - \alpha} \] (4.16)
Based on the in-equation 4.14, we have:

\[ s < \frac{t + \sqrt{t^2 + 4\frac{\theta - \alpha}{1 - \alpha}}}{2} = \frac{t}{2} + \sqrt{\frac{t^2}{4} + \frac{\theta - \alpha}{1 - \alpha}} \]

\[ = \frac{1}{2(X - 1)} + \sqrt{\frac{1}{4(X - 1)^2} + \frac{\theta - \alpha}{1 - \alpha}} \]  

(4.17)

Here, we have both lower bound and upper bound of \( \frac{X_N}{X} \): \( \theta - \alpha < \frac{X_N}{X} < \frac{1}{2(X - 1)} + \sqrt{\frac{1}{4(X - 1)^2} + \frac{\theta - \alpha}{1 - \alpha}} \)

For real applications, usually we have \( X >> 1 \), so \( \frac{1}{X-1} \to 0 \). And also we have \( \frac{\theta - \alpha}{1 - \alpha} < \frac{\theta}{1} = \theta \).

**Definition. Lower Bound Factor (LBF) \( \omega \):** \( \omega \) is defined as the lower bound of the largest identical objects group size ratio \( \frac{X_N}{X} \), so we have \( \omega = \theta - \alpha \).

**Definition. Upper Bound Factor (UBF) \( \gamma \):** \( \gamma \) is defined as the upper bound of the largest identical objects group size ratio \( \frac{X_N}{X} \), so we have \( \gamma = \frac{1}{2(X - 1)} + \sqrt{\frac{1}{4(X - 1)^2} + \frac{\theta - \alpha}{1 - \alpha}} \).

We show how the interval of the largest identical objects group size ratio \( \frac{X_N}{X} \) guides our optimizations in Section 4.6.

In the applications we evaluated, we have not seen an application with a very high \( \alpha \) (we compute \( \alpha \) for each application via exhaustively checking every field of objects), which we further discuss here. For an application with inequivalent objects \( X_1, X_2, ..., X_N \) that belong to the same calling context, high \( \alpha \) indicates that most of the contents of these objects are the same and only a few contents are different, which means these objects are partially replicated. It is worth noting that partially replicated objects can warrant some optimization to move redundant computations, such as approximate computing or data compression; however, it is out of the scope of this work.

The theoretical analysis influences the design decisions in two aspects: on the one hand, the theoretical bounds guarantee the analysis accuracy of OJXPerf’s sampling technique; on the other hand, the bounds, as metrics, help users determine whether the object replicas are significant for optimization.

### 4.6 Evaluation

We evaluate OJXPerf on a 36-core Intel Xeon E5-2699 v3 (Haswell) CPU clocked at 2.3GHz running Linux 4.8.0. The memory hierarchy consists of a private 32KB L1...
Table 4.2: Overview of performance optimization guided by OJXPerf.

<table>
<thead>
<tr>
<th>Real-world Applications</th>
<th>Inefficiency</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Problematic Code</td>
<td>PhaseOptions.java (84)</td>
</tr>
<tr>
<td></td>
<td>⍬ (RF)</td>
<td>83.9%</td>
</tr>
<tr>
<td></td>
<td>⍺ (LBF)</td>
<td>20.8%</td>
</tr>
<tr>
<td></td>
<td>ω</td>
<td>63.1%</td>
</tr>
<tr>
<td></td>
<td>ω (LRB)</td>
<td>1.17±0.02</td>
</tr>
<tr>
<td></td>
<td>WS (×)</td>
<td>7.5%</td>
</tr>
<tr>
<td></td>
<td>WH (%)</td>
<td>11.2%</td>
</tr>
<tr>
<td></td>
<td>WCM (%)</td>
<td>10.3%</td>
</tr>
<tr>
<td></td>
<td>WEI (%)</td>
<td>10.3%</td>
</tr>
</tbody>
</table>


Applications and Benchmarks. The lightweight nature of OJXPerf allows us to collect profiles from a variety of Java and Scala applications obtained from the Awesome Java repository [Kull 2020], such as the Renaissance benchmark suite [Prokopec et al. 2019], Soot [Sable-Research 2019], parquet MR [Twitter 2019], Findbugs [Pugh et al. 2015], Eclipse Deeplearning4J [Skymind 2019], JGFSerialBench [Bull et al. 2001], RoaringBitmap [Lemire et al. 2019], Apache SAMOA [Apache 2017], to name a few. We run these applications with different real inputs released with them or the real inputs that we can find to our best knowledge; the inputs control the parallelism configuration.

Replication. Figure 4.1 shows the replication ratios for more than 50 Java programs obtained from OJXPerf. We can see that several Java programs suffer from significant object replications (replication ratio > 15%). We optimize some of them, as shown in Section 4.7 under the guidance of OJXPerf. For some Java applications (e.g., gauss-mix, log-regression, page-rank, scala-kmeans) with high replication ratios, we only obtain trivial speedups because these applications do not have any hotspot object replicas. For example, the top five object replicas’ accessing times in Renaissance benchmark gauss-mix are less than three. In this case, it is reasonable that there is no benefit to optimize these replicated objects that are used very few. We focus on the hotspot object replicas, which at least are accessed dozens of times.

OJXPerf is able to pinpoint many object replicas that are not reported by existing profilers and guide optimization choices. Table 4.2 summarizes the new findings identified by OJXPerf, which we further elaborate in Section 4.7. In Table 4.2, we report...
replication factor $\theta$, $\alpha$, and lower bound factor $\omega$, which are defined in Section 4.5. Table 4.2 shows that $\alpha$ ranges from 0% to 53% and $\theta$ ranges from 65% to 100%, respectively. As a result, the lower bound factor $\omega$ is usually $> 15\%$, which means that these Java applications at least have 15% objects suffering from object replication.

**Optimization.** It is worth doing the optimization to decrease the creations of objects with the same contents. To guarantee optimization correctness, we ensure the optimized codes do not change semantics for any inputs and pass the validation tests. To avoid system noises, we run each application 30 times and use a 95% confidence interval for the geometric mean speedup to report the performance improvement, according to a prior approach [Su et al. 2019a].

From Table 4.2, we can see that we are able to obtain nontrivial speedups by removing object replicas. The performance improvement comes from the reduction of heap memory usage, cache misses, and executed instructions, which are measured with jmap [Oracle 2011] and perf [Linux 2015]. We detected the object replicas as shown in Table 4.2 without much effort. Object replicas often concentrate around only a few calling contexts making investigation relatively simpler; for example, in all of our case studies, we found the top five objects (sorted by replication factor) account for $\sim 37\%$ of whole-program object replicas on average.

### 4.6.1 False Positives and Negatives

As a sampling-based tool, OJXPerf can introduce false negatives — missing some object replicas. However, the statistics theory guarantees the high probability of capturing object replicas that occur frequently. The false negatives do not hurt the insights obtained from OJXPerf because optimizing infrequently occurred object replicas typically receives trivial speedups.

OJXPerf can also introduce false positives — reporting object replicas that are not replicated because OJXPerf uses sampling instead of exhaustive checking. The false positives occur when most elements of the two objects are the same, with only a few different elements not monitored. However, OJXPerf randomly checks different fields with enough samples to minimize the false positives. We exhaustively check every field of the top five objects (i.e., objects associated with most samples) of all our investigated programs in Table 4.3. From the table, we can see that OJXPerf incurs 5.9% false positives.
Table 1

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time Overhead</th>
<th>Time Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>akka-uct</td>
<td>1.44</td>
<td>0.12</td>
</tr>
<tr>
<td>als</td>
<td>1.12</td>
<td>0.05</td>
</tr>
<tr>
<td>chi-square</td>
<td>1.09</td>
<td>0.02</td>
</tr>
<tr>
<td>db-shootout</td>
<td>1.43</td>
<td>0.05</td>
</tr>
<tr>
<td>SPECjvm2008</td>
<td>9.12</td>
<td></td>
</tr>
<tr>
<td>Renaissance Benchmark Suite</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4: OJXPerf’s runtime and memory overheads in the unit of times (×) on various benchmarks.

### 4.6.2 Overhead Measurement

The runtime overhead (memory overhead) is the ratio of the runtime (peak memory usage) of the execution monitored by OJXPerf to the runtime (peak memory usage) of the native execution. To quantify the overhead, we apply OJXPerf to three well-known Java benchmark suites: Renaissance [Prokopec et al. 2019], Dacapo 9.12 [Blackburn et al. 2018], and SPECjvm2008 [SPEC 2008]. We run all benchmarks with four threads. We run every benchmark 30 times and compute the average and error bar. Figure 4.4 shows the overhead when OJXPerf is enabled at a sampling period of 5M. Some Renaissance and Dacapo benchmarks have higher time overhead (larger than 30%) because they allocate too many objects (e.g., more than 400 million allocations for mnemonics, par-mnemonics,
Table 4.3: Accuracy for OJXPerf’s replica detection.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Replicated</th>
<th>Not Replicated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replicated</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Not Replicated</td>
<td>3</td>
<td>48</td>
</tr>
</tbody>
</table>

Correctness: \((48 + 8) / (0 + 8 + 3 + 48) = 94.9\%\)

False positive rate: \(3 / (3 + 48) = 5.9\%\)

scrabble, akka-uct, db-shootout, dec-tree, neo4j-analytics).

### 4.7 Case Studies

This section shows how OJXPerf pinpoints object replicas in real applications and guides the optimization. Our optimization guarantees the program’s correctness via human inspection, and we have evaluated our transformed code with tests to ensure their correctness. It is worth noting that existing profilers may identify the same object allocation as a hotspot in memory usage; however, they do not know whether this allocation point creates multiple object replicas for potential optimization. OJXPerf, in contrast, quantifies the replication factors for the objects to provide intuitive optimization. We have submitted our optimization patches in several cases and gotten them confirmed or upstreamed, e.g., Soot and Findbugs.

#### 4.7.1 Soot

Soot is a Java optimization framework, which uses containers extensively [Sable-Research 2019]. We run Soot-3.3.0 using the bytecode of the DaCapo benchmark avrora as input. Figure 4.5 shows the snapshot of OJXPerf’s Flame Graphs GUI in VSCode for intuitive analysis. The top pane of the GUI shows the Java source code; the bottom shows the flame graphs of object accesses in their full call stacks. In the flame graphs, the x-axis shows the accesses with their call stacks to object replicas, and the y-axis shows call stack depth, counting from zero at the top. Each rectangle represents a stack frame. The wider a stack frame is, the higher of replication factor of this stack frame. The GUI in Figure 4.5 shows one problematic object \(st\) (highlighted in blue), which is accessed on line 84 in method `getPhaseOptions` of class `PhaseOptions` with many replicas (its replication factor \(\theta\) is 83.9\%, as shown in the top pane of the GUI).
Soot’s execution is divided into a number of phases, such as Jimple Body Creation (jb) phase, Java To Jimple Body Creation (jj) phase, Grimp Body Creation (gb) phase, etc. In the jb phase, the JimpleBodys are built by a phase called jb, which is itself comprised of subphases, such as the aggregation of local variables (jb.a), type assigner (jb.tr), dead assignment eliminator (jb.dae), etc. Each of these subphases that belong to jb phase has its own default option. By investigating the source code, we found that when the soot executes these subphases sequentially, the default option for different subphases is stored in the reported StringTokenizer object, which keeps unchanged.

To eliminate these replicas, we only read the default option when its contents are changed; otherwise, we reuse the default option from the prior subphase. This optimization yields a \((1.17 \pm 0.02) \times\) speedup to the entire program.

4.7.2 Eclipse Deeplearning4J – SameDiff

Eclipse Deeplearning4J integrates with Hadoop and runs on several backends [Skymind 2019]. We run Deeplearning4J using SameDiff, a TensorFlow/PyTorch-like framework for executing complex graphs. This framework is also the lower lever base API for running onnx and TensorFlow graphs. OJXPERF investigates
Listing 4.2: OJXPERF identified the `shapeInfo` array with replicas in Deeplearning4J SameDiff.

the training phase and reports that the replication factor $\theta$ of the input array `shapeInfo` is 64.7%, indicating high redundancies in the computation on this array in method `hasBitSet`, which determines array types, as shown in Listing 4.2.

Deeplearning4J SameDiff builds a directed acyclic graph, whose nodes are differential functions used to compute gradients. In the SameDiffLayer (a base layer used for implementing Deeplearning4J layers with SameDiff), Deeplearning4J provides a set of operations named ”Custom operations” designed for the SameDiff graph. To execute the ”Custom operations” within graph, these operations are stored in a two-dimensional array. Then Deeplearning4J splits this two-dimensional array into different small partitions. Each partition has its own shape identifier array `shapeInfo`, which is used to determine four shape properties: SPARSE, COMPRESSED, EMPTY, and DENSE. We found that the adjacent partitions in the SameDiff graph often have the same shape property due to the good locality among adjacent partitions.

To eliminate redundancies, we first check whether the `shapeInfo` in the current iteration has the same value as in the last iteration. If the `shapeInfo` is unchanged, we reuse the shape property memoized from the previous iteration, which saves the call to `hasBitSet`. This yields a $(1.09 \pm 0.02) \times$ speedup to the entire program.

### 4.7.3 Eclipse Deeplearning4J — AlphaGo Zero

We also run Deeplearning4J using AlphaGo Zero model [Bridgland et al. 2020], which combines a neural network and Monte Carlo Tree Search in an elegant policy iteration framework to achieve stable reinforcement learning. OJXPERF studies the training stage and reports an object, `Map<String, NDArrayCompressor>` `codecs`, which is allocated on line 53 and accessed on line 57 in method `loadCompressors` of class `BasicNDArrayCompressor` with many replicas (its replication factor $\theta$ is 81.3%), as shown
public void init(INDArray parameters) {
...
for (int vertexIdx : topologicalOrder) {
    paramsViewForVertex[vertexIdx] = parameters.get(NDArrayIndex.interval(0,0,true));
    // get method calls loadCompressors to decompress data
}

protected Map<String, NDArrayCompressor> codecs = new ConcurrentHashMap<>();
protected void loadCompressors() {
...
for (NDArrayCompressor compressor : compressors) {
    codecs.put(compressor.getDescriptor(), compressor);
}
}

Listing 4.3: OJXPERF identified the codecs map with replicas in Deeplearning4J AlphaGo Zero.

in Listing 4.3.

The reason for generated replicas is due to constructing the computational graph in the AlphaGo Zero model. To initialize the computational graph, the AlphaGo Zero model uses an existing array parameters for each layer. Given the topological order, the AlphaGo Zero model constructs the computational graph by iterating each subset of array parameters. Since the array parameters is in compressed status, to obtain the elements of array parameters, the program needs to decompress it first. Deeplearning4J framework provides several different compression algorithms (compressor) based on the data type (e.g., FLOAT16, FLOAT8, INT16, etc). Since every subset of array parameters has the same data type, which means we don’t need to load the new compressor and store it again in map codecs in a loop (line 69 of Listing 4.3). To avoid the redundant loading and storing compressor, we check whether the program is processing different subsets in the same array parameters. If so, we use the current compressor directly. This optimization yields a (1.15 ± 0.01) speedup to the entire program.

4.7.4 FindBugs-3.0.1

FindBugs looks for code instances that are likely to be errors [Pugh et al. 2015]. We run Find-Bugs on a real input Java chart library 1.0.19 (a widely used client-side chart library for Java). OJXPERF reports an object that has many replicas, BasicBlock block (an object with a user-defined type), which is accessed on the line 183 in method lookupOrCreateFact of class BasicAbstractDataflowAnalysis, as shown in Listing 4.4.

The replicas come from the algorithm of data-flow analysis used in FindBugs.
public /* final */ Fact getStartFact (BasicBlock block) {
    return lookupOrCreateFact (startFactMap, block);
}

public /* final */ Fact getResultFact (BasicBlock block) {
    return lookupOrCreateFact (resultFactMap, block);
}

private Fact lookupOrCreateFact (Map<BasicBlock, Fact> map, BasicBlock block) {
    Fact fact = map.get (block);
    if (fact == null) {
        fact = createFact ();
        map.put (block, fact);
    }
    return fact;
}

Listing 4.4: The source code highlighted by OJXPERF shows the object block with replicas in Findbugs.

bugs divides a data-flow graph into tiny-sized blocks and creates an object for each block instead of creating a single object for the whole graph. Consequently, most created objects have the same content due to good value locality among adjacent blocks. OJXPERF finds that the method lookupOrCreateFact (line 182 of Listing 4.4) method is usually invoked with the same input BasicBlock block. OJXPERF reports that the replication factor $\theta$ of the input block is 70.3%, indicating many replicas of this object. To avoid the redundant lookup and creation, we check whether a different block is produced in the current iteration. If the block is unchanged, we return fact obtained from the last invocation directly. This optimization yields a (1.25 ± 0.03)× speedup to the entire program.

### 4.7.5 fj-kmeans

fj-kmeans is a benchmark from Renaissance Suite, used to run the k-means algorithm [Prokopec et al. 2019]. We run fj-kmeans using the fork/join framework as input. OJXPERF reports an object that has many replicas, array result, which is allocated on line 5 in method findNearestCentroid and accessed on line 2 in method compute Directly of class JavaKMeans, as shown in Listing 4.5.

The generated replicas are due to the finding nearest centroid algorithm for many different sets of elements. This finding nearest centroid algorithm maintains a collection of centroids, and the distance between each centroid is significant. Then, during some computation periods, because different sets of elements have small distance, the program keeps generating the same centroids and put the centroids’ indices into an array result(line 12 of Listing 4.5), which is the object with many replicas (the replication factor $\theta$
protected Map<Double[], List<Double[]>> computeDirectly() {
    return collectClusters(findNearestCentroid());
}

private int[] findNearestCentroid() {
    final int[] result = new int[taskSize];
    for (...) {
        final Double[] element = data.get(dataIndex);
        for (...) {
            final double distance = distance(element, centroids.get(centroidIndex));
            result[dataIndex - fromInclusive] = centroidIndex;
        }
    }
    return result;
}

private Map<Double[], List<Double[]>> collectClusters(final int[] centroidIndices) {
    // computation with input parameter centroidIndices
}

Listing 4.5: OJXPerf pinpoints the result object with many replicas in fj-kmeans.

is 76.1%) reported by OJXPerf.

To eliminate efficiencies, we check the values in array result produced by findNearestCentroid. If result is unchanged, we reuse the return value of method collectClusters memoized from the last iteration, which avoids the redundant computation. This optimization yields a (1.08 ± 0.04)× speedup to the entire program.

4.7.6 JGFSerialBench

JGFSerialBench is a benchmark from the Java Grande benchmark suite, which is designed to represent scientific and other numerically intensive computation [Bull et al. 2001]. We run JGFSerialBench using its default input. OJXPerf pinpoints arraybase, an object allocated on line 371 in method JGRun of class JGFSerialBench with many redundancies, as shown in Listing 4.6. Its redundancy factor θ is as high as 68.8%. Listing 4.6 shows that arraybase is a two-dimensional array allocated and used in a while loop. To eliminate these replicas, we check whether the value of arraybase is changed across iterations; we bypass the redundant computation by reusing results from the prior iteration. This optimization yields a (1.1 ± 0.03)× speedup to the entire program.

4.7.7 RoaringBitmap

Roaringbitmap is compressed bitmaps that tend to outperform conventional compressed bitmaps such as WAH, EWAH, or Concise [Lemire et al. 2019]. We run RoaringBitmap with SerializationBenchmark as its input. OJXPerf reports array keys accessed on
public void JGFrun(){
...  
while (time < TARGETTIME && size < MAXSIZE){
  ▶arraybase = new arrayitem [size][LENGTH];
  arrayout = new ObjectOutputStream (arrayfout);
  ... // put contents in array base
  for (j=0; j<size; j++) {
    ▶arrayout.writeObject( arraybase[j]);
  }
  }
  }
}

Listing 4.6: OJXPERF pinpoints the arraybase with replicas in JGFSerialBench.

Listing 4.7: The source code highlighted by OJXPERF shows the keys array has replicas in RoaringBitmap.

line 290 in method deserialize of class MutableRoaringArray has many replicas; its replication factor $\theta$ is as high as 100%. Listing 4.7 shows that method deserialize declares an array keys (line 286), fills its slots (line 288), and assigns to a class member array this.keys. OJXPERF reports that array keys is immutable, meaning that that the method deserialize generates the same array keys across different invocations.

To optimize the code, we avoid redundant array updating inside the loop. Instead, we only update array this.keys if the deserialize method generates a different array keys. The performance of RoaringBitmap is measured as latency, i.e., microseconds per operation. Our optimization reduces the latency by $(1.09 \pm 0.01) \times$.

4.8 Summary

In this work, we design and develop OJXPERF, the first lightweight profiler to identify object replicas in Java applications. As a unique feature, OJXPERF combines the use of performance monitoring units, debug registers and lightweight byte code instrumentation for statistical object replica detection. With the evaluation of more than 50 Java applications, we show OJXPERF minimizes false positives and incurs 9% and 6% runtime
and memory overheads, respectively. We further optimize several real-world applications guided by OJXPERF that result in a noticeable reduction in heap-memory demands and significant runtime speedups. Many optimization patches are confirmed or upstreamed by the software developers.
Chapter 5

DroidPerf: Profiling Memory Objects on Android Devices

5.1 Introduction

The free and open-source Android platform has established itself as the dominant mobile operating system on the market. Android-based mobile phones and tablets are the most popular electronic devices. Java and Kotlin are the de-facto programming languages for Android apps, which are run in Android Runtime (ART) virtual machine. ART employs garbage collectors (GC) to manage memory objects allocated on heap, which effectively avoid many memory errors, such as memory leaks.

However, ART is not a panacea for memory inefficiencies, which are rooted at the growing speed gap between CPU and memory in Android ecosystems. Memory inefficiencies not only slow down the app execution, but also drain the battery quickly. One the one hand, apps with poor data locality can result in excessive cache misses, increasing the average memory access latency. On the other hand, ART can introduce memory bloats [Xu 2012] due to the delay of GC for reclaiming unused data objects, which increase the memory footprint. Thus, it is urgent for Android apps to maximize memory efficiencies by enhancing data locality and avoiding memory bloats.

Memory inefficiencies have been extensively explored in native languages in prior studies. Many tools [Berg et al. 2005; Beyls et al. 2006, 2009; Liu et al. 2011, 2013a,b,c, 2014b, 2015; Zhong et al. 2009] that focus on measuring data locality with various metrics, which fall into two categories: software metrics and hardware metrics. A software metric measures data locality independently of architecture, while hardware metric measures
data locality for a specific architecture. Software metrics usually are derived from memory traces, such as cache miss ratio curves [Beyls et al. 2009; Cašcaval et al. 2003], reuse distances [Ding et al. 2003; Rane et al. 2012; Shen et al. 2007; Zhong et al. 2008, 2009], and footprints [Bastoul et al. 2003; Kumar et al. 1998]. As the program runs, hardware metrics, such as cache hits or misses, are collected by hardware performance monitoring units (PMUs). With these metrics, data locality can either be characterized or optimized for programs.

However, these approaches on native languages cannot be directly applied to Android apps because of the unique challenges in the ART abstractions. First, ahead-of-time (AOT) compilation and interpretation on ART separate program source code from its execution behaviors. Second, the ART managed memory system, including virtual machines and GC, further prevents from understanding memory inefficiencies in Android apps. Third, unlike many Java profilers that have the Java Virtual Machine Tool Interface (JVMTI) [Oracle 2007] supported, JVMTI on ART is only partially supported. Hence, some essential functions for profiling tools are not available. Fourth, ART does not have any ASYNC unwind facility (the AsyncGetCallTrace API provided by OpenJDK/oracle JVMs to facilitate non-safepoint collection of stack traces), which means profiler developers are not able to get the call stack inside a signal handler.

By working around these challenges, existing tools on Android, such as Simpleperf [sim 2022], Android Profiler [and 2021], Perfetto [per 2021], and many others [Falaki et al. 2011; Flinn et al. 1999; Huang et al. 2010; Isci et al. 2003; Joseph et al. 2001; Shepard et al. 2010; Xu et al. 2011b; Zhang et al. 2010] pinpoint hotspots in CPU or memory as well as the energy consumption. While these tools can quantify the amount of memory usage, they do not associate inefficiencies with data objects or provide necessary performance statistics, resulting in limited actionable insights for Android app optimization. Section 5.1.1 elaborates on this issue by showing a motivating example.

To address the limitations in existing tools, we develop DROIDPERF, which monitors object lifespan and JVM heap memory consumption and associates these performance statistics with data objects. DROIDPERF distinguishes itself from existing approaches by its ability to detect memory inefficiencies with the combination of minimal byte code instrumentation and statistical sampling on the object level, which makes it applicable to the production environment. Based on an extensive analysis of several popular An-

---

For example, the JVMTI compiled method callbacks (sent when a method is compiled and loaded/unloaded into memory by the VM) CompiledMethodLoad and CompiledMethodUnload cannot be used in ART.
Listing 5.1: The code snippet from SciMark 2.0 Scimark.fft. Optimizing memory bloat by moving the object allocation site at line 3 outside of the loop yields a nontrivial speedup \((1.12 \pm 0.02) \times\) to the entire program.

droid apps, we have discovered that pinpointing memory inefficiencies can lead to new optimization opportunities. In the rest of this section, we show two motivating examples, paper contributions, and organization.

### 5.1.1 Motivating Examples

The purpose of this section is to demonstrate the importance of optimizing memory inefficiencies in Android apps. We run SciMark 2.0 [SPEC 2004], which is a popular benchmark suite to evaluate Android performance [Wang et al. 2020; Zhao et al. 2020], on a Google Pixel 5. Listings 5.1 and 5.2 present two snippets of code that have memory inefficiencies, which are respectively from Scimark.fft and Scimark.lu.

Listing 5.1 shows an array of `data` objects allocated in the method `makeRandom` at line 10. Memory bloat occurs due to this allocation site being invoked 973 times. A loop enclosing the method `makeRandom` can be optimized by moving array allocation outside of the loop and replacing it with a static object array instead, often referred to as the singleton pattern. By addressing the memory bloat, this optimization yields a \((1.12 \pm 0.02) \times\) speedup. **Principle I: pinpointing problematic objects in Android apps can provide more actionable insights for optimization.**

Listing 5.2 shows a memory bloat occurring at line 9 due to the object `T`, which is allocated 1725 times. Similarly, the singleton pattern can be applied by declaring the allocation site of object `T` as a static object and hoisting it outside the loop enclosing the method. Although this optimization addresses memory bloat, a noticeable speedup is not apparent. This is because the data object `T` does not incur high access latency.
1 public static int factor(double[][] A, int[] pivot) {
2   ...
3   for (int j = 0; j < minMN; j++) {
4       x = new_copy(A);
5       ...
6   }
7 }
8 protected static double[][] new_copy(double[][] A) {
9     double[][] T = new double[M][N];
10    for (int i = 0; i < M; i++) {
11        double[] Ti = T[i];
12        ...
13    }
14   ...
15 }

Listing 5.2: The code snippet from SciMark 2.0 Scimark.lu. Optimizing memory bloat by moving the object allocation site at line 3 outside of the loop does not bring any speedup to the entire program.

**Principle II:** only memory usage and allocation times are inadequate to quantify memory inefficiencies; hardware performance statistics (e.g., cache misses) are necessary.

In this paper, we present DROIDPERF, an object-centric profiling following the two aforementioned principles. In concrete terms, DROIDPERF measures cache misses with PMU on individual memory accesses and reports all live objects at this point. For each monitored object, DROIDPERF collects various information including call path, size, and allocation frequency. As an example, DROIDPERF reports that the code snippet of method makeRandom shown in Listing 5.1 accounts for 42.8% of total cache misses. In contrast, new copy method shown in Listing 5.2 only accounts for 1.6% of total cache misses. As a result, the different speedups obtained from the locality optimization can be explained. Hence, an object-centric analysis that includes the cache miss metric would be necessary to identify whether locality optimization could result in significant performance gains.

### 5.1.2 Contribution Summary

Our work proposes DROIDPERF, an object-centric profiler for improving the data locality on ART. DROIDPERF’s contributions are as follows.

- DROIDPERF is able to provide the active JVM heap usage and L1 cache misses ratio for each memory access call path, along with the live object information (allocation site frequency and location). Android applications can use this rich information to guide optimization and obtain significant performance gains.
• **DroidPerf** works with unmodified Android applications, the Dalvik virtual machine and operating system, running on commodity Android CPU processors.

• In a comprehensive evaluation, we demonstrate that **DroidPerf** can offer intuitive guidance to Android’s unique memory inefficiencies, with 14% runtime overhead and 8% memory overhead. With **DroidPerf**, we are able to improve data locality in various Android applications. Some of the optimization patches have been upstreamed to the real-world Android application repositories.

### 5.2 Categorizing Memory Inefficiencies

We studied many Android benchmarks (EEMBC benchmarks [eem 2022], 0xBenchmark suite [0xb 2022], SciMark 2.0 benchmarks [SPEC 2004] and CaffeineMark 3.0 benchmarks [caf 2022]) and more than 100 popular Android apps. Table 5.1 measures many Android app behaviors from four domains: memory intensive, computation intensive, object-allocated intensive, and high cache miss. Based on Table 5.1, we found that the Android apps suffering from high cache miss with memory and object-allocated intensive are likely to have object locality issues. From Section 5.2.1 to Section 5.2.6, we identify six types of root causes of memory inefficiencies and show them with real-world Android apps as examples.

#### 5.2.1 Inefficiency due to Memory Bloat

Memory bloat occurs by allocating (and initializing) many objects that are not reclaimed immediately after their last usage. For instance, allocating objects in a loop where the object’s lifetime is only the scope of the loop body. On Android, the memory bloat is usually due to inappropriate memory management or suboptimal implementation of algorithms.

**Rajawali.** Rajawali is a 3D engine for Android based on OpenGL ES 2.0/3.0 [raj 2022]. **DroidPerf** pinpoints two problematic objects, **cube** and **mEffects**, which are allocated on line 64 and 66 in method **initScene** of class **RenderToTextureFragment**, as shown in Listing 5.3. **DroidPerf** reports that method **initScene** accounts for 15.3% of total cache misses in the entire program.

With further investigation, the memory bloat comes from the algorithm of rendering texture space used in Rajawali. Rajawali divides an entire scene into many tiny-sized
Table 5.1: Measuring behaviors of popular Android apps.

<table>
<thead>
<tr>
<th>App Name</th>
<th>Memory Intensive</th>
<th>Computation Intensive</th>
<th>Object-allocated Intensive</th>
<th>High Cache Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twire</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Foxy Droid</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fresco</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>DNS66</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Rajawali</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>OkHttp</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>Retrofit</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Stetho</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ExoPlayer</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>IjkPlayer</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
</tr>
<tr>
<td>Robospice</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>LeakCanary</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NetGuard</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>Mullvad</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>GmsCore</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Threema</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>CoolReader</td>
<td>✓</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
</tr>
<tr>
<td>RxAndroid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bromite</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>Applozic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FrostWire</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Glide</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>MediaPicker</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Snikket</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>Tusky</td>
<td>✓</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
</tr>
<tr>
<td>LibreTorrent</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>AdAway</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
<td>✕</td>
</tr>
<tr>
<td>WireGuard</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
</tr>
<tr>
<td>ProtonVPN</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✕</td>
</tr>
<tr>
<td>SteamChat</td>
<td>✓</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
fragments, and creates objects `cube` and `mEffects` for every fragment instead of declaring them as global objects for the entire scene. Consequently, the two problematic objects `cube` and `mEffects` are both repeatedly allocated in loops with no overlap in lifecycles across different instances. We hoist the two object allocations out of the loops to avoid memory bloat. This optimization reduces total cache misses by 10.1%, as well as JVM heap memory consumption by 17.4%.

### 5.2.2 Inefficiency due to Long Running Services

On Android, one of the biggest memory-management mistakes is leaving services running when they are not needed. When developers start a service, the system prefers to always keep the process for that service running. As a result, service processes become very expensive since the RAM used by the service remains unavailable to other processes. This reduces the number of cached processes that the system can keep in the LRU cache, making app switching less efficient.

**Applozic.** Applozic brings real-time engagement with chat, video, and voice to Android web, mobile, and conversational apps [app 2022]. DROIDPERF pinpoints many object allocations, which belong to a running services, `MessageClientService`, as shown in Listing 5.4. According to DROIDPERF’s analysis, accessing the service `MessageClientService` accounts for 47.3% of total cache misses.

We find that the service `MessageClientService` in Applozic always keeps running and allocating objects excessively. By further investigating the source code, we find that the service `MessageClientService` is used to synchronize messages to the server. However, we can call this service between a time interval instead of syncing messages all the time. In our optimization, instead of using persistent services, we use an alternative

```
public void initScene () {
    for (...) { // For each fragment
        ▶Cube cube = new Cube();
        ▶mEffects = new PostProcessingManager(...);
        setTransformable3D(cube);
        mEffects.addPass(renderPass);
    }
}
```

Listing 5.3: The problematic source code in Rajawali due to memory bloat.
public class MessageClientService extends MobiComKitClientService implements
A1JobIntentService {
  ...
  public boolean syncMessagesWithServer(List<Message> messageList) {
    ▶ List<Message> messages = new ArrayList<Message>(messageList);
    do {
      ▶ SmsSyncRequest smsSyncRequest = new SmsSyncRequest();
      ▶ smsSyncRequest.setSmsList(new ArrayList<Message>(messages));
    } while (messages.size() > 0);
  } ...
}

Listing 5.4: The problematic source code in Applozic due to Long Running Services.

implementation JobScheduler, which can limit the lifespan of the service when it has
completed its task. Through this optimization, the total cache misses are reduced by
19.6%, and the JVM heap memory consumption is reduced by 18.9%.

5.2.3 Inefficiency due to Misuse of Android APIs

On Android, improper use of Android API could bring memory inefficiency to the whole
application. For example, improper use of Android broadcast APIs can slow down the
system and increase memory footprint.

GmsCore. GmsCore is a framework to allow applications designed for Google Play
Services to run on systems where Play Services is not available [gms 2022]. DROIDPERF
pinpoints an object of a class, TriggerReceiver, which is allocated on line 49 in method
register of class TriggerReceiver, as shown in Listing 5.5. DROIDPERF reports that
accessing the method register accounts for 11% of total cache misses in the entire
program.

By further analyzing the source code, the excessive allocations happen in method
registerReceiver, which is an Android broadcast API supported from Android 7.0 or
higher. When an event of interest occurs on Android, a broadcast is sent. A developer
can create receivers to subscribe to specific broadcasts. A large amount of allocations in
method registerReceiver indicates GmsCore registers a number of receivers. However,
GmsCore does not unregister these receivers, which are no longer needed. Consequently,
too many broadcast receivers bring system overhead and increase memory footprint. Since
registering a receiver needs an activity context, we use method unregisterReceiver to
unregister a receiver when its activity is finished or killed by other processes in our
optimization. By implementing this optimization, the total number of cache misses is
public class TriggerReceiver extends WakefulBroadcastReceiver {
  ...  // Omitted code for brevity
  public synchronized static void register(Context context) {
    if (SDK_INT >= N && !registered) {
      IntentFilter intentFilter = new IntentFilter("...");
      context.getApplicationContext().registerReceiver(new TriggerReceiver(), intentFilter);
      registered = true;
    }
  }

Listing 5.5: The problematic source code in GmsCore due to misuse of Android APIs.

reduced by 7%, and the JVM heap memory consumption is reduced by 10.7%.

5.2.4 Inefficiency due to Runtime Configuration Changes

Unlike traditional computers, smart devices are more portable and subject to frequent runtime configuration changes, such as screen orientation changes, keyboard attachments, language switching, and many others. Such changes can happen at runtime while users interact with the devices.

Twire. Twire is an open-source, ad-free Twitch browser and stream player for Android [twi 2022]. DROIDPERF spots many object allocations occur in two methods getVisualElements and urlToJSONStringHelix of class TopStreamsActivity and class Service, as shown in Listing 5.6. According to the analysis result of DROIDPERF, accessing the methods getVisualElements and urlToJSONStringHelix accounts for 39.1% of total cache misses.

Twire allocates many unnecessary objects because it uses passive restarting-based runtime change handling. When we rotate the screen for watching videos, Twire first deconstructs the current user activity, including destroying all UI components and the internal logic data, then reconstructs the activity with the alternative resources (e.g., layouts) that can match the new configuration (e.g., landscape orientation). To optimize the code, we reuse all of the visual elements in method getVisualElements if only the screen orientation is changed. This optimization reduces cache misses of the entire program by 10.4%, and the JVM heap memory consumption is reduced by 8.6%.
Listing 5.6: The problematic source code in Twire due to runtime configuration changes.

5.2.5 Inefficiency due to Unoptimized Data Structures

Some class packages provided by Java are not optimized for mobile devices. For example, one common issue is that Android developers use HashMap objects excessively. The HashMap implementation is quite a memory inefficient because it needs a separate entry object for every mapping.

Stetho. Stetho is a sophisticated debug bridge for Android apps [ste 2022]. DROIDPERF spots many allocated HashMap objects in class AsyncPrettyPrinterRegistry, ChromeDevtoolsServer, JsonRpcPeer and ResponseBodyFileManager. Access to these classes accounts for 83.5% of all cache misses in the entire program.

Unlike the unoptimized data structure HashMap, the Android framework provides several optimized data structures, including SparseArray, LongSparseArray, ArrayMap, etc. We replace HashMap objects in the aforementioned four classes with ArrayMap objects. The optimization maintains the program’s original syntax while reducing total cache misses by 24.2% and JVM heap memory consumption by 6.9%.

5.2.6 Inefficiency due to Duplicated Objects

On Android, we detect some apps keep generating duplicated objects. Two objects are duplicated if their contents are identical. We observe that generating duplicated objects is the symptom of repeatedly using the same input for a working procedure in Android apps.
private fun analyzeHeap(...) {
    ▶
    val sourceProvider = ConstantMemoryMetricsDualSourceProvider(heapDumpFile);
    ▶
    val closeableGraph = try {
        ▶
        sourceProvider.openHeapGraph(...)
    } catch (throwable: Throwable) {
        ...
    }
    ...
    }

Listing 5.7: The problematic source code in LeakCanary due to duplicated objects.

**LeakCanary.** LeakCanary is a library that helps Android developers to reduce memory crashes [lea 2022]. DROIDPERF spots an object, sourceProvider, which is allocated on line 147 and accessed on line 150 in method `analyzeHeap` of class `AndroidDebugHeapAnalyzer` with many replicas, as shown in Listing 5.7. DROIDPERF reports that accessing the method `analyzeHeap` accounts for 7% of total cache misses. *Note that DROIDPERF is able to detect the problematic object in programs written in Kotlin (a modern statically typed programming language used by more than 60% of professional Android developers).*

LeakCanary’s heap analysis introduces many duplicated objects. LeakCanary analyzes used classes in heap; several classes are the most accessed classes, such as `java.lang.Object`, `java.util.HashMap`, and others. LeakCanary stores each accessed class in an `HeapGraph` instance. Consequently, for a sequence of all classes in the heap dump, LeakCanary usually constructs and opens a separate `HeapGraph` instance for the same consecutive classes. To eliminate the redundant operations, we check whether LeakCanary is processing a different accessed class compared with the last processing. If not, LeakCanary does not need to construct and open a new `HeapGraph` instance. This optimization reduces the total cache misses by 4.3% and the JVM heap memory consumption by 11.4%.

### 5.3 Methodology

Figure 5.1 overviews DROIDPERF’s profiling methodology. DROIDPERF comes with an agent, the ART TI agent. The ART TI agent consists of two functionalities: (1) an object collection is generated by the ART TI agent by capturing object allocation information during the execution, such as the allocation context (class name, method name, and line number) and object allocation times; (2) a Java thread creation callback is subscribed
by the ART TI agent to enable PMU monitoring memory accesses. When a thread incurs a PMU sample, DROIDPERF obtains current live object information, along with the current memory usage and the L1 cache miss of the entire program. With the help of the live object’s call path, DROIDPERF is able to associate each live object seen in the sample with the individual objects in the maintained object collection, as shown in Figure 5.1. DROIDPERF requires no modification to hardware, OS, Dalvik virtual machine, ART, or monitored Android apps, which makes DROIDPERF applicable to the production environment. In this section, we discuss how the ART TI agent works and the DROIDPERF’s concept of object-centric analysis.

5.3.1 ART TI Agent

**Capturing Object Allocation via ART TI Agent.** To capture object allocation, DROIDPERF registers an ART TI callback `VMObjectAlloc`, which is invoked when an object allocation is visible to Java programming languages. Every time an object is allocated, the callback `VMObjectAlloc` returns the object information (e.g., the object pointer, type and size) to DROIDPERF.
Generating Memory Access Samples via ART TI Agent. By leveraging the ART TI agent, DROIDPERF enables and collects PMU samples. To obtain the L1 cache miss in each PMU sample, ART TI enables DROIDPERF to intercept Java thread starts, using which DROIDPERF configures PMUs to sample precise events for cache misses (e.g., `MEM_LOAD_UOPS_RETIRED:L1_MISS`). Furthermore, DROIDPERF installs a signal handler for processing PMU samples. As threads terminate, DROIDPERF stops all PMUs and generates profiles for each thread.

To capture the active memory usage on currently allocated objects in each PMU sample, DROIDPERF utilizes an ART TI callback `ObjectFree`, which is invoked when the garbage collector frees an object. The callback `ObjectFree` can return the size of the freed objects. Since we were already aware of each allocated object's size via `VMObjectAlloc` callback, DROIDPERF is able to report the JVM heap memory consumption on allocated objects within each of the PMU sampled call path. Aside from controlling PMUs, DROIDPERF also utilizes the ART TI agent for capturing the calling contexts for both PMU samples and object allocations.

5.3.2 Object-centric Attribution

Obtaining Calling Contexts. Attributing runtime statistics to a flat profile (i.e., instruction and its enclosing method only) provides insufficient details for optimization. For example, attributing inefficiencies to a common JDK method, e.g., `string.equals()`, offers little insight because `string.equals()` can be invoked from multiple places in a large code base; not all invocation instances have equal contributions to performance bottlenecks. A detailed attribution requires to associate profiles with full calling contexts: `packageA.classB.methodC:line#` → `... → java.lang.String.equals():line#`. Similarly, only knowing the location of keyword `new` offers little insight into object allocation. Thus, DROIDPERF requires obtaining calling contexts for both PMU samples and object allocations.

JVM offers users an API to obtain calling contexts: `GetStackTrace`, which is used to get information about the stack frames of a thread. `GetStackTrace` returns the method ID for each stack frame in the calling context. Method ID uniquely identifies distinct methods and distinct JITted instances of the same method (a single method may be JITted multiple times). With the method ID, DROIDPERF is able to obtain the associated class name and method name by querying JVM. To obtain the line number, DROIDPERF maintains a line number mapping table for each method instance via JVMTI
API GetLineNumberTable.

Constructing Call Path. To construct the full call path for object allocation and PMU samples, DROIDPERF utilizes JVMTI APIs MethodEntry and MethodExit, which are invoked when a method entry event or method exit event is generated. DROIDPERF instruments the start and end of every method call to construct the call path. Therefore, DROIDPERF always knows where the current execution resides in the stack frame. We can always find out the current execution path when a PMU sample callback occurs or when object allocation callbacks occur.

Identifying Objects. When Java objects are allocated on the heap, DROIDPERF will use the allocation call path to uniquely identify them. Many times, a Java application can create multiple object instances via a single allocation site in a loop. All those objects will be represented by a single call path, which naturally aggregates numerous objects with similar behavior. DROIDPERF captures each allocation instance and invokes the ART TI agent to obtain the allocation calling path. As these allocation instances share the same call path, DROIDPERF treats all these instances as a single object.

Object Attribution. On each PMU sample, DROIDPERF reports the current live object’s allocation context (class name, method name, and line number), along with the JVM heap memory consumption and L1 cache miss in the PMU sampled memory access call path. For the live object with high JVM heap memory consumption and L1 cache miss percentage of the entire program, DROIDPERF looks up this live object’s allocation context in the object collection maintained by DROIDPERF. Since a PMU sampled memory call path might involve several live objects, DROIDPERF ranks the live objects by their allocation times and finds which objects’ allocation times are dominant (> 50%) in the current group of live objects. DROIDPERF then attributes the high JVM heap memory consumption, and L1 cache misses with this excessively allocated object.

5.4 Implementation

DROIDPERF works with unmodified Android apps, the Dalvik virtual machine, and the operating system running on commodity Android CPU processors, which can be directly deployed in the production environment. Figure 5.2 overviews DROIDPERF in the system stack. DROIDPERF requires no modification to hardware, OS, Dalvik virtual machine,
ART, or Android application source code. DROIDPERF is primarily comprised of an online data collector and an offline data analyzer. The online data collector can be enabled in two ways. To profile the Android app immediately after ART starts, we can enable the online data collector as an agent by passing ART TI options. The DROIDPERF is also able to be attached to an Android app that’s already running. A profile file is generated for each thread by the online data collector using the ART TI agent. Afterward, the offline data analyzer aggregates the files from various threads, sorts the metrics, and highlights problematic objects that need to be investigated.

The implementation challenges include minimizing measurement overhead and scaling the analysis to multiple threads. In the rest of this section, we discuss how DROIDPERF addresses these challenges.
Figure 5.3: DROIDPERF’s runtime and memory overheads in the unit of times (\(x\)) on various benchmarks.
5.4.1 Online Collector

DROIDPERF supports two modes of monitoring Android apps. By launching DROIDPERF together with the application, Android developers can monitor the end-to-end performance of an Android app. Alternatively, DROIDPERF can attach and detach to any running Android apps to collect the object-centric profile for a while. In particular, this is useful for monitoring long-running Android apps such as Ad & malware blockers or microservices.

All memory-related PMU precise events are accepted by DROIDPERF. Our implementation sets the PMU event to L1 cache miss (MEM_LOAD_UOPS RETIRED:L1_MISS). DROIDPERF is able to collect 20 to 200 samples per second per thread by empirically choosing a sampling period, which has a good tradeoff between runtime overhead and statistical accuracy [Tallent 2010].

DROIDPERF minimizes thread synchronization by allowing every thread to collect PMU samples independently and maintain the calling contexts of the samples in a compact calling context tree [Arnold et al. 1999] (CCT), which merges all the common prefixes of given calling contexts. Only the constructed call path is shared between threads since both object allocation callbacks and PMU sample interrupts need to locate their stack frames. To guarantee the thread safety of DROIDPERF, we use a spin lock to ensure the correctness of the constructed call path across threads.

5.4.2 Offline Analyzer

DROIDPERF’s offline analyzer merges profiles from different threads in order to create compact profile results, which is critical for scalability. As a set of object-centric profiles, each of which is organized as a CCT of a thread, thus they can be merged. A top-down approach is used to merge CCTs. Each thread’s call paths and memory accesses to individual objects can be merged recursively. If the call paths for object allocation are the same, even if they come from separate threads, they are coalesced. Similarly, all the memory accesses to the same objects with their call paths are merged. Also, metrics are summed up when nodes in the CCT are merged. In our experiments, DROIDPERF’s analyzer usually takes less than one minute to merge all profiles.
Table 5.2: Overview of performance optimization guided by DroidPerf.

<table>
<thead>
<tr>
<th>Android Applications</th>
<th>Problematic Code</th>
<th>Type</th>
<th>Optimization</th>
<th>WH (%)</th>
<th>WCM (%)</th>
<th>WEI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS66</td>
<td>AdVpnThread.java</td>
<td>Excessive memory usage in nested loops</td>
<td>Move problematic allocation sites out of loop and reset them upon request</td>
<td>20.6%</td>
<td>16.2%</td>
<td>12.1%</td>
</tr>
<tr>
<td>Rajawali</td>
<td>RenderToTextureFragment.java</td>
<td>Long running services</td>
<td>Avoid using persistent services</td>
<td>11.6%</td>
<td>8.8%</td>
<td>4.0%</td>
</tr>
<tr>
<td>RxAndroid</td>
<td>HandlerScheduler.java</td>
<td>Inefficient using Android broadcast API</td>
<td>Unregister the broadcast receiver which is no longer needed</td>
<td>10.7%</td>
<td>7.0%</td>
<td>6.1%</td>
</tr>
<tr>
<td>Applozic</td>
<td>MessageClientService.java</td>
<td>Runtime configuration</td>
<td>Reuse the visual elements</td>
<td>8.6%</td>
<td>10.4%</td>
<td>14.0%</td>
</tr>
<tr>
<td>GmsCore</td>
<td>TriggerReceiver.java</td>
<td>Runtime configuration</td>
<td>Reuse the visual elements</td>
<td>8.0%</td>
<td>6.5%</td>
<td>12.7%</td>
</tr>
<tr>
<td>Twire</td>
<td>Service.java</td>
<td>Poor data structure</td>
<td>Use Android optimized data structure</td>
<td>6.9%</td>
<td>24.2%</td>
<td>5.1%</td>
</tr>
<tr>
<td>Stream Chat</td>
<td>QueryChannelsLogic.java</td>
<td>Generate duplicated objects</td>
<td>Reuse an object that memorized from the last used point</td>
<td>15.0%</td>
<td>5.8%</td>
<td>7.8%</td>
</tr>
<tr>
<td>Stetho</td>
<td>AsyncPrettyPrinterRegistry.java ChromeDevtoolsServer.java JsonRpcPeer.java ResponseBodyFileManager.java</td>
<td></td>
<td></td>
<td>11.4%</td>
<td>4.3%</td>
<td>8.0%</td>
</tr>
<tr>
<td>MediaPicker</td>
<td>Utility.java</td>
<td></td>
<td></td>
<td>9.4%</td>
<td>4.0%</td>
<td>4.6%</td>
</tr>
<tr>
<td>LeakCanary</td>
<td>AndroidDebugHeapAnalyzer.java</td>
<td></td>
<td></td>
<td>15.0%</td>
<td>5.8%</td>
<td>7.8%</td>
</tr>
<tr>
<td>Foxy Droid</td>
<td>QueryBuilder.java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


5.5 Evaluation

We evaluate DROIDPERF on a Google Pixel 5 (we refer to this smartphone as android-device), which has the mid-range Qualcomm Snapdragon 765G system-on-chip (consisting of eight Kryo 475 cores, an Adreno 620 GPU, and a Hexagon 696 DSP), with 8 GB of LPDDR4X RAM and 128 GB of non-expandable UFS 2.1 internal storage. The cache hierarchy consists of 96KB and 512KB L1 and L2 caches and 16MB L3 cache.

Android apps Because of the lightweight nature of DROIDPERF, we are able to collect profiles from a variety of Android apps obtained from the Awesome Android repository [Stumpp 2022] and the Awesome Android Apps repository [LinuxCafeFederation 2022]. To better show that DROIDPERF is able to tackle real-world problems, we did not cherry-pick Android apps. Instead, we chose many popular Android apps that have thousands of the number of stars and forks, such as Rajawali [raj 2022], RxAndroic [rxa 2022], GmsCore [gms 2022], etc.

Overhead Analysis The runtime overhead (memory overhead) is the ratio of the runtime (peak RSS memory usage) of the execution monitored by DROIDPERF to the runtime (peak RSS memory usage) of the native execution. To quantify the overhead, we apply DROIDPERF to four well-known benchmark suites: EEMBC benchmarks [eem
2022], 0xBenchmark suite [0xb 2022], SciMark 2.0 benchmarks [SPEC 2004] and CaffeineMark 3.0 benchmarks [caf 2022]. We run every benchmark 30 times and compute the average and error bar. Figure 5.3 shows the overhead when DROIDPERF is enabled at a sampling period of 5M. From Figure 5.3 we can see that DROIDPERF typically incurs 14% runtime and 8% memory overhead.

5.6 Case Studies

Given the low overhead, we can easily apply DROIDPERF to a number of popular Android apps, such as DNS66 [dns 2022], Rajawali [raj 2022], Stream Chat [str 2022], GmsCore [gms 2022], MediaPicker [med 2022], Foxy Droid [fox 2022], to name a few. DROIDPERF is able to pinpoint many new data locality issues via its object-centric analysis and guide the optimization. As part of our optimization, we guarantee the program’s correctness with human inspection, and we have tested the transformed code with tests to ensure they are correct. Table 5.2 overviews the performance improvements on several popular Android apps guided by DROIDPERF. It is worth noting that existing profilers may identify the same object allocation as a hotspot in memory usage; however, they do not know whether this allocation site creates locality issues for potential optimization. DROIDPERF, in contrast, quantifies the locality issues for the objects to provide intuitive optimization. We have submitted our optimization patches in several cases and gotten them confirmed or upstreamed, e.g., DNS66 and Rajawali. In the remaining section, we elaborate on how to use DROIDPERF’s GUI in VSCode to profile memory objects in Android apps.

5.6.1 DNS66

DNS66 is a DNS-based host blocker to block ads and malware for Android [dns 2022]. Figure 5.4 shows the snapshot of DROIDPERF’s GUI in VSCode for intuitive analysis. The left half pane of GUI shows the full call path of PMU L1 cache miss event triggered location; the right half shows the Application source code. In the left half pane, the leaf node (highlighted in orange) of call path pinpoints to line 399 in method configurePackages of class AdVpnThread. Under this leaf node, the GUI shows the current live objects information (highlighted in red). In order to expedite the investigation process of spotting potential memory bloat objects and avoid showing a large number of non-important ob-
jects, our GUI only shows the top five allocation sites ranked by their allocation times. As we can see from the live objects information part, the top two allocation sites, objects allowOnVpn and doNotAllowOnVpn on lines 396 and line 397, account for the greatest proportion of the total allocation times in this calling context. On top of the call path, the GUI shows that this memory accessing consumes 32.6% of total heap usage and 19.698% of total L1 cache misses in the entire program. Due to such a high percentage of L1 cache misses and heap usage for accessing these two objects, optimization is worth considering to avoid memory bloat.

The reason for the memory bloat is inappropriate memory management. The workflow of DNS66 is that it first establishes a VPN service, with routes for all DNS servers diverted to it. The VPN service then intercepts all android packages for the servers and forwards any DNS queries that are not blacklisted. When processing with a large number of android packages, DNS66 creates a separate object pair of allowOnVpn and doNotAllowOnVpn for every android package, resulting in memory bloat. Since the life cycles of different allowOnVpn and doNotAllowOnVpn instances do not overlap, which means using a singleton pattern of these objects (i.e., allocating a single object instance and reusing it without creating more instances) is safe and avoids memory bloat. To apply the singleton pattern, we hoist allowOnVpn and doNotAllowOnVpn allocation out of configurePackages method, which is thread-safe. Our optimization reduces the total cache misses by 16.2%, and the JVM heap memory consumption is reduced by 20.6%.

5.6.2 RxAndroid

RxAndroid is one of the most popular libraries for enabling Reactive Programming in Android development [rxa 2022]. DROIDPERF spots a problematic object, scheduled, which is allocated on line 77 in method schedule of class HandlerScheduler, as shown in Listing 5.8. DROIDPERF reports that 14.5% of total cache misses are caused by accessing the method schedule.

Further investigation has revealed that the memory bloat is due to improper memory management. The reported object scheduled is a ScheduledRunnable type object, which is an identifier for Android UI actions that need to be performed on Android’s main thread. By checking the source code, we found that the program keeps allocating ScheduledRunnable identifier scheduled for every UI action. Like other cases, different instances of this object have disjoint life intervals, so we change this object to a static one. When using a new scheduled object, we clear and reuse the static object to avoid
Figure 5.4: The object-centric GUI view of DNS66 shows two problematic objects’ allocation sites in DROIDPERF’s Flame Graphs GUI in VSCode.

Listing 5.8: DROIDPERF identified the scheduled object suffering from memory bloat in RxAndroid.

repeated allocation upon each iteration. This optimization reduces total cache misses by 8.8% and the JVM heap memory consumption by 11.6%.

5.6.3 Stream Chat

Stream chat is a library for building Android chat and messaging applications [str 2022]. DROIDPERF locates many object allocations occurring in method refreshChannel of class QueryChannelsLogic. DROIDPERF reports that accessing the methods refreshChannel accounts for 23% of total cache misses in the entire program.

In light of further investigation, we found that the method refreshChannel is invoked frequently by some runtime configuration changes, such as screen orientation changes, keyboard attachments, and language switching. To avoid excessive refreshing chat chan-
private static int getCameraPhoto(File file) throws IOException {
    ExifInterface exif = new ExifInterface(file.getAbsolutePath());
    int orientation = exif.getAttributeInt(...);
    switch (orientation) {
        ... // computations based on the contents of object exif
    }
}

Listing 5.9: DROIDPERF pinpoints the exif object with many replicas in MediaPicker.

nels, we optimize the code so as don’t invoke the method refreshChannel if there is no request for refreshing channels. As a result of this optimization, the total cache misses are reduced by 6.5%, and the JVM heap memory consumption is reduced by 8%.

5.6.4 MediaPicker.

MediaPicker is a popular Android Library that lets users to select multiple images, video or voice [med 2022]. DROIDPERF identifies an object that has many replicas, exif, which is allocated on line 111 and accessed on line 112 in method getCameraPhotoOrientation of class Utility, as shown in Listing 5.9. DROIDPERF reports that accessing the method getCameraPhotoOrientation accounts for 9.5% of total cache misses.

The generated replicas are due to the users’ habit of picking images. Phone users usually take several photos for a same scene and pick the best photo among these similar photos. Consequently, to view these similar photos, phone users pick each of these similar photos in MediaPicker. Since these similar photos share the similar metadata (including date created, author, file name, content, orientation and more), the program keeps generating the object exif with the same contents, and processing its contents every time the user picks a new photo. To avoid the redundant operations on object exif, we check the fields in object exif. If object exif is unchanged, we reuse the object exif memoized from the last iteration, which avoids the redundant computation. Through this optimization, the total cache misses are reduced by 5.8%, and the JVM heap memory consumption is reduced by 15%.

5.6.5 Foxy Droid

Foxy Droid is an installable catalogue of FOSS (Free and Open Source Software) applications for the Android platform [fox 2022]. DROIDPERF pinpoints query, an object allocated on line 42 in method query of class QueryBuilder with many replicas, as shown
Listing 5.10: DROIDPERF identified the `query` object with replicas in Foxy Droid.

in Listing 5.10. DROIDPERF reports that accessing the method `query` accounts for 7.7% of total cache misses in the entire program.

The replicas come from the database query sequence. Given a query, Foxy Droid uses `rawQuery` method of `SQLiteDatabase` class to retrieve data. Investigating source code further, we found that many consecutive queries for the database are the same. Consequently, the program keeps allocating and generating the same contents of `query` objects. To optimize the code, we only perform the database operations when `query` is changed; Otherwise, we reuse the query result from the prior iteration. As a result of this optimization, cache misses are reduced by 4% and JVM heap memory consumption is reduced by 9.4%.

5.7 Related Work

5.7.1 HotSpot Analysis

[Xu et al. 2011b] investigate the diverse usage behaviors of individual mobile apps at scale, e.g., locality, diurnal behaviors, and mobility patterns. The work [Huang et al. 2010; Shepard et al. 2010] collect traces from real smartphone users, focusing on characterization at only IP and higher layers. [Falaki et al. 2011] develop SystemSens to capture usage context, e.g., CPU and memory, of smartphone. The Android Profiler [and 2021] provides real-time data to help developers to understand how the application uses CPU, memory, network, and battery resources. Dalvik Debug Monitor Server (DDMS) [DDM 2021] provides port-forwarding services, screen capture on the device, thread and heap information on the device, logcat, process, and radio state information, incoming call and SMS spoofing, and location data spoofing.

Perfetto [per 2021] offers services and libraries for recording system-level and app-
level traces. Perfetto provides Heapprofd, a tool that tracks native heap allocations and deallocations (by hooking calls to malloc/free and C++’s operator new/delete) of an Android process within a given period. However, Heapprofd does not track the object allocations that occurred in JVM-based languages. Perfetto also provides Java heap profiling, which reports full retention graphs of managed objects but not call stacks. Without full call stacks, Perfetto does not distinguish allocations by the same routine called from different contexts. For example, an allocation in a common library routine called from two different user code locations. As a result, Android developers lack the precise knowledge of which part of their source code has a conspicuous impact on the locality. Perfetto’s Java heap profiling can also get full snapshots of the managed heap retention graph. However, the retention graph can only output summaries about a trace (e.g., the heap memory usage at different execution time points). In contrast, DroidPerf can associate JVM heap memory information with allocated objects to identify the object locality issues. More importantly, to guarantee the optimization benefits, we have shown the necessity of measuring the L1 cache miss with PMU in section 5.1.1. However, Perfetto lacks performance statistics collected from the hardware. Optimizing without quantifying the underlying hardware can often only yield trivial or negative speedups.

PowerTutor [Zhang et al. 2010] profile the energy consumption of running apps on Android. [Flinn et al. 1999] develop a workstation power modeling technique that assigns energy consumption to procedures within a mobile process. Some power modeling techniques [Isci et al. 2003; Joseph et al. 2001] require deep knowledge of the relationship between processor functional unit activities and their resulting power consumptions.

Although these existing hotspot analyses identify execution hotspots in CPU time, memory usage, battery resources, or energy consumption, they fail to tell whether a resource is being used in a productive manner and contributes to a program’s overall efficiencies. Hence, a heavy burden is on users to make a judgment on whether the reported hotspots are actionable.

5.7.2 Inefficiency Analysis

[Qian et al. 2011] present ARO, a tool used to locate the inefficient resource usage for smartphone apps by considering the cross-layer information ranging from radio resource control to application layer. [Wei et al. 2012] propose ProfileDroid, which is used to profile Android apps in different layers, e.g., user interaction, OS, and network. [Falaki et al. 2010a] investigate network logs from 43 smartphones and found commonly used
app ports. In their extended work [Falaki et al. 2010b], they also analyze the diversity of smartphone usage. [Son et al. 2011] show how to enhance the performance of the JAVA applications by using the Android NDK. The Datadog Profiler [dat 2021] debugs the root cause of app crashes, whether it is because of 3rd party libraries, network requests, or large media files.

A group of studies attempted to improve the performance of mobile apps via OS infrastructure support [Cuervo et al. 2010; Ficek et al. 2010; Higgins et al. 2010]. [Cuervo et al. 2010] present MAUI, which supports fine-grained code offload to maximize energy savings with minimal burden on the programmer. [Ficek et al. 2010] propose SS7Box, a tool that signals mobile devices by network providers via notification channel to save resources. [Higgins et al. 2010] provide a clean intermediate interface for apps by the OS.

Orthogonal to the aforementioned inefficiency analysis profilers, DROIDPERF addresses a different inefficiency problem with the usage of PMU. To the best of our knowledge, DROIDPERF is the first lightweight sampling-based profiler to pinpoint memory inefficiencies on Android.

5.7.3 Lightweight tools

[Barbhuiya et al. 2020] propose a lightweight anomaly-based intrusion detection system that can detect zero-day malware efficiently and effectively. [Zhao et al. 2012] propose a lightweight framework named RobotDroid that uses active learning on Android apps to induce an accurate detection model with minimal labeled samples. [Arp et al. 2014] design a lightweight method of Android malware detection, applying linear-time static analysis and learning techniques for efficiency. [Yang et al. 2015] propose a lightweight approach to profile smartphones’ power consumption which can integrate energy data into the development and testing process. [Kamiyama et al. 2018] propose lightweight online logging on devices to test the energy-efficiency of applications in real user environments. [Qiong et al. 2016] propose a lightweight and automatic approach to estimate the method-level energy consumption for Android apps. [Dai et al. 2013] develop a lightweight technique for extracting fingerprints based on identifying invariants in the generated network traces.

Although these existing works are lightweight, none of these tools focus on memory issue detection.
5.8 Summary

In this work, we present DROIDPERF, a lightweight Android profiler. It is designed to perform object-centric analysis to uncover memory inefficiencies in Android apps running on the ART platform. DROIDPERF utilizes the lightweight hardware PMU available on commodity Android CPUs. DROIDPERF requires no changes to hardware, OS, Dalvik virtual machine, ART, or Android app source code. DROIDPERF incurs minimal overhead, typically 14% runtime overhead and 8% memory overhead. DROIDPERF is therefore applicable to production environments. DROIDPERF identifies six types of inefficiencies that lead to locality issues in Android apps: high memory usage in loops, persistent services, improper use of Android APIs, excessive use of Android unoptimized data structure, runtime configuration changes, and duplicated objects. Guided by DROIDPERF, we are able to perform code optimization, obtaining significant performance gains. Additionally, many optimization patches are acknowledged or upstreamed by the software developers.
Chapter 6

Conclusions and Future Directions

In this dissertation, we investigate how to identify object-level inefficiencies that occurred on JVM and Android Runtime and obtain insightful optimization guidance by developing lightweight object-centric profiling tools. Specifically, we work on the following topics:

We first present DJXPerf, the very first lightweight Java profiler that performs object-centric analysis to identify data locality issues in Java applications. DJXPerf leverages the lightweight Java byte code instrumentation and the hardware PMU available in commodity CPU processors. DJXPerf works for off-the-shelf Linux OS and Oracle Hotspot JVM, as well as unmodified Java applications. DJXPerf incurs low overhead, typically 8.5% in runtime and 6% in memory. These features make DJXPerf applicable to the production environment. DJXPerf is able to identify a number of locality issues in real-world Java applications. Such locality issues arise due to traditional spatial/temporal data locality, memory bloat, as well as NUMA locality. Guided by DJXPerf, we are able to perform optimization, which yields nontrivial speedups. Moreover, many optimization patches are confirmed or upstreamed by the software developers.

Next, we design and develop OJXPerf, the first lightweight profiler to identify object replicas in Java applications. We show both theoretical analysis and implementation details of OJXPerf to warrant accurate analysis, low measurement overhead, and rich optimization guidance. As a unique feature, OJXPerf combines the use of performance monitoring units, debug registers and lightweight byte code instrumentation for statistical object replica detection. OJXPerf does not require any modification to software and hardware stacks. With the evaluation of more than 50 Java applications, we show OJXPerf minimizes false positives and incurs 9% and 6% runtime and memory over-
heads, respectively. The case study section shows that OJXPerf is able to detect many object-level redundancies in real-world Java applications. Guided by OJXPerf, we can optimize many benchmarks and real-world applications, yielding significant speedups. Many optimization patches are confirmed or upstreamed by the software developers.

Lastly, we target on object-level inefficiencies occurred on Android Runtime by developing DroidPerf, a lightweight Android profiler. It is designed to perform object-centric analysis to uncover data locality problems in Android applications. DroidPerf utilizes the lightweight hardware PMU available on commodity Android CPUs. DroidPerf requires no changes to hardware, OS, Dalvik virtual machine, ART, or Android application source code. DroidPerf incurs minimal overhead, typically 14% runtime overhead and 8% memory overhead. DroidPerf is therefore applicable to production environments. DroidPerf identifies six types of inefficiencies that lead to locality issues in Android applications: high memory usage in loops, persistent services, improper use of Android APIs, runtime configuration changes, and duplicated objects. These locality issues are a result of traditional spatial/temporal data locality as well as memory bloat. Guided by DroidPerf, we are able to perform optimization, providing significant performance gains. Additionally, many optimization patches are acknowledged or upstreamed by the software developers.

For future work, we are considering the following research directions:

- We will extend our object-level inefficiencies to inefficient use of allocated objects, which can also have a conspicuous impact on performance. For example, an object is created and holding many data, but it is looked up only a few times. In this case, programmers may be able to determine which data in this object will definitely not be retrieved by inspecting the code, and then avoid adding these data to this object. Also, we will extend DJXPerf and OJXPerf to other popular managed languages, such as Python and JavaScript.

- Unlike Intel has the X86 Encoder Decoder (XED), Android Runtime lacks a software library for encoding and decoding ARM instructions. To have more insights on profiling Android applications with a performance monitor unit (PMU), one future direction for DroidPerf is to develop a software library that can take sequences of a fixed number of bytes along with machine mode information and produce a data structure describing the opcode, operands, and flags. With such detailed instructions information, DroidPerf could obtain the memory address that each PMU sample triggered, then DroidPerf can give more direct profiling results on various inefficiencies.
References


Baum, David abd Baum, E. (2016). Twitter migrates core infrastructure to java virtual machine and supports more than 400 million tweets a day. https://go.java/twitter.html.


