OJXPerf: Featherlight Object Replica Detection for Java Programs

Bolun Li
North Carolina State University
Raleigh, North Carolina, USA
bli35@ncsu.edu

Hao Xu
College of William and Mary
Williamsburg, Virginia, USA
hxu07@email.wm.edu

Qidong Zhao
North Carolina State University
Raleigh, North Carolina, USA
qzhao24@ncsu.edu

Pengfei Su
University of California, Merced
Merced, California, USA
psu9@ucmerced.edu

Milind Chabbi
Scalable Machines Research
milind@scalablemachines.org

Shuyin Jiao
North Carolina State University
Raleigh, North Carolina, USA
sjiao2@ncsu.edu

Xu Liu
North Carolina State University
Raleigh, North Carolina, USA
xliu88@ncsu.edu

Abstract

Memory bloat is an important source of inefficiency in complex production software, especially in software written in managed languages such as Java. Prior approaches to this problem 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.

We develop 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 incurs 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 noticeable reduction in memory usage (up to 11%) and a significant speedup (up to 25%).

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510083

ACM Reference Format:
Bolun Li, Hao Xu, Qidong Zhao, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. 2022. OJXPerf: Featherlight Object Replica Detection for Java Programs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510083

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 [21, 38, 57, 59–62, 64, 65] and improve GC [5, 10, 27, 36, 49, 66].

However, there is another cause of memory bloat and inefficiency that has hardly been studied — replica objects — which is the focus of this paper. 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* [6], and often results in redundant re-computations [38, 54].

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, 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.

1.1 Observation

With the help of OJXPerf, Figure 1 quantifies the ratio of object replicas over the total number of objects in several real applications listed at [32] and two popular Java benchmark suites—Dacapo 9.12 [2] and Renaissance [44], showing that replicated objects are pervasive in modern Java software packages. Based on many case studies investigated in this paper, 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 1 shows a problematic method `readNext` from Parquet-MR [29], 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 [51] and LDoctor [50], 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 [45] 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 paper:

- 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.

1.2 Paper Contributions

In this paper, 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.

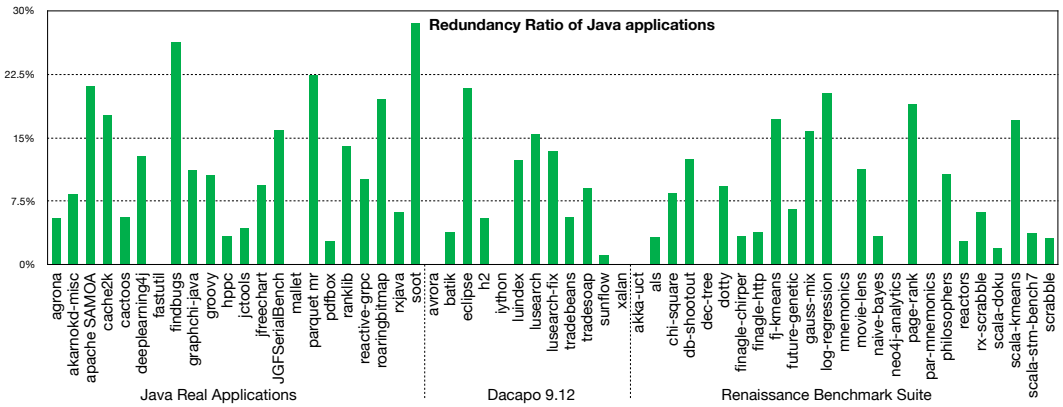


Figure 1: Percentage of replicated objects over all the objects in various Java applications.

```
90 private void readNext() {
91   ...
92   currentBuffer = new int[currentCount];
93   ►byte[] bytes = new byte[numGroups * bitWidth];
94   ►new DataInputStream(in).readFully(bytes);
95   for (int valueIndex = 0, byteIndex = 0; ...; ...) {
96     ►packer.unpack8Values(bytes, byteIndex, currentBuffer,
           valueIndex);
97   }
98 }
```

Listing 1: Object replicas in Parquet MR. The replica object `bytes` is allocated on line 93, initialized on line 94 and used on line 96.

- 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 (Dacapo [35], NPB [16], Grande [9], SPECjvm2008 [14], and the most recent Renaissance [44]) 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.

1.3 Paper Organization

The paper is organized as follows. Section 2 covers the related work and distinguishes OJXPerf. Section 3 offers some background knowledge. Section 4 depicts OJXPerf's methodology. Section 5 describes the implementation details of OJXPerf. Section 6 discusses the theoretical guarantee of OJXPerf's analysis accuracy. Section 7 evaluates OJXPerf's accuracy and overhead. Section 8 describes case studies of OJXPerf. Section 9 discusses the threats to validity. Finally, Section 10 presents our conclusions.

Tools	Redundancy detection	Cross-object analysis	Probabilistic analysis	Sampling with hardware support	Runtime overhead**
OJXPerf	✓	✓	✓	✓	9%
OEP [37]	✓	✓*	×	×	202×
Cachetor [31]	✓	×	×	×	133×
Toddler [39]	✓	×	×	×	16×
JOLT [48]	×	✓	✓	×	3%
Memoizelt [15]	✓	×	×	×	202×
Ldoctor [50]	✓	×	×	×	5%

*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.

Table 1: Comparing OJXPerf with other state-of-the-art inefficiency analysis tools/approaches.

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 1.

2.1 Software Approaches

Hotspot Analysis. Netbeans Profiler [42], JProfiler [19], YourKit [24], VisualVM [13], and Oracle Developer Studio Performance Analyzer [11] 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.

Inefficiency Analysis. Unlike hotspot analysis, inefficiency analysis tools identify code regions leading to resource wastage instead of resource usage. Cachetor [31] combines value profiling and dependence profiling to pinpoint operations that repeatedly generate an identical value. MemoizeIt [15] identifies methods that repeatedly perform identical computation. JOLT [48] identifies and optimizes object churn in a virtual machine. Toddler [39] identifies redundant memory load operations in loop nests. The follow-up work [50] 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.* [58] 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 [63] 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 [56] to identify reusable data structures to avoid frequent object allocations. OEP [37] 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.

2.2 Hardware Approaches

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

Hotspot Analysis. Linux Perf [34], Async-profiler [43], and Oprofile [3] 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. Sweeney *et al.* [20] develop a system that provides a graphical interface to alleviate the difficulty in interpreting PMU results. Hauswirth *et al.* [25] present vertical profiling that captures and correlates performance problems across multiple execution layers (application, VM, OS, and hardware). Georges *et al.* [23] study methods exhibit similar and dissimilar behaviors by measuring the execution time for each method invocation using PMU. Lau *et al.* [33] present a technique that allows a VM to determine whether an optimization improved or degraded by measuring CPU cycles. Remix [18] employs PMU to identify inter-thread false sharing on the fly. JXPerf [51] 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.

3 Background

We introduce some essential facilities that OJXPerf leverages based on Java virtual machines (JVM) and CPU processors.

Java Virtual Machine Tool Interface (JVMTI). JVMTI, a native programming interface of the JVM, is loaded during the initialization of the JVM. JVMTI provides a VM interface for the full breadth of tools that need access to VM state, including but not limited to profiling, debugging, monitoring, thread analysis, and coverage analysis tools.

Hardware Performance Monitoring Unit (PMU). PMU is hardware built inside a processor to measure its performance parameters. We can measure parameters like instruction cycles, cache hits, cache misses, branch misses, and many others, depending on the supported hardware. PMU supports lightweight measurement.

Intel processors also support Precise Event-Based Sampling (PEBS) [30]. PEBS is a profiling mechanism that logs a snapshot of the processor state at the time of the event, allowing users to attribute performance events to actual instruction pointers (IPs). Also, PEBS provides an effective address (EA) at the time of the sample when the sample is for a memory load or store instruction. In PEBS, the event type may be chosen from an extensive list of performance-related events to monitor, e.g., cache misses, remote cache hits, branch mispredictions. AMD processors provide similar capabilities via instruction-based sampling.

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 [17, 47] 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.

4 Methodology

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

Definition 4.1. **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 (70~300× slowdown reported in [26]).

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 different[1]. 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.

```
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 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 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 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 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 $\{O_1, O_2, O_3, O_4\}$.

[1]two objects of different size, e.g., arrays allocated in the same context are easily ruled out of duplication due to size difference

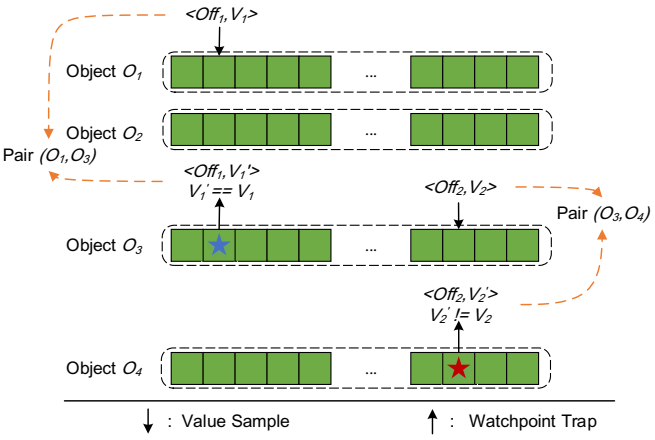


Figure 2: Watchpoint scheme for object replica detection. Off_1 (Off_2) presents memory offset with value V_1 (V_2) for Object O_1 (Object O_3). When a watchpoint trap of memory access happens at offset Off_1 (Off_2), OJXPerf compares their corresponding values V_1 and V_1' (V_2 and V_2').

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 6.

5 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. Figure 3 overviews the design of these two agents. 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.

5.1 Java Agent

The Java agent monitors object allocation, which leverages `java.lang.instrument` API and `ASM` framework. The Java agent inserts pre- and post-allocation hooks to intercept each object allocation. Then, a user-defined callback is invoked on each allocation to obtain the object information, such as the object pointer, type, and size. For a given Java class we want to instrument, the Java agent scans the byte code of this class, instruments `new`, `newarray`, `anewarray`, and `multianewarray`, and obtains the memory range of every object following an existing technique [1].

How to present an object allocation is a challenging question. We adopt a simple and perhaps the most intuitive approach that

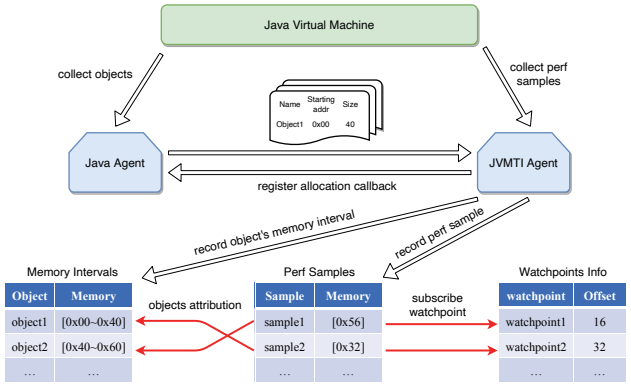


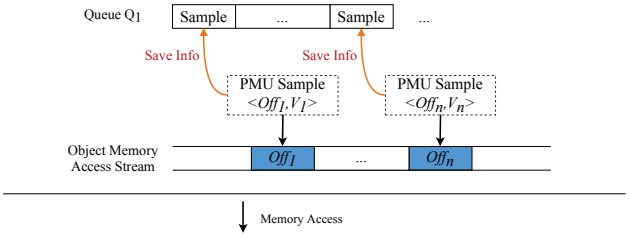
Figure 3: Overview of OJXPerf's profiling.

developers can identify with — the allocation context leading to the object allocation. A Java application can often 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. OJXPerf leverages the `AsyncGetCallTrace()` [40] API provided by Oracle Hotspot JVM to determine the calling contexts at any point during the execution. OJXPerf then inserts a $\langle key, value \rangle$ pair into a map $\mathcal{M}$, where key is the memory range and $value$ is the allocation context.

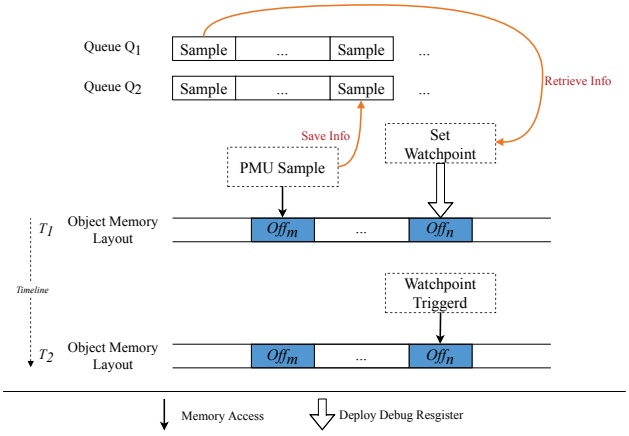
5.2 JVMTI Agent

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 [52]. Moreover, the JVMTI agent captures the calling contexts for both PMU samples and object allocations described in Section 5.1. To minimize synchronization, each thread collects PMU samples independently and maintains a thread-local compact calling context tree (CCT) [7], 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 $\langle Off, V \rangle$ to represent it and adds this tuple to queue Q_1, as shown in Figure 4a. 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 stored at the sampled memory address. Upon a sample taken from O_2, OJXPerf not only adds a tuple $\langle Off_m, V_m \rangle$ to queue Q_2, but also retrieves a sample ($\langle Off_n, V_n \rangle$) from queue Q_1 and uses a debug



(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: Workflow of object-level redundancy detection.

register to set up a watchpoint at the offset Off_n of O_2, as shown in Figure 4b. OJXPerf compares values at the the same offset 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 [46], 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 [53, 55] has shown that reservoir sampling guarantees the fairness of the measurement with a limited number of debug registers.

5.3 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 6.

5.4 Discussions

As a sampling-based approach, OJXPerf may introduce false positives and false negatives, which are elaborated in Section 7.1. Our theoretical analysis to be described in the next section bounds the analysis accuracy.

6 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 6.1. **Replication Factor (RF) θ:** *For a set of objects that suffer from object replication, the replication factor θ is the probability of last accessed object to be bit-wise same as the current accessed object. We define θ 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.*

$$\theta = \frac{\text{num equivalent access times}\langle Object O\rangle}{\text{num equivalent + num different access times}\langle Object O\rangle} \tag{1}$$

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 ($O_2^{offset} = O_1^{offset}$), 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:

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

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

Then, the θ can be rewritten using A, B, C as:

$$\theta = \frac{A+B}{A+B+C} = A + B \tag{2}$$

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

$$\alpha = \frac{B}{B+C} \geq \frac{B}{A+B+C} = B \tag{3}$$

Combine Equation (2) and Inequality (3), we have:

$$A = \theta - B \geq \theta - \alpha \tag{4}$$

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 $\binom{X}{2}$ object pairs. Among these $\binom{X}{2}$ object pairs, there will be $\sum_{n=1}^{N}\binom{X_n}{2}$ identical object pairs. Considering we can estimate identical object pairs ratio by $\frac{A}{A+B+C} = A$ and Inequality (4), we can state:

$$\frac{\sum_{n=1}^{N}\binom{X_n}{2}}{\binom{X}{2}} = A \geq \theta - \alpha \tag{5}$$

Then, $\frac{\sum_{n=1}^{N}\binom{X_n}{2}}{\binom{X}{2}}$ can be derived as:

$$\frac{\sum_{n=1}^{N}\binom{X_n}{2}}{\binom{X}{2}} = \frac{\sum_{n=1}^{N} X_n(X_n-1)}{X(X-1)} < \frac{\sum_{n=1}^{N} X_n^2}{X^2} \tag{6}$$

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

$$\begin{aligned}\sum_{n=1}^{N}(\frac{X_n}{X})^2 &= \sum_{n=1}^{N-1}(\frac{X_n}{X})^2 + \frac{X_N}{X} * \frac{X_N}{X} \\ &= \sum_{n=1}^{N-1}(\frac{X_n}{X})^2 + (1 - \sum_{n=1}^{N-1}\frac{X_n}{X}) * \frac{X_N}{X} \\ &= \frac{X_N}{X} - \sum_{n=1}^{N-1}\frac{X_n}{X}(\frac{X_N - X_n}{X})\end{aligned} \tag{7}$$

Since $X_N > X_{N-1} > X_{N-2} > ... > X_1$, we have

$$\sum_{n=1}^{N-1}\frac{X_n}{X}(\frac{X_N - X_n}{X}) > 0 \tag{8}$$

Furthermore, combining (5), (6), (7), (8), we then obtain

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

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 5, we have:

$$\frac{\sum_{n=1}^{N}\binom{X_n}{2}}{\binom{X}{2}} = A > \frac{\binom{X_N}{2}}{\binom{X}{2}} \tag{9}$$

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

$$\begin{aligned}\frac{\binom{X_N}{2}}{\binom{X}{2}} = \frac{X_N(X_N-1)}{X(X-1)} &= \frac{X_N}{X}(\frac{X_N}{X} + \frac{X_N - 1}{X-1} - \frac{X_N}{X}) \\ &= \frac{X_N}{X}(\frac{X_N}{X} - \frac{X - X_N}{X(X-1)})\end{aligned} \tag{10}$$

We also have:

$$\frac{X - X_N}{X(X-1)} = \frac{1}{X-1} - \frac{1}{X-1} * \frac{X_N}{X} \tag{11}$$

We then denote $\frac{X_N}{X} = s$ and $\frac{1}{X-1} = t$, equation 10 can be rewritten as:

$$\frac{\binom{X_N}{2}}{\binom{X}{2}} = s(s - t + st) \tag{12}$$

Combining with equation 9, we have this in-equation:

$$A > s(s - t + st) = (t+1)s^2 - st > s^2 - st \tag{13}$$

Solving this in-equation, we then have:

$$s < \frac{t + \sqrt{t^2 + 4A}}{2} \tag{14}$$

Focusing on A in equation 2 and 3, we have:

$$A = \theta - B = \theta - \alpha * (B + C) = \theta - \alpha * (1 - A) \tag{15}$$

Solving it, we then have:

$$A = \frac{\theta - \alpha}{1 - \alpha} \tag{16}$$

Based on the in-equation 14, we have:

$$\begin{aligned} s < \frac{t + \sqrt{t^2 + 4\frac{\theta-\alpha}{1-\alpha}}}{2} &= t/2 + \sqrt{t^2/4 + \frac{\theta - \alpha}{1 - \alpha}} \\ &= \frac{1}{2(X-1)} + \sqrt{\frac{1}{4(X-1)^2} + \frac{\theta - \alpha}{1 - \alpha}} \end{aligned} \tag{17}$$

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 6.2. **Lower Bound Factor (LBF) ω:** *ω 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 6.3. **Upper Bound Factor (UBF) γ:** *γ 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 7.

In the applications we evaluated, we have not seen an application with a very high α (we compute α 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 α 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 paper.

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.

7 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 cache, a private 256KB L2 cache, a shared 46MB L3 cache, and 128GB main memory. OJXPerf is compatible with JDK 1.5 and any of its successors. We run all applications with JDK 1.8.0_161.

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 [32], such as the Renaissance benchmark suite [44], Soot [41], parquet MR [29], Findbugs [45], Eclipse Deeplearning4J [28], JGFSerialBench [9], RoaringBitmap [8], Apache SAMOA [22], 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 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 8 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 2 summarizes the new findings identified by OJXPerf, which we further elaborate in Section 8. In Table 2, we report replication factor θ, α, and lower bound factor ω, which are defined in Section 6. Table 2 shows that α ranges from 0% to 53% and θ ranges from 65% to 100%, respectively. As a result, the lower bound factor ω 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 [51].

From Table 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 [12] and perf [34]. We detected the object replicas as shown in Table 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 ~37% of whole-program object replicas on average.

7.1 False Positives and Negatives

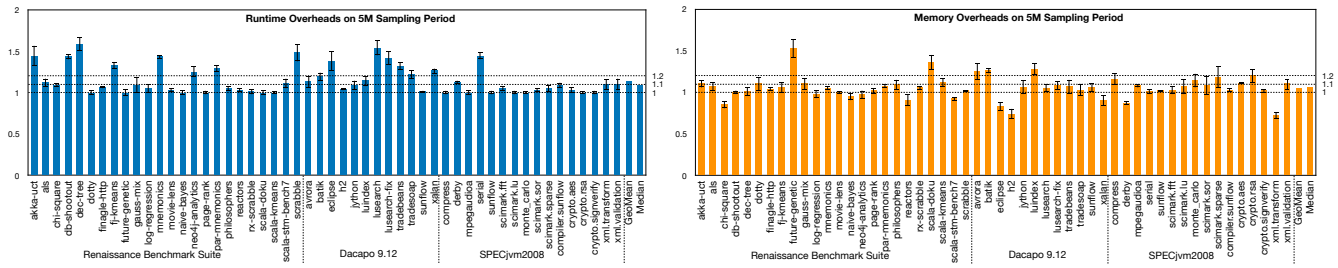
As a sampling-based tool, OJXPerf can introduce false negatives — missing some object replicas. However, the statistics theory guarantees the high probability of capturing object replicas that occur frequently. The false negatives do not hurt the insights obtained from OJXPerf because optimizing infrequently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

OJXPerf can also introduce false positives — reporting object replicas that are not replicated because OJXPerf uses sampling

Real-world Applications	Inefficiency				Optimization			
	Problematic Code	θ (RF)	α	ω (LBF)	WS (×)	WH (%)	WCM (%)	WEI (%)
Soot [40]	PhaseOptions.java (84)	83.9%	20.8%	63.1%	1.17±0.02	7.5%	11.2%	10.3%
Deeplearning4J SameDiff [28]	ArrrayOptionsHelper.java (59)	64.7%	47.2%	17.5%	1.09±0.02	1.1%	5.1%	7.9%
Deeplearning4J AlphaGo Zero [4]	BasicNDArrayCompressor.java (57)	81.3%	12.5%	68.8%	1.15±0.01	9.5%	4.3%	6.0%
FindBugs-3.0.1 [44]	BasicAbstractDataflowAnalysis.java (183)	70.3%	44.7%	25.6%	1.25±0.03	11.2%	21.3%	10.7%
fj-kmeans [43]	JavaKMeans.java (220)	76.1%	39.1%	37.0%	1.08±0.04	4.8%	2.9%	4.1%
JGFSerialBench [9]	JGFSerialBench.java (371)	68.8%	53.6%	15.2%	1.1±0.03	3.4%	5.7%	9.3%
RoaringBitmap [7]	MutableRoaringArray.hava (288)	100.0%	0.0%	100.0%	1.09±0.01	9.2%	8.2%	11.4%
Apache SAMOA [22]	SerializeUtils.java (97)	93.5%	18.1%	75.4%	1.16±0.01	8.6%	17.7%	12.1%

WS: Whole-program speedup. WH: Whole-program heap usage reduction. WCM: Whole-program L1 cache miss reduction. WEI: Whole-program executed instructions reduction

Table 2: Overview of performance optimization guided by OJXPerf.



(a) Runtime overheads. **(b) Memory overheads.**

Figure 5: OJXPerf's runtime and memory overheads in the unit of times (×) on various benchmarks.

		Replica Detection	
		Replicated	Not Replicated
Actual	Replicated	8	0
	Not Replicated	3	48
Correctness		(48 + 8) / (0 + 8 + 3 + 48) = 94.9%	
False positive rate		3 / (3 + 48) = 5.9%	

Table 3: Accuracy for OJXPerf's replica detection.

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 3. From the table, we can see that OJXPerf incurs 5.9% false positives.

7.2 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 [44], Dacapo 9.12 [2], and SPECjvm2008 [14]. We run all benchmarks with four threads. We run every benchmark 30 times and compute the average and error bar. Figure 5 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, scrabble, akka-uct, db-shootout, dec-tree, neo4j-analytics).

8 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.

8.1 Soot

Soot is a Java optimization framework, which uses containers extensively [41]. We run Soot-3.3.0 using the bytecode of the DaCapo benchmark avrora as input. Figure 6 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 6 shows one problematic object `st` (highlighted in blue), which is accessed on line 84 in method `getPhaseOptions`

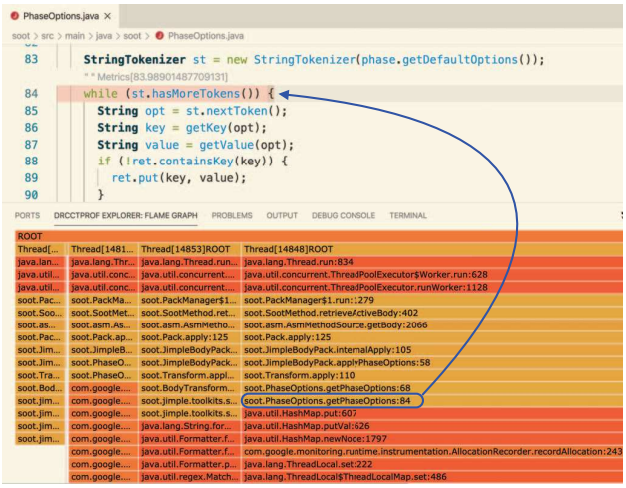


Figure 6: The Flame Graphs GUI of Soot shows a problematic object `st` with its accesses in full call stacks.

of class `PhaseOptions` with many replicas (its replication factor θ 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 `st` 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.

8.2 Eclipse Deeplearning4J – SameDiff

Eclipse Deeplearning4J integrates with Hadoop and runs on several backends [28]. 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 the training phase and reports that the replication factor θ 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 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

```
58 public static ArrayType arrayType(long[] shapeInfo) {
59   ▶val opt = Shape.options(shapeInfo);
60   if (hasBitSet(opt, ATYPE_SPARSE_BIT))
61     return ArrayType.SPARSE;
62   else if (hasBitSet(opt, ATYPE_COMPRESSED_BIT))
63     return ArrayType.COMPRESSED;
64   else if (hasBitSet(opt, ATYPE_EMPTY_BIT))
65     return ArrayType.EMPTY;
66   else
67     return ArrayType.DENSE;
68 }
```

Listing 2: OJXPerf identified the `shapeInfo` array with replicas in Deeplearning4J SameDiff.

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.

8.3 Eclipse Deeplearning4J – AlphaGo Zero

We also run Deeplearning4J using AlphaGo Zero model [4], 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 θ is 81.3%), as shown in Listing 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 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 $(1.15 \pm 0.01)\times$ speedup to the entire program.

8.4 FindBugs-3.0.1

FindBugs looks for code instances that are likely to be errors [45]. 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

```
58 public void init(INDArray parameters) {
59   ...
60   for (int vertexIdx : topologicalOrder) {
61       paramsViewForVertex[vertexIdx] = parameters.get(NDArrayIndex
           .interval(0,0,true));
62       //get method calls loadCompressors to decompress data
63   }
64 }
65 ▶protected Map<String, NDArrayCompressor> codecs = new
       ConcurrentHashMap<>();
66 protected void loadCompressors() {
67   ...
68   for (NDArrayCompressor compressor : compressors) {
69     ▶codecs.put(compressor.getDescriptor(), compressor);
70   }
71 }
```

Listing 3: OJXPerf identified the `codecs` map with replicas in Deeplearning4J AlphaGo Zero.

```
176 public/* final */Fact getStartFact(BasicBlock block) {
177   return lookupOrCreateFact(startFactMap, block);
178 }
179 public/* final */Fact getResultFact(BasicBlock block) {
180   return lookupOrCreateFact(resultFactMap, block);
181 }
182 private Fact lookupOrCreateFact(Map<BasicBlock, Fact> map,
       BasicBlock block) {
183   ▶Fact fact = map.get(block);
184   if (fact == null) {
185     fact = createFact();
186     map.put(block, fact);
187   }
188   return fact;
189 }
```

Listing 4: The source code highlighed by OJXPerf shows the object `block` with replicas in Findbugs.

`lookupOrCreateFact` of class `BasicAbstractDataflowAnalysis`, as shown in Listing 4.

The replicas come from the algorithm of data-flow analysis used in FindBugs. Findbugs 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) method is usually invoked with the same input `BaiscBlock block`. OJXPerf reports that the replication factor θ 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 $(1.25 \pm 0.03)\times$ speedup to the entire program.

8.5 fj-kmeans

fj-kmeans is a benchmark from Renaissance Suite, used to run the k-means algorithm [44]. 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 `computeDirectly` of class `JavaKMeans`, as shown in Listing 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

```
1 protected Map<Double[], List<Double[]>> computeDirectly() {
2   ▶return collectClusters(findNearestCentroid());
3 }
4 private int[] findNearestCentroid() {
5   ▶final int[] result = new int[taskSize];
6   ...
7   for(...) {
8   final Double[] element = data.get(dataIndex);
9     for(...) {
10      final double distance = distance(element, centroids.get(
            centroidIndex));
11      ...
12      result[dataIndex - fromInclusive] = centroidIndex;
13    }
14  }
15  return result;
16 }
17 private Map<Double[], List<Double[]>> collectClusters(final int[]
       centroidIndices) {
18  //computation with input parameter centroidIndices
19 }
```

Listing 5: OJXPerf pinpoints the `result` object with many replicas in fj-kmeans.

distance, the program keeps generating the same centroids and put the centroids' indices into an array `result` (line 12 of Listing 5), which is the object with many replicas (the replication factor θ 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 $(1.08 \pm 0.04)\times$ speedup to the entire program.

9 Threats to Validity

The threats reside in validating OJXPerf's optimization guidance. The limited scope of replica detection and the sampling strategy does not reveal the ground truth of object replication. Any reported replication is input and execution specific. Different inputs can result in different profiles, and the effects of optimization on unseen inputs will remain unknown. In our studies, we use real inputs for the applications to ensure the optimization is valid. Finally, our optimization may sometimes break the program readability by inserting conditional checks. Developers need to decide whether to adopt our optimization given their priority on software performance.

10 Conclusions

In this paper, 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. OJXPerf is open source at https://github.com/Xuhpclab/jxperf.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This research was supported by NSF 2050007 and a Google gift.

References

[1] 2013. Obtain object address. https://jrebel.com/rebellabs/dangerous-code-how-to-be-unsafe-with-java-classes-o\bjects-in-memory/4/.

[2] 2018. DaCapo Benchmark Suite 9.12. https://sourceforge.net/projects/dacapobench/files/9.12-bach-MR1/.

[3] 2018. OProfile. http://oprofile.sourceforge.net.

[4] 2020. AlphaGo Zero. https://deepmind.com/blog/article/alphago-zero-starting-scratch.

[5] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. 2004. Prefetch injection based on hardware monitoring and object metadata. SIGPLAN Not. 39, 6, 267–276.

[6] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann.

[7] Matthew Arnold and Peter F. Sweeney. 1999. *Approximating the Calling Context Tree via Sampling.* Technical Report 21789. IBM.

[8] RoaringBitmap authors. 2019. RoaringBitmap. https://github.com/RoaringBitmap/RoaringBitmap.

[9] Mark Bull, Lorna Smith, Martin Westhead, David Henty, and Robert Davey. 2001. Java Grande benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite.

[10] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-guided Proactive Garbage Collection for Locality Optimization. SIGPLAN Not. 41, 6, 332–340.

[11] Oracle Corp. 2017. Oracle Developer Studio Performance Analyzer. https://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/o11-151-perf-analyzer-brief-1405338.pdf.

[12] Oracle Corporation. 2011. jmap Memory Map. https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html.

[13] Oracle Corporation. 2018. All-in-One Java Troubleshooting Tool. https://visualvm.github.io.

[14] Standard Performance Evaluation Corporation. 2008. SPECjvm2008 benchmark suite. https://www.spec.org/jvm2008.

[15] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. Proceedings of the 2015 ACM SIGPLAN Interna- tional Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015), 607–622.

[16] NASA Advanced Supercomputing Division. 2018. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[17] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. 1982. Guidelines for creating a debuggable processor. Proceedings of the first international symposium on Architectural support for programming languages and operating systems (ASPLOS), 100–106.

[18] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 251–265. https://doi.org/10.1145/2908080.2908090

[19] ej-technologies GmbH. 2018. THE AWARD-WINNING ALL-IN-ONE JAVA PROFILER. https://www.ej-technologies.com/products/jprofiler/overview.html.

[20] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. 2004. Using hardware performance monitors to understand the behavior of Java applications. In Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04).

[21] Lu Fang, Liang Dou, and Guoqing Xu. 2015. PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 296–320. https://doi.org/10.4230/LIPIcs.ECOOP.2015.296

[22] Apache Software Foundation. 2017. Apache SAMOA: Scalable Advanced Massive Online Analysis. https://samoa.incubator.apache.org.

[23] Andy Georges, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. 2004. Method-Level Phase Behavior in Java Workloads. Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 270–287.

[24] YourKit GmbH. 2018. The Industry Leader in .NET and Java Profiling. https://www.yourkit.com.

[25] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-Oriented Applications. Proc. of Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 251–269.

[26] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. 2006. Generating object lifetime traces with Merlin. ACM Transactions on Programming Languages and Systems.

[27] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Program Locality. SIGPLAN Not. 39, 10, 69–80.

[28] Skymind Inc. 2019. Deep Learning for Java. https://deeplearning4j.org.

[29] Twitter Inc. 2019. Parquet MR. https://github.com/apache/parquet-mr.

[30] Intel Corporation. 2010. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Number 253669-032.

[31] Nguyen Khanh and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 268–278.

[32] Andreas Kull. 2020. Awesome Java. https://github.com/akullpp/awesome-java.

[33] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. 2006. Online performance auditing: Using hot optimizations without getting burned. In Proc. Conf. on Programming Language Design and Implementation (PLDI 2006), New York, USA, 239–251.

[34] Linux. 2015. Linux Perf Tool. http://www.brendangregg.com/perf.html.

[35] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. OOPSLA '06 Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, 169–190.

[36] Trishul M. Chilimbi and James R. Larus. 1998. Using Generational Garbage Collection to Implement Cache-conscious Data Placement. SIGPLAN Not. 34, 3, 37–48.

[37] Darko Marinov and Robery O'Callahan. 2003. Object Equality Profiling. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 313–325.

[38] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 268–278. https://doi.org/10.1145/2491411.2491416

[39] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 562–571.

[40] Nitsan Wakart. 2016. The Pros and Cons of AsyncGetCallTrace Profilers. http://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html.

[41] Sable Research Group of McGill University. 2019. Soot. https://sable.github.io/soot/.

[42] Oracle Corp. 2016. NetBeans profiler. https://profiler.netbeans.org.

[43] Andrei Pangin. 2018. Async-profiler. https://github.com/jvm-profiling-tools/async-profiler.

[44] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[45] Bill Pugh, Andrey Loskutov, and Keith Lea. 2015. FindBugs. http://findbugs.sourceforge.net.

[46] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. ACM Transactions on Mathematical Software (TOMS), 37–57.

[47] Mark Scott Johnson. 1982. Some requirements for architectural support of software debugging. Proceedings of the first international symposium on Architectural support for programming languages and operating systems (ASPLOS), 140–148.

[48] Ajeet Shankar, Mattew Arnold, and Rastislav Bodik. 2008. JOLT: Lightweight Dynamic Analysis and Removal of Object Churn. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 127–142.

[49] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. 2002. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. SIGPLAN Not. 37, 11, 13–25.

[50] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 370–380.

[51] Pengfei Su, Qingsen Wang, Chabbi Milind, and Xu Liu. 2019. Pinpointing Performance Inefficiencies in Java. The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia.

[52] Nathan R. Tallent. 2010. *Performance analysis for parallel programs from multicore to petascale.* Ph.D. thesis. Department of Computer Science, Rice University.

[53] Qingsen Wang, Xu Liu, and Milind Chabbi. 2019. Featherlight Reuse-Distance Measurement. Proceedings of The 25th IEEE International Symposium on High Performance Computer Architecture, 440–453.

[54] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 47–61. https://doi.org/10.1145/3037697.3037729

[55] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18), 440–453.

[56] Guoqing Xu. 2012. Finding reusable data structures. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1017–1034.

[57] Guoqing Xu. 2013. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-World Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 111–130. https://doi.org/10.1145/2509136.2509512

[58] Guoqing Xu, Matthew Arnold, and Nick Mitchell. 2009. Go with the Flow: Profiling Copies To Find Runtime Bloat. Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), 419–430.

[59] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 419–430. https://doi.org/10.1145/1542476.1542523

[60] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow down Causes of Memory Leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/1993498.1993530

[61] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing. *ACM Trans. Softw. Eng. Methodol.* 23, 3, Article 23 (June 2014), 50 pages. https://doi.org/10.1145/2560047

[62] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-Used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 160–173. https://doi.org/10.1145/1806596.1806616

[63] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-Used Containers to Avoid Bloat. Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)., 160–173.

[64] Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static Detection of Loop-Invariant Data Structures. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 738–763. https://doi.org/10.1007/978-3-642-31057-7_32

[65] Dacong Yan, Guoqing (Harry) Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, David R. Kaeli and Tipp Moseley (Eds.). ACM, 87. https://dl.acm.org/citation.cfm?id=2544151

[66] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving program locality in the GC using hotness. PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, 301–313.