Incorporate Program Analysis into Persistence by Reachability Model

Zhang Yilin Omkar Dhawal Shigeru Chiba

V. Krishna Nandivada Tomoharu Ugawa

The Persistence by Reachability (PBR) model, an effective abstraction for object persistence, instantly persists objects upon their becoming reachable from predefined durable roots. The PBR model provides a promising approach towards realizing persistent memory for programs with large interconnected data. Despite the various implementations proposed in prior studies, a recurring inefficiency is related to the costs arising out of the presence of a large number of expensive write barriers. In this paper, we present a scheme that reduces the cost due to these write-barriers to a significant extent.

The large number of these write-barriers arises from the naive assumption that all objects are considered shared among threads. Under this assumption, while a thread is writing to an object, another thread may make it reachable from the durable roots. Therefore, every write to an object must check (say, by invoking a write-barrier) if the object is made persistent during the write. However, if it can be established that the object being written to can only be accessible from a single thread (a.k.a. thread-local), then the corresponding write-barrier could be elided. We first found, through a runtime reference analysis on various benchmarks, that 2 – 66% of write operations were writing to thread-local objects. To automatically identify and eliminate these redundant write-barriers, we incorporated a multi-stage program analysis/transformation scheme into the PBR model. We perform a static flow-sensitive, context-insensitive points-to and escape analysis using an on the fly call-graph to annotate each write operation, whether the written object is thread-local at that point. Consequently, for those thread-local object writes, the JIT compiler generates code without any write-barriers. We have implemented the optimization in a replication-based PBR implementation. Our evaluation across various benchmarks demonstrated that our optimization reduced most of the redundant barriers, thus enhancing overall performance.

1 Introduction

The Persistence-by-Reachability (PBR) model serves as a feasible object persistence model, automatically guaranteeing the integrity of persistent data [5] [6]). It shifts the burden of specifying each object to be persistent or non-persistent from programmers to the runtime. As a consequence, programmers only annotate some variables as *durable roots*, and the PBR model instantaneously persists objects as they become reachable from those durable roots.

However, prior implementations of the PBR model suffer from significant overhead, even when the program has no persistence requirements. To illustrate this fact, we compared the execution time between CCJava [5], a replication-based implementation of the PBR model, and the original Java virtual machine (JVM) for programs without persistent objects. We used benchmarks from the DaCapo suite [2] for this comparison. As shown in Figure 1, CCJava was on average 2.34 times slower than the original JVM, reaching up to 4.44 times slower for the luindex benchmark (Section 2).

The overhead arises from the inherent requirement of the PBR model: the write barrier must handle persistent objects and non-persistent differently, while a non-persistent object may be made persistent asynchronously. This requires the write barrier to perform a costly memory barrier instruction in its hot path.

To reduce write-barrier overhead, we propose

Yilin Zhang, Shigeru Chiba, Tomoharu Ugawa, Dept. of Information Science and Technology, The University of Tokyo.

Omkar Dhawal, V. Krishna Nandivada, Dept. of CSE, Indian Institute of Technology Madras.

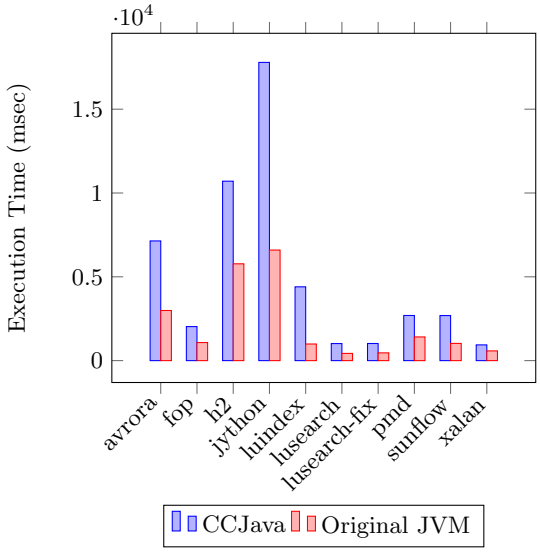

Fig. 1 Elapsed times for unpersistent executions.

to distinguish unnecessary write barriers by static analysis. Prior implementations assume that all objects are candidates for persistence, thereby invoking the write barrier for each write. Consequently, the performance decreases dramatically. However, if the object being written to, or the *target object*, is always thread local at a write operation, the write barrier for the write is unnecessary as a thread local object can never be persistent. We propose to use a static analysis to find such write operations (Section 4) and elide the corresponding write barriers during JIT compilation.

We implemented a JIT compiler pass in the C1 compiler of OpenJDK to eliminate write barriers based on the results of the static analysis (Section 3). Our performance evaluation revealed that 11.7% of the write barriers were eliminated for the luindex benchmark in the DaCapo benchmarks, translating to an 10% speed increase (Section 5).

2 Write barriers in CCJava

This section provides an overview of the role of the write barrier and potential optimization scenarios within CCJava. CCJava achieves object persistence by duplicating the object from DRAM to persistent storage in non-volatile memory (NVM) and consistently updating the two copies in every subsequent write operation. As the original copy is reserved in DRAM, facilitating direct reads, CCJava requires only write barriers (unlike the prior work [6] that needed both read and write barriers).

Algorithm 1: Write Operation

```
1 Function write(object o, field f, value v)
2     o.f ← v;
      /* write barrier start        */
3     store fence;
4     if o has replica then
5         replica.f ← v;
      /* write barrier end          */
```

2.1 Functions of Write Barriers

The write barrier within CCJava fulfills two critical functions:

1. to replicate the object from DRAM to NVM instantly when making the object reachable from a predefined variable,
2. to write to the replica if the target object has a replica.

More precisely, as shown in Algorithm 1, the write operation in CCJava comprises the normal update in DRAM (Line 2) and a write barrier (Lines 3 – 5). First, an expensive memory fence, which prevents memory operations from being reordered, must be placed between the write to the original object and the check for the replica's existence. This ordering is vital for target objects being made persistent asynchronously by another thread, as any write that may happen after the object is made persistent must update the replica as well. Then, Line 4 checks if object o has a replica. If replica exists, the write barrier updates the same field in the replica to the same value.

2.2 Redundant Write-Barrier

The write barrier is superfluous if the target object could be identified as a non-persistent object at that point. Further, in CCJava, an object that is only reachable from local variables of a single thread, or a *thread-local object*, cannot become persistent, as persistent objects must be reachable from the durable roots, which are static fields. Therefore, the write barrier can be eliminated when the target objects are always thread-

```
1  // a global variable
2  AddressBook book;
3  addRecord(String name, String
       address) {
4      record = new Record();
5      // record is thread-local
6      record.name = name;
7      record.address = address;
8      // record escapes
9      book.add(record);
10     record.inBook = true;
11 }
```

Fig. 2 Motivating example.

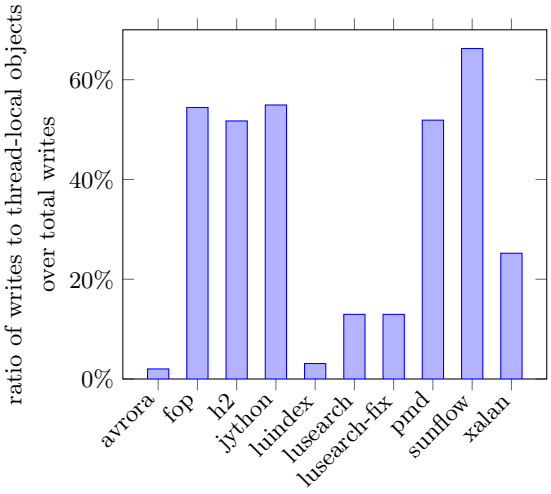

Fig. 3 Ratio of writes to thread-local objects over total writes.

local at the time of writing. Conversely, the write barrier must be maintained if any target object cannot be guaranteed to be thread-local.

Writing to thread-local objects is common a phenomenon. For example, in the code snippet in Figure 2, Lines 6 and 7 initialize the object created on Line 4, bound to `record`. This object is thread-local on these lines, and thus write barriers are not necessary for them. However, at line 9, the `record` object may become non-thread-local because `book.add` may assign its reference to a static variable or a non-local thread object. Consequently, any subsequent writes to fields of the `record` object must be accompanied by a write barrier. Specifically, on Line 10, a write barrier is necessary to ensure that the update to the `record.inBook` field is properly replicated.

2.3 Approximate Amount of Unnecessary Write-Barrier

To approximate how many write barriers could be eliminated, we first developed a naive dynamic escape analysis. We emphasize that the purpose of this analysis is not an optimization, but an investigation of the feasibility of the optimization. Therefore, we do not care about runtime overhead of this analysis.

This analysis tracks objects that have never been pointed to from any object or static variable. Such objects are a subset of thread-local objects, and the writes to them do not need a write barrier. We counted the number of such writes. More precisely, we introduced a new field, `is_referenced`, to the object header, managing it with atomic operations as follows.

- The `is_referenced` flag is set whenever the object is assigned to any field of any object.
- The `is_referenced` flag is checked at every write to any object. If false, the write is counted as not requiring a write barrier.

Note that a write instruction at a single code point may be executed multiple times. Some of its execution may write to thread-local objects, while the others may write to objects that are not thread-local. In this analysis, we counted the number of writes to thread-local objects for such write instructions, while the static analysis-base optimization cannot eliminate barriers for such writes.

Figure 3 shows the fraction of writes to thread-local objects for selected DaCapo benchmarks. We selected these benchmarks because they were previously evaluated in CCJava, providing a meaningful basis for further comparison and optimization. The results revealed that an average of 33.5% of write barriers at runtime was unnecessary, reaching up to 66% for the sunflow benchmark.

3 System Design

3.1 Overview

Our approach to identify and eliminate unnecessary write barriers consists of two components: static analysis and the JIT compiler.

3.2 Static Analysis

The goal of static analysis is to annotate whether each writing bytecode in a class file always writes to thread-local objects or not. In Java, writing bytecodes are those that write to objects, that is, `putfield`, and array writing bytecodes such as `aastore`. This analysis is performed offline.

3.3 JIT Compilation

Our JIT compiler accepts the annotated bytecode and generates the corresponding machine code. If a writing bytecode is annotated as always writing to thread-local objects, the compiler does not generate a write barrier for it.

We extend the C1 compiler within OpenJDK to eliminate the write barrier. The C1 compiler compiles bytecodes into machine code through the high-level representation called HIR and the low-level representation called LIR. When the HIR is compiled to LIR, a write barrier generator function is invoked to compile the writing bytecodes. We develop a new write barrier function that emits the CCJava's write barrier code[†1]. The barrier function chooses to emit the write barrier or a single `move` instruction of x86 without the write barrier by referring to the annotation.

It is important to note that only methods on the hot path are compiled by the JIT compiler, meaning that not all statically analyzed methods undergo compilation. Consequently, if a write is identified as not requiring a write barrier, but is not JIT-compiled, the unnecessary write barrier is still executed in the interpreter. However, our optimization on the JIT compiler suffices, as our preliminary evaluation showed that, for benchmark programs without durable roots, the removal of all write barriers in the interpreter leads to negligible performance improvement.

†1 CCJava [5] provided only an interpreter implemetation

4 Static analysis

In this section, we describe the static analysis component that we have designed to identify the relevant object thread-escape information and propagate the same to the JVM.

4.1 Abstract Objects

In this manuscript, for the ease of exposition, we identify each abstract object allocated at Line x, by the symbol O_x (for example, the variable `record` in Figure 2 points to O_4). Thus, O_x represents all the objects that may allocated at that line, during program execution. In our implementation, to maintain the correspondence between the analysis results computed during static analysis and JVM, we use the byte-code-index (instead of source code line number) to distinguish the abstract objects allocated in each method. Thus, for an object allocated at the byte-code-index (BCI) i is represented by O_i. In addition to these abstract objects, we have additional special abstract objects to include: (i) a single abstract object called E_{obj} to denote thread-escaping objects. (ii) different *dummy* objects representing the objects that may be created within the un-analyzed library methods. An interesting property of these special objects is that dereferencing any field of such objects, returns a similar (E_{obj} or dummy) object.

4.2 Points-to and Escape Analysis

Considering the very high costs [8] involved in performing context-sensitive analysis, we use a context-insensitive escape/points-to analysis. Similarly, to handle cases that an object may be thread-local for a part of the program, before it escapes and hence the fences are only required for the latter part of the lifetime of the object, we perform a flow-sensitive escape/points-to analysis [3] [9]. In the analysis, at each program-point, we maintain points-to information using the maps abstract-stack (ρ) and abstract-heap (σ) as shown in Figure 4. If a variable may point to E_{obj}, then any field dereference via that variable will return E_{obj}.

The static analysis computes a map FenceRequired, such that given a `putfield` or array-write instruction at BCI x in a method m, FenceRequired(m, x) returns true, if the variable dereferenced at BCI x

Vars	= Set of variables.
AObjs	= Set of abstract objects.
ρ	: Vars $\rightarrow P$(AObjs)
σ	: AObjs $\times$ Fields $\rightarrow P$(AObjs)

Fig. 4 Sets and maps used to maintain points-to + escape information at each program location. We use $P(X)$ to denote the power set of X.

of method m may point to a thread-escaping object (E_{obj}). The static-analysis component shares this information with the JVM. For efficiency, we only emit those (m, x) pairs, for which the FenceRequired map returns true.

4.3 Implementation

We now describe some heuristics in our implementation to make our analysis more precise, especially when we cannot analyze certain parts of the code (for example, library calls), or when we cannot track the precise points-to graph (for example, arrays and collection classes).

4.3.1 Modelling arrays and Collection objects

Modelling objects stored in an array is challenging as statically it is difficult to precisely model the array index. A similar issue occurs in collection classes, where different objects are stored inside the objects of collection classes; this issue compounds, with collection classes like HashMaps, where the HashMap objects store objects corresponding to both keys and values.

To address this complexity, we associate a typeless abstract field `summaryF` with each array and collection object; this field may point to any object that may be stored in the array/collection-class-object. Note that since some collection-class objects store different types of objects (for example, HashMap objects described above), the corresponding `summaryF` field may point to objects of different types. To ensure correctness, we use the available type-information when dereferencing from the abstract field. For example, say we create a HashMap object with key type `A` and value type `B`, then we add both keys and values to the same abstract field `summaryF` . If we try to iterate over the keySet of the HashMap, then the iterator must be of type `A`. In such cases, we use the available type information, and only consider that subset of objects pointed-to by the `summaryF` field of type `A`, to improve the precision of our analysis while maintaining soundness.

4.3.2 Handling Library method calls

Analyzing Java library classes during static analysis has one main issues: the available library during static analysis may not match that available during the program execution (as the program may be running on an entirely different system than that used by the static analyser). This difference in implementations restricts us from statically analyzing library methods precisely. To keep the analysis sound, we could mark all the objects reachable from arguments of library method calls and the object returned by library methods as escaping. Though this approach makes our analysis sound, it is very conservative, causing a drastic drop in precision. We employ a heuristic-based approach to reduce the conservativeness of our analysis. We manually analyzed the library method calls and derived general rules to update the points-to information without being overly conservative. Table 1 shows the heuristics used to conservatively analyze library method calls.

Case 1 covers library methods where fields of the objects pointed by arguments can point to objects created inside. We assign a special dummy object which will inherit the same type as the object's field whose points-to-set was found empty. If the dummy object is dereferenced, we create another whose type will be the same as the accessed field. Case 2 deals with methods like `arrayCopy`, where we pass two arrays as arguments, and the contents of one array are copied to the second array. Case 3 handles receiver fields when the receiver is a collection class object. This case addresses methods that deal with accessing the elements of the collection clasess; for example, for the collection, such as `add`, `insert`, `put`, etc. Case 4 handles the case where the receiver is not a collection. Case 5 provides steps to assign objects reachable from the receiver and arguments to the return variable conservatively. If the return variable points to an empty set despite conservative assumptions, it implies that the library method creates and returns a new object. We handle such cases by creating a dummy object that shares the same type as the

return variable.

4.3.3 Callgraph construction

A call graph is a data structure used to resolve the targets for a call site statically. A call site can have more than one possible target statically due to polymorphism. Various techniques, like Class Hierarchy Analysis (CHA) [4], Rapid type Analysis (RTA) [1], are used to construct call graphs with varying levels of precision and efficiency. We construct a call graph on-the-fly using the points-to information available during the Points-to analysis. We iterate over all the objects the receiver may point to and, based on their type, select all possible target methods. This on-the-fly call graph construction technique is very precise and yields better results than CHA and RTA. If the receiver points to E_{obj}, then we fall back to CHA.

5 Evaluation

We conduct two evaluations. Initially, we assess the extent to which the static analysis reduces the number of write barriers. Next, we evaluate the resultant performance improvements.

5.1 Evaluation Settings

We use the luindex benchmark of the DaCapo benchmark suite, version dacapo-9.12-MR1-bach †2. The default benchmark scale configuration is employed.

During the evaluations, no durable root is set and thus no object is made persistent. The configurations used in our experiments are presented in Table 2.

For all Java versions, we consistently applied the following JVM parameters, reflecting the constraints of our current implementation that only supports normal pointers and utilizes the interpreter along with the C1 compiler:

- `-XX:-UseCompressedOops` – Disable the use of compressed ordinary object pointers
- `-XX:-UseCompressedClassPointers` – Disable the use of compressed class pointers
- `-XX:+UseSerialGC` – Enable the use of the Serial Garbage Collector
- `-XX:+TieredCompilation` – Enable the use of tiered compilation
- `-XX:TieredStopAtLevel=1` – Set the tiered compilation to stop at level 1, i.e., C1 compiler

5.2 Write-barrier Reduction

We discovered that 11.7% of writes in JIT-generated code at runtime correspond to write bytecodes which always write to thread-local objects. This translates to that 11.7% write-barriers can be eliminated in the JIT-generated code.

The percentage 11.7% is higher than the estimated 3% for luindex in Section 2.3. The static analysis's better result over our dynamic analysis for the luindex benchmark is expected, as the static analysis considers thread-local objects while dynamic analysis only considers non-referenced objects, which represent just a small subset of thread-local objects.

5.3 Execution Time Improvement

Figure 5 shows the execution time for luindex benchmark of original CCJava with C1 compiler enabled, our proposal and original Java. Eliminating 11.7% write barriers leads to a 10% speed increase.

However, this enhancement is restricted and still falls short of the performance of original Java without additional work. The primary reason is the limited number of eliminated write barriers. Other factors include the time required to handle annotated bytecodes and the remaining unnecessary write barriers in write bytecodes on the slow path. Although these latter two reasons have a minor impact, they still contribute to the overall effect. Further investigation and measurements to address these issues are left for future work.

6 Related Work

Many atomic and semi-automatic persistence models have emerged since the advent of NVM, thanks to the close-to-DRAM performance and persistence characteristics of NVM.

Espresso [10] is a dedicated persistence model for managed runtime like Java, creating a persistent heap on NVM to store the persistent objects. Despite its innovation, it introduces new keywords (such as `pnew` to allocate persistent objects) into Java. This not only alters the language but also requires programmers to specify all persistent objects explicitly, which is error-prone and increases

†2 `https://sourceforge.net/projects/dacapobench/files/9.12-bach-MR1/`

Case	Rule
1. for all arguments	For all reachable objects from the arguments, if field of any object points to an empty set, then add a dummy object with same type as the field's type to the points-to set.
2. argument is array type	If there is any other argument that points to an array object then unify the points-to sets of the `summaryF` fields of both the arrays.
3. receiver is collection object	a) Add all arguments to the `summaryF` field of the receiver. b) If any argument is a collection object or array object then add all the objects pointed by the field `summaryF` of that argument to the abstract field `summaryF` of the receiver.
4. receiver is NOT a collection	Objects reachable from all the arguments should be assigned to the fields of receiver after type-checking.
5. return type is non-void and non-primitive	a) If call statement invokes non-static method then add all reachable objects from receiver's points-to set to the points-to set of return variable after type-checking. b) Add all reachable objects from the argument to the points-to set of return variable after type-checking. c) After step a) and b), if return variable still points to an empty set then create a new dummy object with same type as the return variable.

Table 1 Handling of library functions.

Component	Specification
Memory Modules	Two 256 GB second-generation Intel Optane DC persistent memory
DRAM	96 GB DDR4
CPU Sockets	Two, with an Intel Xeon Gold 6354 processor (18 cores per socket)
Operating System	Linux 5.1.0 Ubuntu 20.04.3 LTS
Compiler	GCC version 9.3.0 for HotSpot VM
HotSpot VM Build	"Server" configuration

Table 2 Server configurations used in our experiments.

the complexity of development.

AutoPersist [6] marked a significant step by introducing PBR model into Java. Unlike earlier approaches, programmers do not have to designate each object's memory type, i.e., NVM or DRAM, but only identify the durable roots. The runtime system moves an object from DRAM to NVM if it becomes reachable from these roots, thereby guaranteeing the integrity of persistent data. The drawback to this approach lies in the substantial runtime overhead, as discussed in Section 2.1. Write operations must be meticulously designed to ensure consistency and thread safety, while read operations necessitate persistence checks due to the forwarding pointer left after an object has been moved.

CCJava [5], on which our work is based, is a replication-based implementation of PBR model. In contrast of *AutoPersist*, it copies the persistent objects from DRAM to NVM and maintains both, allowing for direct reading in DRAM, leading to

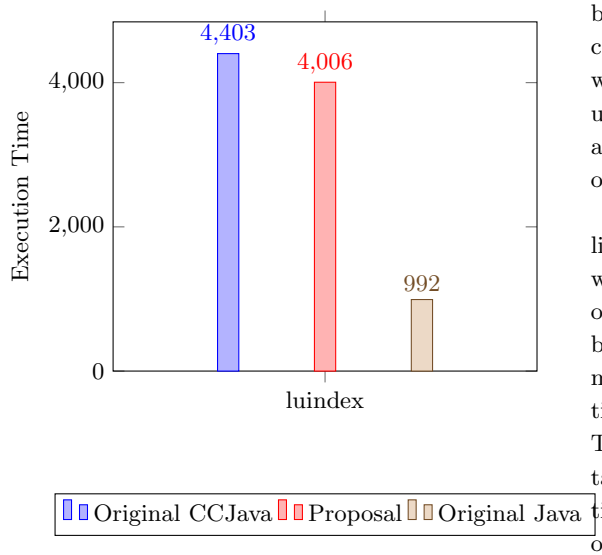

Fig. 5 Execution time improvement evaluation

elimination of the persistence check in read operations.

quickCheck [7] previously identified the overhead associated with persistence checks in the PBR model. Their method collected persistence check profile information (tracking if the target object was persistent or not at each check) to generate code with optimizations, such as aligning likely branches with the main execution path. The significant difference between quickCheck and our work is that quickCheck does not eliminate write barrier, which we do, but just rearranges the generated code.

Escape analysis has long been instrumental in determining whether synchronization operations on an object can be eliminated or not [3]. To the best of our knowledge, our work is the first to apply escape analysis to optimize write barriers in the PBR model.

7 Conclusion

In this work, we uncovered the write-barrier overhead in prior implementations of the PBR model, and evaluated it particularly in CCJava.

Our solution, rooted in the annotation of bytecodes and optimization of JIT-generated code, has yielded promising improvements.

A primary area of interest lies in also integrating dynamic analysis into the PBR model, along with current static analysis, to eliminate more write-barriers. Though we have developed a dynamic escape analysis capable of eliminating write-barriers when objects are not referenced, the current cost is unrealistic for production use. Future efforts may aim to optimize this process, making it a practical option for improving performance.

Another potential avenue for research is to establish an upper bound for the number of unnecessary write barriers as a performance metric. Currently, our evaluation focuses on the reduction of write barriers in runtime and the consequent improvement in performance. However, assessing the effectiveness of our static analysis remains a challenge. This goal might be accomplished by monitoring the target objects of each write bytecode during runtime, thereby providing a clearer understanding of our static analysis's accuracy and potential.

In conclusion, this research contributes both a practical tool to improve PBR model performance and a theoretical understanding to guide future investigations. The observed correlation between write barrier elimination and execution speed emphasizes the value of pursuing this line of inquiry, pointing toward exciting possibilities for advancing both technology and methodology in the field.

Acknowledgements This work was supported by JSPS KAKENHI Grant Number JP22H03566.

References

[1] Bacon, D. F. and Sweeney, P. F.: Fast Static Analysis of C++ Virtual Function Calls, *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, New York, NY, USA, Association for Computing Machinery, 1996, pp. 324–341.

[2] Blackburn, S. M. et al.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, New York, NY, USA, Association for Computing Machinery, 2006, pp. 169–190.

[3] Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S.: Escape Analysis for Java, *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, New York, NY, USA, Association for Computing Machinery, 1999, pp. 1–19.

[4] Dean, J., Grove, D., and Chambers, C.: Optimization of Object-Oriented Programs Using Static

Class Hierarchy Analysis, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, Tokoro, M. and Pareschi, R.(eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, 1995, pp. 77–101.

[5] Matsumoto, K., Ugawa, T., and Iwasaki, H.: Replication-Based Object Persistence by Reachability, *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2022, New York, NY, USA, Association for Computing Machinery, 2022, pp. 43–56.

[6] Shull, T., Huang, J., and Torrellas, J.: AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, New York, NY, USA, Association for Computing Machinery, 2019, pp. 316–332.

[7] Shull, T., Huang, J., and Torrellas, J.: QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks, *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, New York, NY, USA, Association for Computing Machinery, 2019, pp. 137–151.

[8] Smaragdakis, Y., Bravenboer, M., and Lhoták, O.: Pick Your Contexts Well: Understanding Object-Sensitivity, *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, New York, NY, USA, Association for Computing Machinery, 2011, pp. 17–30.

[9] Whaley, J. and Rinard, M.: Compositional Pointer and Escape Analysis for Java Programs, *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, New York, NY, USA, Association for Computing Machinery, 1999, pp. 187–206.

[10] Wu, M., Zhao, Z., Li, H., Li, H., Chen, H., Zang, B., and Guan, H.: Espresso: Brewing Java For More Non-Volatility with Non-Volatile Memory, *SIGPLAN Not.*, Vol. 53, No. 2(2018), pp. 70–83.