Escaping the Clone Zone: Java Runtime-Managed Snapshots Current and Future Work

Matthew C. Davis, Mark Hills
East Carolina University, Greenville, NC, USA
davismat16@students.ecu.edu, mhills@cs.ecu.edu

Abstract—**Mutable objects shared across modules may lead to unexpected results as changes to the object in one module are visible to other modules sharing the object. It is common practice in Java to defensively create a private copy of the input object state via cloning, serialization, copy constructor, or external library. No universal approach exists and each common solution has limitations or problems. The subject of this paper is a novel runtime-managed approach to declaratively unshare input object state. This approach is evaluated using an experimental OpenJDK 10. The paper summarizes the motivation for the work, describes the experimental implementation, discusses ongoing empirical work regarding desirable properties of the snapshot function, and describes plans for future research.**

I. Background

An immutable object is characterized by the inability for its abstract state to change [1, p.1816] after instantiation. An example of an immutable type in Java is the String class [2, p.21]. If clients request a mutated (changed) state, the String object instance constructs and returns a new String object instance with the desired abstract state. The abstract state of the original String object instance remains unmodified.

Sharing immutable objects is safe due to their unchanging abstract state characteristic: another module sharing the object cannot unexpectedly change its state [2, p.116-117]. Immutable objects cannot be corrupted, interfered with, or observed while in an inconsistent state by other threads [3]. These properties make immutable objects useful, fast, and safe to share across threads in concurrent applications.

Software engineers rarely green-field an entire object landscape. In the author's experience, system inputs and outputs are often composed of existing types over which the software engineer or architect lacks specification authority. In these situations, shared mutable objects may not be universally avoidable, and defensive steps must be adopted to prevent unexpected modification to shared mutable object state.

Java 10 lacks a native and universal functionality to unshare mutable object state with abstract and consistent semantics. Clone is non-universal with under-specified semantics [4]–[6], Serialize is non-universal and may throw unexpected runtime errors [4], [5], and copy constructors are non-universal and type-dependent; consequently, using shared mutable objects in standard Java may require knowledge of an object's actual type and implementation to understand the options available to protect shared object state from unexpected modification.

Further, if two otherwise-unrelated types implement a particular interface as well as Cloneable, there is no guarantee both underlying types implement the same clone() semantics when the server method interacts with objects uniformly by their apparent interface type. This problem similarly applies to parameterized types and methods acting as a server for disparate input types with unknown clone() semantics.

II. Motivation

The motivation for this research is prompted by the reality that existing options are not universal, have under-specified behavior, or require external libraries [4]–[6]. For two decades, software engineers have contended with these problems in Java. Similar to past efforts to improve Java – notably Pizza's parametric abstraction enhancements [7] – it is apparent Java's present limitations are not necessarily a prediction of its future capabilities. Offering plausible alternative paths forward is the motivation for this research.

III. Previous Work

The previous work [8] proposed and evaluated a new `__snap__` keyword for the Java Language Specification [9] as a novel, intent-based declaration requiring the Java runtime to unshare the state of the actual method parameter. The abstract behavior of a formal method parameter declared with this modifier may be intuitively understood as a guarantee the object referenced in the actual parameter possesses an abstract state that is, at method execution start, non-shared. The method may subsequently share the object with collaborating methods and objects during its execution – but at the time method execution begins, the actual input object state is non-shared.

Two evaluations were conducted. The first evaluated a transformation approach that was found to be unsuitable due to non-universality and under-specified semantics of the underlying methods available within standard Java [8, § 5.3]. The same issues exist for annotation-based approaches that similarly rely on existing Java methods. A second, successful, evaluation modified OpenJDK 10's javac compiler to emit a new 0xcb asnap opcode for `__snap__`-decorated method parameters. At method execution, the 0xcb opcode instructs the modified OpenJDK 10 HotSpot JVM to perform a memory snapshot of the object at the top of the operand stack and replaces the reference of the shared input object with a reference to the unshared input object copy. This second evaluation was a successful, minimal, experimental implementation for evaluation purposes.

In Figure 1, the method designer intends to receive as input an unshared view of objInput's abstract state. Of course, the method designer may choose to share this view with other collaborating objects and methods by explicitly passing the object outside the method as a reference or method parameter. But in the initial case, the addAll() method alone has visibility to the point-in-time state of objInput at the time the actual parameter is loaded at the start of method execution. To be clear, the intent is not to make mutable objects immutable, but rather to unshare mutable state within a specific method context as required by the method designer.

Steady-state benchmarking [8, § 7.3.3] illustrated the poor performance of serialization as shown in Figure 2. To improve plot scale, Figure 3 omits Serialization. This second plot shows Copy Constructor's penalty at the mean relative to Snap, which shows a penalty at the mean relative to clone(). HotSpot's Clone() is heavily optimized by many engineers across years of development. Snap is not optimized; however, it benchmarks as more performant than copy constructor and serialization.

This approach was shown to be performant relative to the extant alternatives and frees the method designer from the tricky task of predicting the underlying actual type of the object for the purposes of determining how best to copy it via clone, serialize, copy constructor, etc.

IV. Current Work

Frequency of Practice: As of this writing, the relative frequency of the existing methods to unshare mutable object state does not appear to be well-understood. Preliminary work is underway to analyze publicly-available Java repositories such as GitHub to assess the relative frequency of usage. Source code analysis using Rascal MPL [10] is expected to be preferable to bytecode analysis due to (a) the extant functionality within Rascal for Java analysis, (b) the fewer steps needed to draw conclusions on programmer practice when analyzing source code relative to bytecode, and (c) the sufficient quantity of Java code publicly available. Identifying the use of copy constructors and common third party copy libraries will be undertaken.

Depth guarantee: The currently-implemented abstraction was intended for research and is limited to snapshotting two layers of the object graph, similar to some implementations of clone(). An empirical study to determine the distribution of typical object graph depths is in the planning stages. Snapshotting the entire graph presently appears to be the optimal approach as it maximally unshares the object, but data may refute or confirm this view. It is observed solutions such as serialize, GSON, and cloning libraries choose to copy the entire graph [4], [11], [12]. A further option is to let the method designer decide the depth, but one must question how the method designer would intelligently select the "correct" depth as the choice requires knowledge of type-specific behavior similar to the conundrum present today in standard Java.

V. Future Work

Simplify Snapshot Load: Evaluate the tradeoffs of converting the aload/asnap/astore/aload pattern to one opcode.

Escape Analysis: Evaluate omitting snapshots when the input object is fully captured.

JIT Compiler Support: The bytecode interpreter is adapted. For optimal performance, c1 and c2 JIT support is also needed.

Platform Independence: Avoiding one native x64 routine would result in platform independence.

Evaluate Predictable Behavior: It would be interesting to determine via user study whether software engineers find the `__snap__` approach more predictable than extant approaches.

Type Exception: Excluding immutable, singleton, and enum types from the snapshot process is future work.

```java
// Snap/unshare objInput state prior to iterating
public void addAll(__snap__ ArrayList<String> objInput) {
    for(String str : objInput) {
        this.add(str);
    }
}
```

Fig. 1. __snap__ example

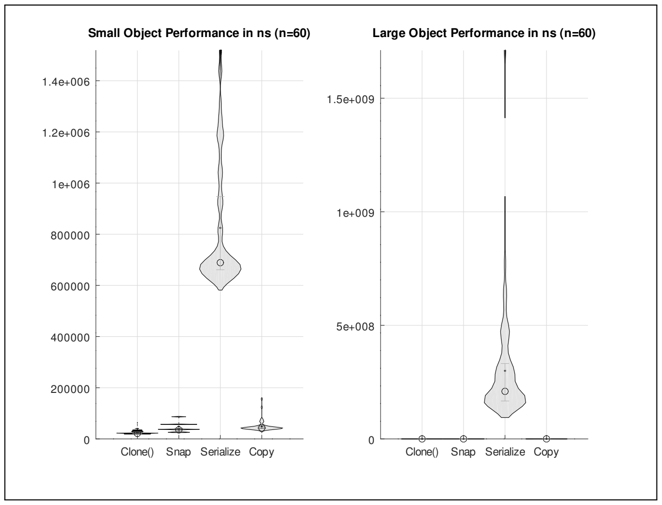

Fig. 2. Benchmarking Results - Violin Plot - w/ Serialization

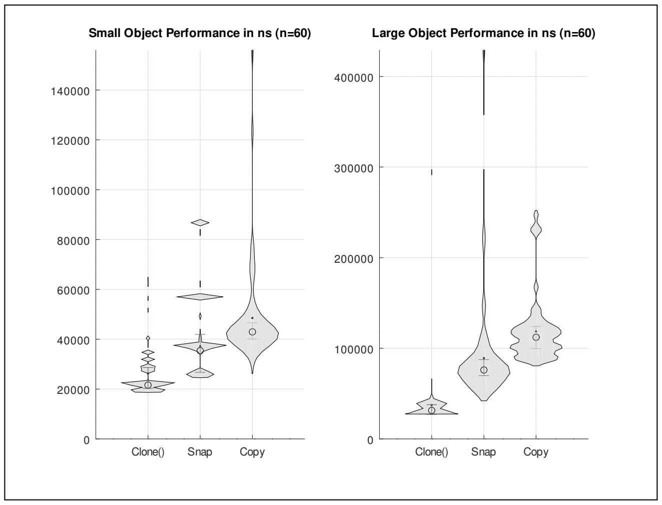

Fig. 3. Benchmarking Results - Violin Plot - w/o Serialization

Bytecode Verification: Implementation of bytecode verification in HotSpot for 0xcb is future work.

Garbage Collection: The minimal implementation can fail during garbage collection and its correction is future work.

Long-term work includes: adapting remaining OpenJDK tools, evaluating differential snapshots and different block sizes, omitting snapshots for immutable objects, reasoning about object state over time, and further exploration of identity relationships among snapshots of the same object.

References

[1] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994. [Online]. Available: http://doi.acm.org/10.1145/197320.197383

[2] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[3] "Immutable Objects (The Java^{TM}Tutorials >> Essential Classes > Concurrency)," Accessed: 2018-10-21. [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

[4] "Java SE 10 & JDK 10 API Specification," 2018, Accessed: 2018-09-22. [Online]. Available: https://docs.oracle.com/javase/10/docs/api/overview-summary.html

[5] V. Ruzicka, "Java Cloning Problems," 2017, Accessed: 2018-09-22. [Online]. Available: https://www.vojtechruzicka.com/java-cloning-problems/

[6] J. Bloch, *Effective Java*, 3rd ed. Addison-Wesley Professional, 2017.

[7] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1997, pp. 146–159.

[8] M. C. Davis, "Applying mutable object snapshots to a high-level object-oriented language," 2018. [Online]. Available: http://hdl.handle.net/10342/7032

[9] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith, "The Java Language Specification: Java SE 10 Edition," 2018, Accessed: 2018-09-16. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se10/html/index.html

[10] P. Klint, T. van der Storm, and J. J. Vinju, "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation," in *Proceedings of SCAM 2009*. IEEE, 2009, pp. 168–177.

[11] "google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back," 2018, Accessed: 2018-10-02. [Online]. Available: https://github.com/google/gson

[12] K. Kougios and et. al., "kostaskougios/cloning: deep clone java objects," 2018, Accessed: 2018-10-02. [Online]. Available: https://github.com/kostaskougios/cloning