APPLYING MUTABLE OBJECT SNAPSHOTS TO A HIGH LEVEL OBJECT-ORIENTED LANGUAGE

by

Matthew C. Davis

November, 2018

Director of Thesis: Mark Hills, PhD

Major Department: Computer Science

Software Engineers are familiar with mutable and immutable object state. 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. When provided a mutable object as input in Java, it is common practice to defensively create a new private copy of the object bearing the same state via cloning, serializing/de-serializing, specialized object constructor, or third-party library. No universal approach exists for all scenarios and each common solution has well-known problems.

This research explores the applicability of concepts within the Computer Engineering storage field related to snapshots. This exploration results in a simplified method of memory snapshotting implemented within OpenJDK 10. A novel runtime-managed method is proposed for declaring intent for object state to be unshared within the method signature. Preliminary experiments evaluate the attributes of this approach.

A path for future research is proposed, including differential snapshots, alternative block sizes, improving performance, and exploring a tree of snapshots as a foundation to reason about changes to object state over time.

APPLYING MUTABLE OBJECT SNAPSHOTS TO A HIGH LEVEL OBJECT-ORIENTED LANGUAGE

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

Matthew C. Davis

November, 2018

APPLYING MUTABLE OBJECT SNAPSHOTS TO A HIGH LEVEL OBJECT-ORIENTED LANGUAGE

by

Matthew C. Davis

APPROVED BY:

DIRECTOR OF THESIS:

Mark Hills, PhD

COMMITTEE MEMBER:

M.N.H. Tabrizi, PhD

COMMITTEE MEMBER:

Sergiy Vilkomir, PhD

CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:

Venkat Gudivada, PhD

DEAN OF THE GRADUATE SCHOOL:

Paul J. Gemperline, PhD

DEDICATION

To my wife, Tara, who always took care of everything.

To my three boys, how quickly you grew.

I missed all of you so much.

(Luke, I owe you a game of soccer)

ACKNOWLEDGEMENTS

This thesis would not be possible without the support of Dr. Mark Hills, who always believed in me and patiently guided me through the research process while allowing this thesis to take on its own life and be my own work.

Much appreciation is due to Dr. Nasseh Tabrizi and to the Computer Science department for providing the opportunity to pursue my research interests in my home state and for providing the environment conducive to academic research.

Acknowledgement is given to Shurtape Technologies and Sunil Tandon for support of this research and providing flexibility at the key moments when nights after work and weekends alone were insufficient.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Mutable and Immutable State

This chapter provides an overview of mutable and immutable state object types as they relate to this research. This chapter then describes why mutable types cannot be completely avoided. Defensive methods presently employed by Java practitioners to protect mutable state are summarized. The research motivation is stated, the hypothesis is proposed, a prediction is made, and a paper structure is described to experiment and collect observations to test the hypothesis.

1.1 Introduction

Software Engineers are familiar with mutable and immutable object state and take steps to ensure shared objects with mutable states do not subsequently change unexpectedly after a mutable object is passed by reference to a software module.

Common solutions to freeze the abstract state of a mutable object in Java include cloning, serializing/de-serializing, or using a specialized object constructor. All of these approaches have known problems and no universal approach exists for all objects within Java. While using only immutable types in theory may be a solution to these concerns, the software engineering team may not control the specification of all objects the system under development must use, such as the standard class library.

1.2 Mutable State

A mutable object is characterized by the ability for its state to change [1, p.1816] after its instantiation and construction. A specific example of a mutable object in Java is an instance of the ArrayList class. During its lifetime, its abstract state changes as clients add and remove objects from the list using mutator methods such as add() and clear(). This ability of an object instance to change state is the defining characteristic of mutable objects.

1.2.1 Advantages

Mutable objects naturally model real-life objects, which are expected to change over time. For objects with state that change frequently, specifying mutability may convey a performance advantage over an immutable alternative as new objects are not created for each new abstract state transition [2, p.116-117].

1.2.2 Disadvantages

Sharing of mutable objects is less safe than sharing of immutable objects: the changes to the mutable object made by one part of the system are also visible – perhaps unexpectedly – to the other parts of the system sharing the object [2, p.116-117]. Consequently, care must be taken to either expect these changes or to create an unshared copy of the object with the same abstract state using the methods outlined later in this section. These steps may require additional testing and complexity within the modules using the mutable objects to address these situations.

1.3 Immutable State

An immutable object is characterized by the inability for its state to change [1, p.1816] after its instantiation. An example of an immutable object in Java is an instance of the String class [2, p.21]. During an instance's lifetime, its abstract state never changes. If clients request a mutated (changed) state, such as a trim or substring via mutator methods, the String object instance constructs and returns a new String object instance with the desired abstract state. The state of the original String object instance remains unmodified.

This inability of an object instance to change state is the defining characteristic of immutable objects. One advanced technique for implementing immutable objects is Persistent Data Structures, which Michael J. Steindorfer [3, § 1.4] defines as:

> [...] an immutable data structure that is a Directed Acyclic Graph (DAG) and consists of separate immutable segments. A persistent data structure is incrementally constructed by linking new immutable data segments to an existing DAG of immutable data structure segments.

The basic example described by Steindorfer [3, § 1.4] is the cons-list originating from LISP [4]. This type consists of a base case empty list, $\emptyset$, and a list cell type consisting of head/this and tail/rest. Adding one atom, A, to $\emptyset$ results in the creation of a new a list/cell, a, composed of A as head/this and $\emptyset$ as tail/rest. Adding a second atom, B, to list cell a results in the creation of a new list/cell, b, composed of B as the head and a as the tail. In this manner, a's representation only contains A and a pointer to b. b's representation only contains B and a pointer to $\emptyset$. This avoids duplication of atoms comprising the list instances and provides incremental, almost version-controlled, history of changes to the object state leading from inception as $\emptyset$ up to and including current state, b.

From the client's perspective, the immutable character of the persistent data structure is observable, but in fact the distinguishing characteristic is the internal representation hidden from the client. The representation of a persistent data structure is characterized by its composition of a new immutable DAG from the immutable predecessor DAG plus an immutable delta. This new immutable DAG represents the successor object instance state.

1.3.1 Advantages

Sharing of 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]. Due to this inherent characteristic, safe shared use of immutable type instances is simpler to implement and test than with mutable objects.

As described within the Java Tutorials [5], immutable objects cannot be corrupted, interfered with, or observed while in an inconsistent state by other threads. These properties make immutable objects useful, fast, and safe to share across threads in concurrent applications.

1.3.2 Disadvantages

Immutable objects are less suited to model objects in real life, which are expected to change over time. For objects with frequently-evolving abstract state, the object creation and destruction required for each evolution of an immutable object may incur a performance penalty [2, p.116-117].

1.4 Discussion

Mutability is a design decision and is part of a type's specification [2, p.116-117]. A modern system development team typically lacks specification authority for all objects it will use. If it did have that authority, it is likely not practical to avoid re-using existing, well-tested functionality. Consequently, it is not always practical to avoid shared use of mutable objects. This is evident by the number of cloned mutable objects encountered in many Java projects[1].

1.5 Preventing Unexpected Mutability: Current Practices

When a shared mutable object is passed by reference into a software module and outside changes to the shared object's abstract state cannot be accepted, it is necessary to adopt one of the defensive methods below to ensure the abstract state of the shared object does not change unexpectedly.

1.5.1 Use Immutable Objects

Using immutable objects is a simple solution when the software module input may be restricted to immutable objects. When the project team lacks specification authority to the types the module accepts – i.e., standard Java Collections – this is not a viable solution.

[1]Formally evaluating this frequency is suggested as future work in Chapter 8

1.5.2 Object.clone()

Object's clone() method is intended to be overridden to produce a copy of an instance whose type implements the Cloneable marker interface. The exact nature and depth of the copy is under-specified and thus its behavior is expected to vary based on the type [6, Object.clone()].

Further, it is not guaranteed a type marked as Cloneable has overridden its supertype's clone() method [7] even if current best practice per Bloch is to do so and immediately call the supertype's clone() method at the top of the current type's implementation [8]. The informal caution with clone() is, "you get what you get." For these reasons, clone()'s behavior across types is inconsistent and its implementation is not universal to all types.

1.5.3 Serialize/De-serialize

Java provides a predefined mechanism for copying an object graph to a data stream called Serialization. This reliably outputs a deep copy but requires all types within the graph implement the Serializable marker interface. The reverse of this process is De-serialization, which accepts a data stream containing a serialized object graph and re-constructs the corresponding object instances [6, java.io.Serializable].

Serialize/de-serialize is an alternative to Object.clone() and provides a predictable depth of copy. As described in the API specification, if an object is not marked as Serializable by the designer, the object may not be serialized [6].

Consequently, serialization is more predictable than clone(), but similarly non-universal and more expensive from a runtime perspective [7] partially due to the round trip from object graph to bit stream and back again.

1.5.4 Copy Constructor

If specification authority is possessed for a type, a copy constructor may be specified that accepts a type instance as input. The copy constructor creates a new object as a copy of the old [7]. Similar to previous methods, this is non-universal.

1.5.5 Reflection

If a type must be copied, the project lacks specification authority for the type, and the type does not support the methods above, Java's Reflection API [6, java.lang.reflect] provides a last-ditch capability to inspect the object's state and produce a copy [7].

Using Reflection may be a brittle approach improperly reliant upon point-in-time assumptions about the underlying type implementation, which may evolve. Subsequent changes to the type implementation or provision of an alternate implementation may break these assumptions.

For this reason, external packages are available to automate dynamically inspecting the class and creating a copy of the object graph [9, 10] [11, cloneBean()].

1.6 Motivation

The motivation for this research is the reality that the preceding options are not universal, have under-specified behavior, or require external libraries. For almost two decades software engineers have contended with these problems in Java with no further solutions available.

Similar to past efforts to improve Java – notably Pizza's parametric abstraction enhancements [12] – 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.

1.7 Hypothesis

The hypothesis of the research is that a universal Java mechanism is feasible that allows a method designer to universally specify when a method's actual parameter should be guaranteed to possess non-shared state with predictable semantics.

1.8 Prediction

To test the hypothesis, this paper predicts that a mechanism may be implemented in Java that is: universal (applies to all object instances), predictable (behavior is consistent across types), self-contained (does not require external libraries), and has reasonable performance (relative to existing methods).

1.9 Organization of this Paper

Chapters 2 and **3** provide further background context, **Chapter 4** specifies an abstract solution, **Chapters 5** and **6** outline alternative implementation approaches, and **Chapter 7** evaluates the stronger alternative against the prediction. Future work is outlined in **Chapter 8**. Conclusions are drawn in **Chapter 9**.

1.10 Summary

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 lacks specification authority. In these situations, shared mutable objects cannot be universally avoided, and defensive steps must be adopted to prevent unexpected modification to shared mutable object state. The existing defense options available in Java 10 are inconsistent, non-universal, or both.

Chapter 2

Java Platform

This chapter provides an overview of the Java programming language and its key supporting tools, which together represent a development platform that has remained influential and in wide practical use for over two decades. As the Java platform is the basis for this research, a basic understanding of its components, design, and operation benefits the reader.

2.1 Java Language

The primary characteristics of Java are described in the Java Language Specification [13] and summarized below with emphasis on details relevant to this paper.

Object Oriented. Java programs are organized around objects. In Java, objects are instances of classes. Classes define common structure and behavior of their object instances. The internal behavior and structure of an object is hidden to clients: instead, clients are provided a contract of abstract behavior by which they may interact with the object.

High-level. The underlying machine details and representations are not available to the program as this would defeat Java's goal of maximum portability and reduce the safeness of the language.

Concurrent. The language specification provides for concurrency of programs such that methods and data elements may be guarded/locked to ensure only one thread may use them at a time. Given that execution environments differ, the Java language specifies the allowed behaviors of concurrent programs such that consistent behavior may be obtained across platforms.

Strong and Static typing. Each Java object and object reference is declared in the source code to be of a specific type. At compile time the type declaration is checked against the reference usage to ensure the declared type is compatible with its usage. The goal is to increase program reliability by minimizing runtime errors and unexpected behavior. Some potentially-unsafe operations such as unchecked downcasts are allowed by the compiler. These generate a runtime error if the operation is found to be unsafe at runtime. Hence, Java's basic type safety is still robust. As of Java 10, type inference is expanded to local variables [14]. The type is still inferred at compile time, which maintains Java's strong and static typing guarantees.

Simplified syntax similar to C++. Java uses C++-ish syntax in an attempt to appear familiar to software engineers – not for source-level compatibility with C++. Many of the complexities of C++ are omitted such as manual memory management.

An assumption of **Automatic memory management**. The language provides constructs for objects to clean up after themselves upon garbage collection once they are no longer being used but is largely neutral about the implementation mechanism of garbage collection, which has varied widely in form and method over time.

Portable. The Java Language Specification[13] and the Java Virtual Machine Specification[15] are purposefully designed to eliminate Java-program-observable dependencies or assumptions about the execution hardware. The language specification provides a platform-neutral set of described behaviors and representations such that the the operation of a compiled Java program executed on a compliant JVM appears

to be functionally equivalent regardless of the underlying hardware. This portability of execution is one of the more notable aspects of Java and the mechanism of this characteristic is described in subsequent sections below.

Other Java language features exist, such as exceptions and closures, but they will not be discussed in this paper.

2.1.1 Object Model

Similar to typical class-based and object-oriented languages, Java object templates are called classes and are defined in source code. These classes include code (behavior), data (state), and contracts (method signatures, object type, interfaces). In Java, all code and data are declared inside a class.

Each Java class must occupy a position in the class hierarchy and inherits from one "parent" class. This parent class' implemented behavior and state are implicitly bestowed upon the new class. By default, classes directly inherit from Java's fundamental type, Object, but the actual parent class is decided by the software engineer. Java's one-parent approach avoids the well-known difficulties[1] and hard-to-remember rules required for languages such as C++ where a class is allowed multiple parents.

In addition to the object type hierarchy, classes may implement interfaces, which are a set of method signatures that represent a contract of behavior. If a class implements an interface, it is contractually obligated to implement all its required methods and this obligation is checked at compile time. It is important to note the contract enforcement is focused on the implemented method signatures and not on the logic within the methods, which form part of the required behavior. This latter check is not provided by Java and is left in the responsibility of the software engineer.

[1] To some extent, these problems have been introduced in the form of default interface methods

As of Java 8, interfaces may include a default method implementation. The goal of default methods is to allow interfaces to evolve without breaking classes that previously implemented the un-evolved interface. Previously, once objects implemented an interface, the interface could effectively not be modified without also modifying all implementing classes [16].

A side-effect of this enhancement is default methods may conflict if, for instance, two interfaces are implemented by one class and both interfaces contain a default implementation for the same method signature. In this case, despite the design goal, implementing classes may still require modification in reaction to interface evolution.

A class may be used to create many different objects, each with its own state (data) but all sharing a common set of behaviors (code) and contracts (method signatures, object type, interfaces). Creating an object from a class is called instantiation as it creates a distinct instance of the object class in memory. This memory holds the state (data) of the object instance. Data that is static, or held in common to all instances of a class, is stored elsewhere in memory. Classes may be declared as abstract, which means they may not be instantiated; rather, these classes only exist in the type hierarchy to bestow state and behavior to child classes and to provide a common type to which all descendants are co-variant.

2.1.2 Type System

Java's type system may be succinctly described as strong, static, safe, and covariant with support for parameterized types.

Strong and Static

As a statically-typed language, the type of every Java expression or variable is determined at compile time. As a strongly-typed language, each variable has a known

type and its contents and operations must be compatible with that type. By identifying the types during compilation, the allowable operations are known and checked to detect errors before the program is executed by a user [13, § 4].

Safe

The preceding attributes provide a greater level of safety as type checking is performed to ensure all operations on an expression or variable are valid [13, § 4]. For instance, multiplying two strings is not a valid operation.

Covariant

While it is not the purpose of this paper to discuss substitutability at length, a short discussion will suffice.

Given an object hierarchy of Object with subtype Vehicle with subtypes Car and Bus (see Figure 2.1), covariance intuitively is the idea that an object of type Car may also be typed as its ancestor types – Vehicle and Object – in that order of precedence.

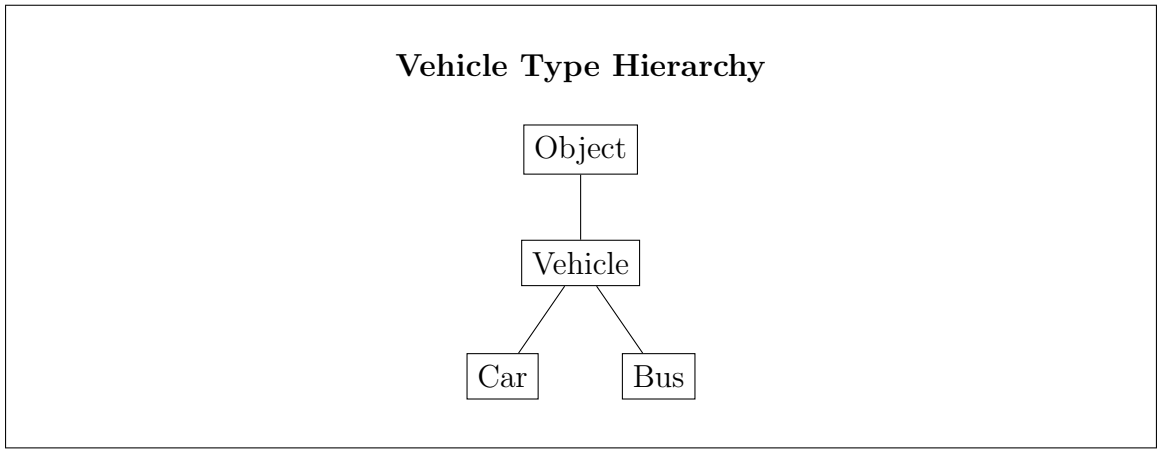

Figure 2.1: Vehicle Type Hierarchy

That is, an object of type Car is a Vehicle, which is an Object. The implication is that wherever Vehicle is used, Car or Bus may be substituted given that a Car is also

a Vehicle and a Bus is also a Vehicle. That is the intuitive description of co-variance.

The implication, of course, is that subtypes must syntactically and behaviorally adhere to the contract of the parent in order for covariance within a type hierarchy to produce reasonable outcomes. Java enforces the syntactic aspects of this contract but the behavioral aspect is within the domain of the software engineer to enforce. This latter behavioral requirement was formally described by Liskov and Wing in 1994 [1] and is generally referred to as the Liskov Substitution Principle.

Parameterized Types

Also called parametric polymorphism – or very loosely, generics – parameterized types ensure type safety but allow a class or method or type to use more than one type. The classic example is a List of objects. In Java 1-4, Lists and Collections were comprised of members of type Object. This is now called a "raw type" [13, § 4.8].

This scheme was troublesome to the practitioner – and one the author personally found frustrating at the time – as any Object was legally insertable into the collection and a downcast cast to the desired type was required upon retrieving the object from the collection. This operation may result in a runtime error when incompatible types were previously and erroneously inserted into the collection. See Figure 2.2 for a simplified example of raw type usage with no compile-time errors that will result in an obvious runtime error due to the unchecked cast.

Starting in Java 5, types may accept additional type arguments as parameters, which are checked at compile time to ensure type safety. Unlike some other languages, once the Java compiler has validated type safety, it erases the parameterized type information and the type is output as a raw type in the Java bytecode in a process called Type Erasure [13, § 4.6].

```
1  // Raw type
2  ArrayList myList = new ArrayList();
3
4  myList.Add(new String("4"));
5  myList.Add(new Integer(5));  // oops! (but valid at compile time)
6  myList.Add(new String("6"));
7
8  String myString = "";
9
10 while(myList.size()) {
11     myString = myString + ((String) myList.get(0)) //runtime error!
12 }
```

Figure 2.2: Java 1 raw type example

Type Erasure is important to understand as not all type information is available at runtime. See Figure 2.3 for an example of type-safe parameterized type usage in contrast to Figure 2.2. The software engineer may include parameters in classes, methods, and interfaces to provide reusable functionality for multiple types while remaining type-safe.

```
1  // Parameterized type
2  ArrayList myList<String> = new ArrayList<>();
3
4  myList.Add(new String("4"));
5  myList.Add(new Integer(5));  // Compile-time error
6  myList.Add(new String("6"));
7
8  String myString = "";
9
10 while(myList.size()) {
11     myString = myString + myList.get(0)
12 }
```

Figure 2.3: Java 5+ parameterized type example

Auto-boxing and Unboxing

For performance, traditional stack-based primitive types are provided by Java such as integers, floats, and arrays as well as flexible but slower heap-based object-oriented analogs of those primitives: Integer, Float, and various collection types.

To simplify programming in Java 5 and later, the Java compiler uses Java's strong and static typing attributes to automatically convert primitives to reference types and vice-versa according to rules defined in the Java Language Specification [13, § 5.1.8].

Figure 2.4, for instance, is invalid code in Java 1.4 due to the assignment of incompatible types, but is valid in 1.5+ due to the rules defining implicit conversion between primitive types (i.e., int) and their object variants (i.e., Integer) defined within the language specification and implemented within the compiler.

```
1 // Auto-boxing (Java 5+)
2 int myIntPrimitive  = 5;
3 Integer myIntObj    = myIntPrimitive;
4
5 // Auto-unboxing (Java 5+)
6 int myIntPrimitive2 = myIntObj;
```

Figure 2.4: Java 5+ auto-boxing/unboxing example

Type Inference

Type inference is a compile-time Java construct where the compiler automatically infers the type of an expression or variable and is described in the Java Language Specification [13, § 18]. A full discussion of this topic is not relevant to this paper and only a short discussion is provided for background.

In the original Java language, all types were explicitly input into source code and checked by the compiler. In a departure from this approach, Java 7 [17] introduced the

diamond on the right-hand side (RHS) of the equals, for instance, when instantiating a parameterized type and assigning it to a variable of clear type. In this case the required instantiation type is obvious to the compiler and this simple type inference is performed at compile time – it is not necessary to explicitly add it. See the right-hand-side of line 2 in Figure 2.3 for an example of this simple type inference.

Further expansion of type inference was specified for Java 8 [18] in support of Lambda formals and in Java 10 [13] on the left-hand side (LHS) of the equals to infer the type of a declaration based on its usage without explicit type declaration [14].

2.2 Compiler

Java's standard compiler, javac, is a component of OpenJDK, is written in Java, and is responsible for accepting Java source code as input and emitting Java bytecode in the form of a platform-neutral Java .class file. To complete this operation, javac executes seven phases as described in and summarized below [19].

2.2.1 Parse

In the initial step, javac starts with an unknown raw input file (hopefully containing valid Java code) and parses it into a stream of tokens. This token sequence is the input to construct an abstract syntax tree (AST) representing the input file [19].

2.2.2 Enter

Each node in the abstract syntax tree (AST) is visited and each program symbol is registered and assigned its appropriate scope based on its tree position. Each class' parameters, interfaces, and parent are determined and scoped to the class. At the end, the set of top-level classes is saved into a queue for the attribute step [19].

2.2.3 Annotate

Java compiler annotations are a compiler extension mechanism whereby compiler operation may be extended in limited ways at compile time based on annotations added within the code [19].

2.2.4 Attribute

Each top-level class is evaluated to determine which external items are referenced, which may trigger parsing and entering of additional source files. In this phase, name resolution and type checking occur as well as the conversion of run-time literals to compile-time constants, which is referred to as constant folding [19].

2.2.5 Flow

This step reviews the tree to ensure all statements are reachable, all variables are used, final variables are assigned only once, and checked exceptions are handled [19].

2.2.6 Desugar

The desugar phase converts elements of the language that exist only in the Java Language Specification for source code and are not supported within JVM bytecode. These include inner classes, foreach loops, assertions, and class literals [19].

2.2.7 Generate

This step outputs the Java .class files from the de-sugared abstract syntax tree according to the input source and compiler options [19].

Bytecode Instruction Groups

- Load/store
- Conversion
- Arithmetic/logic
- Object creation/manipulation
- Operand stack management
- Control transfer
- Method invocation and return
- Specialized tasks (i.e., exceptions)

Figure 2.5: Java 10 bytecode instruction groups

2.3 Bytecode

Jave bytecodes are the instruction set of a fictionalized machine – the Java Virtual Machine (JVM). Each Java bytecode instruction is composed of one byte that represents the opcode along with zero or more bytes for operands. Approximately 80% of the 256 available bytecodes are presently in use as of JDK10.

The bytecode mnemonic prefix/suffix refers to the operand type and numerics refer to source/target of the operation. I.e., d2f converts double to float, fload_0 loads a float from local variable 0 and is a faster form of fload with operand 0.

Some bytecodes not in use according to the standard are used internally by the compiler or HotSpot for various purposes, particularly in HotSpot where bytecodes are re-written in memory with unused bytecodes that represent fast paths – more efficient codepaths where the necessary safety pre-conditions have been previously validated by the JVM and may be skipped going forward.

Prefix/Suffix	Operand Type	Prefix/Suffix	Operand Type
a	Reference	f	Float
b	Byte	i	Integer
c	Char	l	Long
d	Double	s	Short

Table 2.1: Java Bytecode Prefixes

2.4 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is a program that executes on a host computer and emulates many aspects of an actual computing machine: registers, instruction pointer, memory, pointers, IO, etc. This program is intended to accept machine instructions of a fictionalized computer called the Java Virtual Machine. These instructions are called bytecode. While it is possible to create a computer that natively executes Java bytecode [20], that is not usually the point.

The Java Virtual Machine accepts bytecodes in a .class file as input, translates the bytecode instructions into the native instructions of the host computer, and executes those instructions.

In this way, compiled Java bytecode may be executed on many different types of host computer without change and without regard for the target processor, architecture, number of registers, etc. Further, platform-specific optimizations may be flexibly selected and applied by the JVM at the time of execution rather than being targeted in advance at compile time.

The key, of course, is whether a Java Virtual Machine exists for the host computer's architecture. Generally this is not a significant problem. The reference implementation of the JVM is HotSpot, which is part of OpenJDK and is licensed under the GNU Public License v2 with the classpath exception.

As of August 2018 HotSpot implementations exist within the main OpenJDK repository for a wide variety of platforms including Windows x86/x64, Linux x64, Solaris x64/SPARC, MacOS x64, and ARM 32/64. Third parties provide additional ports of HotSpot to other platforms such as IBM's port for zOS/POWER.

A "Zero" HotSpot implementation within the main OpenJDK repository excises platform-specific assembly code with the goal of simplifying porting to a new plat-

form at the initial cost of performance. Native optimizations may be subsequently implemented over time to improve performance.

JVMs have important aspects that depart from this fictionalized machine metaphor. These aspects simplify the job of the typical software engineer and are outlined below. HotSpot is no exception.

2.4.1 Garbage Collectors

The purpose of the garbage collector is to re-claim memory that was previously allocated but is no longer in use. This effectively provides automatic memory management. A variety of methods are employed to efficiently re-claim unused memory. The intent is to free the software engineer from direct memory management.

It is noteworthy that garbage collection / automatic memory management is not necessarily required. Recently, OpenJDK provided a "no op" garbage collector which does nothing and is intended for applications that are short-lived or create little to no garbage [21] .

2.4.2 Bytecode Interpreter

Given the JVM is a fictionalized machine executing on non-fictional hardware, a translation must be undertaken from the Java bytecode to the underlying machine instructions. The bytecode interpreter executes native instructions for each Java bytecode and often includes some basic bytecode re-writing to improve performance.

2.4.3 Just-in-time Compilers

The Just-in-time (JIT) compilers implemented in many JVMs allow method execution to bypass the JVM's bytecode interpreter by asynchronously compiling and optimizing heavily-used methods into native code for more direct and faster execution.

Within the reference HotSpot JVM, two compilers are implemented using C++ and native assembler: c1 and c2. Both compile Java bytecode methods into native machine code of the underlying machine.

c1 is intended as a client compiler that has a fast warm-up and contains intermediate and less-costly optimizations [22]. The idea is that clients have fewer users, tend to run applications for shorter periods, and will exercise the code less intently than a multi-user server; consequently, this balance of fast compilation and fast startup makes sense for clients.

c2 is intended as a server compiler that has a longer warm-up and contains advanced and more-costly optimizations [22]. The idea is a server has a larger number of users, may run for a longer time, and will have methods that are more heavily used than in a client situation; thus a longer warm-up and more-costly optimizations may be a good trade-off for servers.

In reality, servers and clients use both compilers in HotSpot's current implementation, which employs a tiered compilation strategy to provide an optimal and dynamic level of optimization regardless of whether the host computer is a client or a server [23, § Tiered Compilation].

This paper is not about the JIT compilers and the specific optimization strategies. For the purposes of this paper, the above description is sufficient for understanding.

2.5 Summary

Java, as a whole, has remained an active and evolving development platform for more than two decades. While not designed as a research language [13, § 1], the platform continues to increase in capability, is actively maintained, is widely-used, and is a stable and open platform for extension. In these ways it is an appropriate and real-life platform for evaluating this paper's proposal.

Chapter 3

Storage Snapshots

This chapter provides an overview of storage snapshots, which are a common technology employed in storage systems. A high-level understanding is beneficial to the reader as snapshot terminology is used throughout the remainder of this paper.

3.1 Introduction

Similar to a photograph, a snapshot is a point-in-time image of something as it existed at a particular moment [24]. Storage Snapshots, meaning snapshot capability within a storage system, is a mature technology with multiple applications that have been discussed in the literature for several decades [25, 26, 27]. The design goals of storage snapshots are typically:

1. Provide a stable image of the storage state (the data) at a particular moment in time, either for faster recovery after a system failure or for other applications such as accidental file deletion recovery, backups, etc. This image lifespan may be short or long.

2. Do the above without consuming a large amount of space or overwhelming the limited I/O capabilities of the underlying storage hardware.

An overview of snapshot technology is provided by Xiao, et al. [28, § 2].

3.2 Application

Snapshot storage technology is widely-used and presently available in a variety of products including those from VMware, RedHat, and Microsoft, including nearly all commercial storage systems from IBM, NetApp, Dell, HPE, and others. These products allow consistent snapshots while a system is running as well as provide a point-in-time image of the system for rollback or other applications.

Storage snapshot technology must not be confused with the transaction control techniques utilized by database management systems logically described by Haerder [29] and later adapted for Software Transactional Memory [30] in an effort to increase software parallelism and reliability without resorting to traditional program lock semantics. This latter approach has its own research area around snapshots (i.e., [31]), which is distinct from the topic of this section. Further, storage snapshots should not be confused with backup and recovery solutions, which are intended to survive total media failure and therefore do not rely on the blocks of the active (running) image to partially compose the point in time backup image.

3.3 Approaches

Two primary approaches are employed to snapshot a running storage area: copy on write (COW) and redirect on write (ROW) [32, § IIA]. These are considered differential snapshots as they work by fixing either the active data or the snapshot (point-in-time) data as a coherent image and subsequently track the differential changes between the active image and the point in time [24]. Both approaches have trade-offs, which are discussed below. A third approach, which involves a complete copy of the data, is also described.

3.3.1 Copy-on-write

In the copy-on-write (COW) approach [28, § 2], a snapshot triggers an evaluation of writes and creation of a map indicating blocks altered since the snapshot as well as pointers into a snapshot area to the original-state block copies.

Upon write, if the write evaluation determines the target block has not been altered since the snapshot, the current value of the target block is copied to a new block in the snapshot area and the map is updated indicating the block has been altered along with a pointer to the block containing the original value. Subsequently, the write that triggered the copy continues and overwrites the target block.

To re-construct the original state at the time of the snapshot, the system uses the map to overlay the original block values over the active state of the volume. Removing the snapshot is as simple as removing or de-allocating the block copies.

The advantage of this approach is faster read performance of the active volume as the active-state data is not fragmented as it is with redirect-on-write. The disadvantage is the three I/O operations described above on the first write to a block post-snapshot [32, § IIA] [27, § 1].

3.3.2 Redirect-on-write

In the redirect-on-write (ROW) approach [28, § 2], a snapshot triggers a redirection of writes to a snapshot area and creation of a map indicating blocks altered since the snapshot as well as a pointer to the redirected active-state block.

If the write is to a block that has not yet been written since the snapshot, the map is updated indicating the block has been altered and a pointer to the redirected block is added. Subsequently, the write that triggered the redirection continues and is written to the redirected block in the snapshot area.

To re-construct the original state at the time of the snapshot, the system may simply read the original volume – it remains unchanged. Removing the snapshot is more I/O intensive as the system must use the map to copy the fragmented active-state values back to the original block locations.

The advantage of this approach over copy-on-write is the elimination of the three I/O operations required for the first write to a copy-on-write block. The disadvantage is the active-state data is fragmented in the storage medium, which may degrade sequential read performance; further, removing the snapshot is I/O intensive [32, § IIA] [27, § 1] as the active-state image is re-assembled by applying the redirected writes back to the source image.

3.3.3 Full image

A full image snapshot is simply an entire copy of the source storage area to a target storage area. This approach is non-differential, which means it carries the disadvantage of requiring $n \cdot x$ storage to store n snapshots of x amount of data. The advantage is fast read and write performance to both copies of the data [24].

In practice, a storage system is required to meet an I/O and time budget for a snapshot operation. A complete copy of a large volume can easily exceed the fixed budget and impair performance, which may make full copies infeasible in a live system. This difficulty may be addressed by combining differential and non-differential techniques.

In the case of NetApp[1] SnapMirror, the storage system maintains its I/O and time budget by combining techniques: an inexpensive (I/O and time) differential snapshot is created against the source volume, and this point-in-time differential snapshot image is then copied to the target [33, § 1.2.2].

[1]IBM N Series is a re-badged NetApp FAS

3.4 Discussion

Choosing a snapshot approach is an exercise in selecting trade-offs [24]. A **full-image** snapshot will require more storage than the two differential approaches and in practice may require an underlying differential snapshot implementation to meet time and I/O budgets, similar to NetApp's SnapMirror technology [33, § 1.2.2]. Within the differential snapshot methods:

Copy-on-write provides faster reads of the active-state image at the cost of three I/O operations for each block when it is first modified post-snapshot [28, § 2] and additionally allows for a lower-cost removal of a snapshot from media as the original (unchanged) blocks are outside the original image allocation and may simply be de-allocated. In essence, COW is "pay me now."

Redirect-on-write reduces the I/O necessary for each block when it is first modified post-snapshot at the expense of slower reads against the active state image and more expensive removal of snapshots [28, § 2]. In essence, ROW is "pay me later."

In addition to the type of snapshot employed, block sizes influence the relative performance of the implementation [34, § 4]. The specific bitmap technique employed to manage the block redirection step for reads and writes is an additional influence to consider [32]. The specific implementation details of these techniques are beyond the scope of this paper and mentioned here for the reader's benefit.

3.5 Summary

Storage snapshot technology has a long history and is widely employed in storage and virtualization systems as described within this chapter. In commercial systems the user/operator is generally not confronted with any choice of snapshot methods – it "just works."

But software engineers designing or implementing a snapshot facility in a new or existing system should: understand the options and trade-offs; choose a snapshot methodology based on the expected application parameters; and recognize that, even within the same methodology, implementation details may be reasonably expected to influence the overall performance and throughput of the system.

Chapter 4

Proposal: Intent-based Object Snapshots in Java

This chapter outlines a proposal to insert a new __**snap**__ keyword into the Java Language Specification [13] as a novel intent-based declaration that an actual method parameter must possess unshared state. The runtime then unshares the object state as-needed according to the design intent. The syntax and semantics of this proposed keyword are discussed herein. Chapters 5 and 6 subsequently outline two alternative implementation options for this proposal.

4.1 Introduction

As previously described, when a software engineer needs to fix a point-in-time state of a shared mutable input object, direct methods (clone, serialize, copy constructor) explicitly perform this operation within the method body. But each method has varying or under-specified outcomes that are not guaranteed, do not work universally, or do not clearly convey the intent.

Rather than explicitly perform one of the Java-native operations on the object within a method body, this proposal allows the method designer intent to be declared in the formal parameter declaration of the method signature, similar to **final**, and the development environment enforces the method design intent to ensure the state of the object indeed is not shared during execution.

```
1 // Snap/unshare objInput state prior to iterating
2 public void addAll(__snap__ ArrayList<String> objInput) {
3     for(String str : objInput) {
4         this.add(str);
5     }
6 }
```

Figure 4.1: __snap__ example

In Figure 4.1, 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.

To be clear, the intent is not to make a mutable object immutable, but rather to unshare its mutable state within a specific context as determined appropriate by the method designer.

The mechanism by which this is implemented is not as important as the guarantee that state alterations outside the method will not be visible within the method and vice-versa – unless the method explicitly takes action to the contrary. Consequently, in this section only the Java Language Specification [13] adaptations are discussed. Adaptations involving the JVM or development environment are discussed in subsequent chapters.

4.2 Abstraction

The abstract behavior of a formal method parameter declared with modifier **__snap__** may be intuitively understood as a guarantee that the object referenced in the actual parameter possesses an abstract state that is, at invocation of the method,

non-shared. The method may take subsequent steps to share the object with collaborating methods and objects during its execution, but at the time the method receives the actual parameter, the state is non-shared.

Given an object contains a graph of object relationships, the depth of this guarantee must be understood. For the purposes of this abstraction, the depth of the guarantee is to the object in question as well as its direct descendants. This is shown in Figure 4.2. Extending this guarantee deeper into an object graph similar to serialize is described in Chapter 8 as future work.

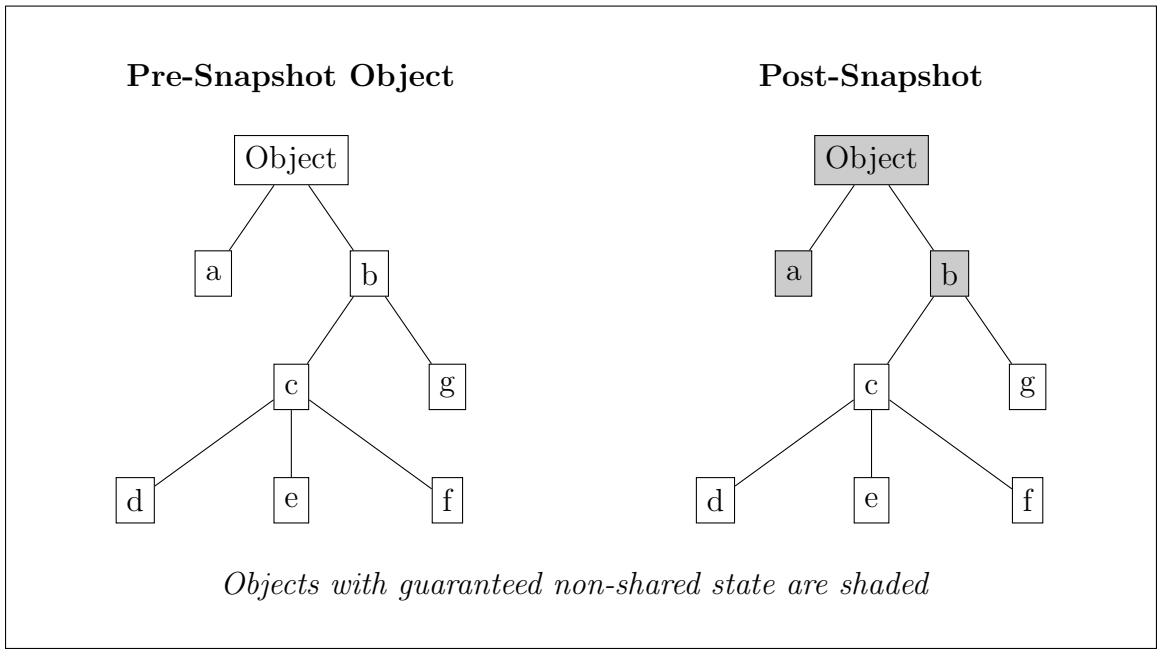

Figure 4.2: Depth of non-shared state guarantee

4.2.1 Type

The apparent and actual types of the original and point-in-time object instances are identical in this proposal.

4.2.2 Equality and Identity

The original (pre-snap) and point-in-time (post-snap) object instances may or may not be the same identity. Unlike clone(), serialize, object constructor, and third-party library, it is not guaranteed that identities will differ. Consequently, the equality operator may return true or false, depending on the circumstances. As outlined later within this chapter, circumstances exist when the method designer's intent of a point-in-time/non-shared state is realized without any action by the runtime. For example, an enum type is by definition an immutable type and therefore no action is needed to achieve the intent.

A discussion regarding identity equality of snapshots is laid out in Chapter 8, future work, including whether snapshots of an object should share a common identity.

4.2.3 Snapshot Navigability

New object instances created by the __**snap**__ keyword may have a distinct identity and in this proposal no mechanism is defined to navigate or analyze relationships among related identities beyond the Java-native double-equals equality operator.

4.3 Lexical Structure: Keyword __snap__

This proposal centers on a new Java keyword, __**snap**__, which requires the following adaptations to the Java Language Specification[13]. Keywords, by definition, cannot be used as identifiers [13, § 3.9]. To prevent naming collisions during the research, underscores are used within the __snap__ keyword (Figure 4.3) as a differentiator as snap could conceivably be an identifier extant within existing code.

Java Keywords

- abstract
- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- final
- finally
- float
- for
- if
- goto
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while
- _
- __**snap**__

Figure 4.3: JLS10 § 3.9 - adapted keyword list

4.4 Syntax: Keyword __snap__

The formal method parameter grammar defined in the Java Language Specification [13, § 8.4.1] must be modified to include __**snap**__ in addition to **final** as a valid modifier as shown in Figure 4.4.

The __**snap**__ keyword in this context is only valid for reference types, as primitives are unshared stack-based objects passed by value and therefore lack the same concerns as shared reference types of mutable state. This is expressed in Figure 4.5.

Given __**snap**__ is a declaration of design intent, it has no effect on enum types, which are immutable. In the case of enums, no compile or runtime error is raised and instead no snapshot operation is undertaken. This is expressed in Figure 4.6

```
FormalParameterList:
  ReceiverParameter
  FormalParameters , LastFormalParameter
  LastFormalParameter

FormalParameters:
  FormalParameter {, FormalParameter}
  ReceiverParameter {, FormalParameter}

FormalParameter:
  {VariableModifier} UnannType VariableDeclaratorId

VariableModifier:
  Annotation
  final
  __snap__

ReceiverParameter:
  {Annotation} UnannType [Identifier .] this

LastFormalParameter:
  {VariableModifier} UnannType{Annotation} ...
      VariableDeclaratorId
  FormalParameter
```

Figure 4.4: JLS10 § 8.4.1 - adapted formal parameter grammar

It is a compile-time error if **__snap__** appears more than once as a modifier for a formal parameter declaration or if the UnannType of the formal parameter declaration is not a reference type.

Figure 4.5: JLS10 § 8.4.1 - adapted formal parameter rules

It is neither a compile-time nor a runtime error if **__snap__** is a modifier on a formal parameter with apparent or actual type enum. Rather, at runtime no snap operation is performed on enum types.

Figure 4.6: JLS10 § 8.4.1 - adapted formal parameter rules for enums

4.5 Summary

In this chapter a set of modifications to the Java Language Specification [13] is described that accommodate a new keyword, __**snap**__. This keyword is lexically and syntactically defined in addition to its abstract meaning and behavior.

The following two chapters outline two alternative implementation strategies that align with the abstraction described in this Chapter.

Chapter 5

Transformation to Standard Java Syntax

This chapter discusses a transformation approach to implementing the __**snap**__ keyword from Chapter 4. A transformation approach accepts code in an enriched syntax or Domain-Specific Language (DSL) and transforms the input to another, usually standard, language. Herein, the transformation target language is Java 10.

5.1 Introduction

As outlined in Chapter 1, Java 10 lacks a native and universal functionality to snapshot – or fix – a point-in-time image of shared mutable object state with abstract semantics consistent in the view of the software engineer.

Consequently, when working with shared mutable objects in the standard Java language, knowledge of an object's type and type implementation may be required to understand what options, if any, are available for protecting shared object state against 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 is interacting with the 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.

5.2 Keyword __snap__

A method parameter modified by a **__snap__** keyword is demonstrated in Figure 5.1. The syntactical position of this modifier is similar to final. The **__snap__** keyword declares a need to snapshot or unshare the current object state after the function prologue but before method body uses the object instance.

```
1 // Create and return a snapshot of obj
2 public T snap(__snap__ T obj) {
3     return obj;
4 }
```

Figure 5.1: __snap__ keyword example: pre-transformation

The shared identity of the object passed to the method must not be visible to the method body; rather, only the identity of the unshared snapshot may be visible; in other words, the method body only has visibility to the snapshot copy of the input object but not to the actual input object itself. In this way, the method body works only with the non-shared mutable object state without concern for the underlying details of the snapshot operation involved in creating this non-shared state.

Out of the software engineer's view, the transformation removes the non-standard **__snap__** keyword from the method signature and inserts Java code atop the method body to copy each input **__snap__** object's state to a new identity.

The experimental mechanism in Figure 5.2 evaluates the object type to determine whether it may be cloned or serialized and – based on that information – to do so.

```
1 public T snap(T obj) {
2     obj = (SnapTree.getInstance()).snap(obj);   //inserted
3     return obj;
4 }
```

Figure 5.2: __snap__ keyword example: post-transformation

5.3 Problems with this Approach

Rather quickly this approach encounters a number of problems. First, the facility cannot be universal as it is not mandatory for all objects to be Cloneable, Serializable, or both. Some objects are neither and this facility could not work for them. Further, there is no effective mechanism for discovering copy constructors at runtime.

Second, the semantics of the snapshot operation may not be predetermined without knowing or inspecting the underlying input type. Particularly in the instance of clone, where there is explicitly no guarantee of semantics, this approach propagates this under-specified behavior to the method body. This variation in behavior may be surprising to a software engineer.

Third, serializing and cloning may throw checked exceptions that must be handled or propagated up the stack. Given the transparent intent of the snapshot operation, it is probably not appropriate to require these checked exceptions be added to the method signature. It is even less appropriate to eat the exception and return the original object, which breaks the contract and may cause future errors due to the sharing of state that the employing method innocently believed to be non-shared.

5.4 Summary

The three issues presented themselves early in the evaluation process. While the process abstracts some complexity to benefit the software engineer, the facility seems dangerous to use in practice.

Some improvements are likely possible at build time but the fundamental issues remain using the native capabilities. A further and possibly viable alternative could be to employ a non-native third-party mechanism such as the GSON [35], cloning [36], or other [9, 10, 11] libraries. This is described in Chapter 8 as future research.

Chapter 6

Direct implementation in OpenJDK 10

This chapter describes the process of directly implementing the __**snap**__ keyword from Chapter 4 in OpenJDK 10. This approach implements the keyword in the parse and generate phases of the javac compiler and defines a new JVM bytecode, 0xcb asnap in the Java Virtual Machine Specification [15]. The HotSpot 10 reference JVM is modified to accept the new bytecode and perform full-image object snapshots.

6.1 Introduction

As outlined previously, Java 10 lacks a native, universal, predictable method to unshare mutable object state. This chapter describes implementing full-copy object snapshots directly in Java 10 using an additional keyword and JVM bytecode. This approach bypasses the limitations of the native Java mechanisms. At the time experimental implementation commenced, OpenJDK 10 was the latest OpenJDK release. Both the javac compiler and HotSpot JVM are adapted.

While most of the implementation is platform independent, four lines of x64-specific code exist that are not presently ported to other platforms[1].

Within this chapter, code listings are reformatted to fit the written page. Updated or inserted code is indicated by the initials, MCD.

[1]See Chapter 8 for future work extending the functionality to further platforms

6.2 Java Virtual Machine Specification

The Java Virtual Machine Specification [15] defines the abstract operation of a compliant Java virtual machine while leaving many details of its internal representation (i.e., objects [15, § 2.7]) open to the JVM implementer as explicitly outlined in section 2.13 of the specification [15, § 2.13].

As this direct implementation proposal alters the abstract behavior of the Java Virtual Machine – in particular by modifying the list of bytecodes – the specification must be adapted to include the changes proposed. This section lists the changes needed and subsequent sections detail the implementation within HotSpot 10 on x64.

§ 2.11.1 Types and the Java Virtual Machine [15, § 2.11.1]. This table is adapted to include a new row, Tsnap, with asnap added to its intersection with the reference column (Table 6.1).

§ 2.11.5 Object Creation and Manipulation [15, § 2.11.5]. In this section, the asnap bytecode must be inserted as shown in Figure 6.1.

§ 4.10.1.9 Type Checking Instructions [15, § 4.10.1.9]. In this section, type safety constraints are defined for each relevant Java bytecode. Here instructions must be added for asnap that require the top of the operand stack be a pointer to a reference type (Figure 6.2).

§ 6.5 JVM Instructions [15, § 6.5]. Each bytecode is specified in this section including its form, format, operand stack pre/post condition, and semantics. Here asnap is added as a valid bytecode, 0xcb (Figure 6.3).

§ 7 Opcode Mnemonics by Opcode [15, § 7]. In this section an insertion is required to add asnap to the References mnemonic table (Figure 6.4).

With the specification amended, implementation within a target environment – in this case OpenJDK 10 on x64 – may proceed.

bytecode	**byte**	**short**	**int**	**long**	**float**	**double**	**char**	**reference**
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn
Tsnap								**asnap**

Table 6.1: Adapted Java Virtual Machine instruction set

> [...]
> Snapshot a class instance (resulting in a new object identity): asnap.

Figure 6.1: JVMS § 2.11.5 - adapted object creation and manipulation

```
[...]
asnap

An asnap instruction is type safe iff the one can validly
pop a reference type or array of reference types off the
incoming operand stack.

instructionIsTypeSafe(asnap, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame,
           [class('java/lang/Object', _PoppedStackFrame),
            arrayOf(class('java/lang/Object', _PoppedStackFrame))],
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
[...]
```

Figure 6.2: JVMS § 4.10.1.9 - asnap Type Checking Instruction

```
asnap
  Operation: Snapshot object instance
  Format:    asnap
  Forms:     asnap = 203 (0xcb)
  Operand Stack:
     ..., objectref →
     ..., objectref
  Description:
     The objectref on top of the operand stack is popped off,
     and a new object identity of the same type and size is
     created. The contents of the source object are copied
     to the new object identity. Direct reference type
     members within the new objectref are also copied anew
     such that a two-layer deep clone is the net effect.
  Notes:
     The asnap instruction does not call the object
     type's constructor.
```

Figure 6.3: JVMS § 6.5 - asnap bytecode specification

```
[...]
203 (0xcb)    asnap
```

Figure 6.4: JVMS § 7 - asnap mnemonic

6.3 Javac Compiler

Java's standard compiler, javac, is a component of OpenJDK and emits Java bytecode in the form of a .class file corresponding to a valid stream of input Java source code. The compiler completes a set of tasks in pursuit of this end as described in Chapter 2 and not repeated here.

This section discusses the modified parts of the javac compilation process necessary to implement the keyword **__snap__** and emit the new 0xcb bytecode, asnap.

6.3.1 Keyword: __snap__

Within javac a new keyword, **__snap__**, is defined, which is scoped to the formal parameter modifier of a method signature. This scoping position is similar to that of the formal parameter modifier final as seen in Figure 6.5 and consistent with the adapted Java Language Specification [13, § 8.4.1] described in Chapter 4.

```
1 // Create and return a snapshot of obj
2 public T snap(__snap__ T obj) {
3     return obj;
4 }
```

Figure 6.5: __snap__ keyword example

This keyword must be defined as a token for parsing, mapped to a new snap flag describing the formal parameter in the abstract syntax tree, and then the proper bytecodes emitted when loading an actual parameter bearing the **__snap__** keyword flag for its first use subsequent to method invocation.

Parse Step

Two declarations are required to represent **__snap__** internally within the compiler. First, declare **__snap__** a modifier for Java language elements (Figure 6.6).

Second, declare __**snap**__ flags and masks for Java language elements (Figure 6.7)

```
1  public enum Modifier { [...]
2      [...]
3      /** The modifier {@code static} */       STATIC,
4      /** The modifier {@code final} */        FINAL,
5      /** The modifier {@code __snap__} */     SNAP, /* MCD */
6      /** The modifier {@code transient} */    TRANSIENT,
7      [...]
```

Figure 6.6: src/java.compiler/share/classes/javax/lang/model/element/Modifier.java

```
1  /* Access flags & other modifiers for Java classes & members */
2  public class Flags { [...]
3      /* Standard Java flags.*/
4      public static final int PUBLIC  = 1;
5      public static final int PRIVATE = 1<<1; [...]
6      public static final long SNAP   = 1L<<40;  /* MCD */
7      [...]
8      public static final long
9          ExtendedStandardFlags      =
10             (long)StandardFlags | DEFAULT | SNAP, /* MCD */
11         ModifierFlags              =
12             ((long)StandardFlags&~INTERFACE) | DEFAULT | SNAP, /*MCD*/
13         InterfaceMethodMask        =
14            ABSTRACT | PRIVATE | STATIC | PUBLIC | STRICTFP | DEFAULT,
15         AnnotationTypeElementMask= ABSTRACT | PUBLIC,
16         LocalVarFlags              = FINAL | PARAMETER | SNAP, /* MCD */
17         ReceiverParamFlags         = PARAMETER; [...]
```

Figure 6.7: src/jdk.compiler/share/classes/com/sun/tools/javac/code/Flags.java

The parser maps the input sequence of Java tokens into an abstract syntax tree. First, __**snap**__ must be defined as a valid token for the javac token scanner to recognize it (Figure 6.8). Upon encountering this token, code is needed to set the corresponding modifier flags in the abstract syntax tree in method JavacParser.modifiersOpt() (Figure 6.9).

```
1  public enum TokenKind
2      implements Formattable, Filter<TokenKind>{ [...]
3      FINAL("final"),
4      SNAP("__snap__"),   /* MCD */ [...]
```

Figure 6.8: src/jdk.compiler/share/classes/com/sun/tools/javac/parser/Tokens.java

```
1  protected void skip(boolean stopAtImport,
2                      boolean stopAtMemberDecl,
3                      boolean stopAtIdentifier,
4                      boolean stopAtStatement) {
5      while (true) {
6          switch (token.kind) { [...]
7              case FINAL:
8              case SNAP:    /* MCD */
9              [...]
10 ParensResult analyzeParens() { [...]
11     outer: for (int lookahead = 0 ; ; lookahead++) {
12         TokenKind tk = S.token(lookahead).kind;
13         switch (tk) { [...]
14             case FINAL:
15             case SNAP:   /* MCD */
16             [...]
17 protected JCModifiers modifiersOpt(JCModifiers partial) {[...]
18     while (true) {
19         long flag;
20         switch (token.kind) { [...]
21         case FINAL: flag = Flags.FINAL; break;
22         case SNAP : flag = Flags.SNAP; break;   /* MCD */   [...]
```

Figure 6.9: src/jdk.compiler/share/classes/com/sun/tools/javac/parser/JavacParser.java

Generate Step

While mapping the de-sugared or flat (no inner classes, assertions, for-each loops, etc) Java abstract syntax tree to bytecodes in the Generate step, we need to set the "snap needed" flag on each formal method parameter bearing the snap flag. This step is performed in Gen.visitIdent() (Figure 6.10).

```
1  public void visitIdent(JCIdent tree) {
2      Symbol sym = tree.sym;
3      if (tree.name == names._this || tree.name == names._super) {
4          [...]
5      } else if (sym.kind == VAR && sym.owner.kind == MTH) {
6          /* MCD Set snap indicator if snap flag set */
7          if((sym.flags() & Flags.SNAP)==0
8              || (sym.flags() & Flags.PARAMETER) == 0) {
9              /* MCD Original code below */
10             result = items.makeLocalItem((VarSymbol)sym);
11             /* MCD Original code above */
12         }
13         else {
14             // Indicate snap is needed after load
15             LocalItem tmpResult = items.makeLocalItem((VarSymbol)sym);
16             tmpResult.needSnap = true;
17             result = tmpResult;
18
19             // Prevents needSnap from being set again and again
20             sym.flags_field -= Flags.SNAP;
21         }
22     }  [...]
```

Figure 6.10: src/jdk.compiler/share/classes/com/sun/tools/javac/jvm/Gen.java

To accomplish this step, the "snap needed" flag must first be added to the local variable classes and – if the flag was set (see Figure 6.10) – then when the variable is first loaded off the method stack, the "snap needed" flag is cleared, the asnap bytecode is emitted, the result is stored back on the method stack to replace the reference to the original object, and then the reference is re-loaded off the method stack to put the snapped reference back on top of the operand stack, which returns us to the same state as before the snapshot, but with the original reference replaced by the snapshot reference on the method stack and the operand stack (Figure 6.11).

The asnap bytecode is defined as 0xcb in Opcode.java and ByteCodes.java as shown in Figures 6.12 and 6.13. The bytecode must be specified in Instruction.balance() to prevent an invalid bytecode error. This step is shown in Figure 6.14.

```
1  /** An item representing a local variable. */
2  class LocalItem extends Item { [...]
3     /* MCD Indicates whether local var must be snapped on load. */
4     boolean needSnap = false;
5
6     Item load() {
7        if (reg <= 3)
8           code.emitop0(iload_0 + Code.truncate(typecode) * 4+reg);
9        else
10          code.emitop1w(iload + Code.truncate(typecode), reg);
11          /* MCD If a snap is required, snap on first load only */
12          if(needSnap) {
13             // Only snap on the first load, not after
14             needSnap = false;
15             // It only makes sense to snap reference types
16             if(type.isReference()) {
17                code.emitop0(asnap);  // Snap the reference
18                this.store();         // Store the new reference
19                this.load();          // Re-load new ref
20             }
21        }
22        /* MCD End Snap Logic */  [...]
```

Figure 6.11: src/jdk.compiler/share/classes/com/sun/tools/javac/jvm/Items.java

```
1  /**See JVMS, chapter 6.
2   * <p>In addition to providing all the standard opcodes
3   * defined in JVMS, this class also provides legacy support
4   * for the PicoJava extensions. [...] */
5  public enum Opcode {
6      NOP(0x0),
7      [...]
8      ASNAP(0xcb), /* MCD Add asnap Opcode */
9      [...]
```

Figure 6.12: src/jdk.jdeps/share/classes/com/sun/tools/classfile/Opcode.java

In Figure 6.15, asnap is declared along with its corresponding name and length. Figure 6.16 adds support to emit 0xcb asnap bytecode as a zero-parameter bytecode. For successful operation it is also necessary to define the bytecode mnemonic.

```
1  /** Bytecode instruction codes, as well as typecodes used as
2   *  instruction modifiers. [...] */
3  public interface ByteCodes {
4      /** Byte code instruction codes.*/
5      int illegal          = -1,
6          nop              = 0,
7          [...]
8          asnap            = 203, /* MCD Add asnap Opcode*/
9          ByteCodeCount    = 204; /* MCD Was 203 (last + 1) */
10         [...]
```

Figure 6.13: src/jdk.compiler/share/classes/com/sun/tools/javac/jvm/ByteCodes.java

```
1  /** Balance the stack */
2  int balance() {
3      switch (opc) {
4        case opc_dead:  case opc_label: case opc_iinc:
5        [...]
6        case opc_asnap: /* MCD Add asnap OpCode */
7          return 0;
8        [...]
9      }
10     throw new CompilerError("invalid opcode: " + toString());
11 }
```

Figure 6.14: src/jdk.rmic/share/classes/sun/tools/asm/Instruction.java

```
1  int opc_asnap = 203;  /* MCD Add asnap OpCode */ [...]
2  /* Opcode Names */
3  String opcNames[] = {  [...]
4      "breakpoint",
5      "asnap"              /* MCD Add asnap OpCode */
6  };
7  [...]
8  /* Opcode Lengths */
9  int opcLengths[] = {  [...]
10     1    /* MCD Add asnap OpCode */
11 };
```

Figure 6.15: src/jdk.rmic/share/classes/sun/tools/java/RuntimeConstants.java

```
1  /* Emit an opcode with no operand field.*/
2  public void emitop0(int op) {
3      emitop(op);
4      if (!alive) return;
5      switch (op) { [...]
6      case asnap:       /* MCD Add asnap Opcode */
7          break;[...]
8  private static class Mneumonics {
9      private final static String[] mnem =
10         new String[ByteCodeCount];
11     static { [...]
12         mnem[breakpoint] = "breakpoint";
13         mnem[asnap] = "asnap";   /* MCD Add asnap Opcode */
14          [...]
```

Figure 6.16: src/jdk.compiler/share/classes/com/sun/tools/javac/jvm/Code.java

In some cases optimizations are possible but at the expense of greater impact to the current code base. This is visible in the Generate step of javac where execution may be simplified by replacing the aload/asnap/astore/aload pattern with a more-complex asnap bytecode, but this would reduce the generality and simplicity of asnap's current function and require more extensive modification to the existing code base, which is outside the scope of this research[2].

6.3.2 Bytecode asnap (0xcb)

The new bytecode, 0xcb asnap, triggers an object-level snapshot within the JVM. This bytecode takes no parameters, pops an object reference off the operand stack, snaps the reference (creating a new instance), then pushes the new reference onto the operand stack. The implementation is described in the HotSpot JVM section below.

[2]See Chapter 8, future work

6.3.3 Supporting Tools

The scope of OpenJDK is well beyond a runtime and compiler tool chain. A set of supporting tools such as disassemblers and analyzers are included that rely on a knowledge of the Java Language Specification [13] and the Java Virtual Machine Specification [15], the knowledge of which is often represented in the form of Java classes. While it is not the purpose of this research to adapt supporting tools in OpenJDK, Figure 6.17 adds the snap flag to class and method dump output to ease troubleshooting and as an example.

```
1  /** Return flags as a string, separated by " ". */
2  public static String flagNames(long flags) {
3      StringBuilder sbuf = new StringBuilder();
4      int i = 0;
5      long f = flags & StandardFlags;
6      while (f != 0) { [...] }
7      return sbuf.toString();
8  }
9  //where
10     private final static String[] flagName = {
11         "PUBLIC", "PRIVATE", "PROTECTED", "STATIC", "FINAL",
12         "SUPER", "VOLATILE", "TRANSIENT", "NATIVE", "INTERFACE",
13         "ABSTRACT", "STRICTFP", "SNAP"}; /* MCD */
```

Figure 6.17: src/jdk.compiler/share/classes/com/sun/tools/javac/jvm/ClassWriter.java

6.4 HotSpot JVM

Java's reference JVM, HotSpot, is a component of OpenJDK and accepts compiled .class files containing Java bytecode input. HotSpot contains both a bytecode interpreter as well as two bytecode compilers, c1 and c2. Java bytecodes are initially executed by the bytecode interpreter and as the profiler detects that a method is heavily used (and meets certain criteria such as method size), it is compiled into

native instructions to improve execution performance. As the hot spots in the code may shift during execution, methods may go through multiple iterations of being interpreted, c1-compiled, c2-compiled, and back to interpreted. The mechanism of this operation is not the topic of this paper, but the intuition is important to understand.

This section discusses the modified parts of the HotSpot reference JVM necessary to minimally implement bytecode 0xcb asnap according to the semantics described earlier in this chapter. The basic requirement for the HotSpot JVM is to recognize the new 0xcb asnap bytecode as valid and when encountered:

1. Pop the reference off the operand stack
2. Determine the actual class of the object reference
3. Allocate a new, empty object of the same class on the heap
4. Copy the input object contents (two layers deep) to the new object[3]
5. Push the new object reference onto the top of the operand stack

HotSpot is primarily written in C++ with selected platform-specific assembler included for speed of execution. To keep the research size small, the four lines of necessary native code were implemented only for x64. Porting to other platforms is a task for future research.

The two layer copy performs the same basic copy operation for any direct class members of JVM internal type *instanceobject* or *arrayobject*. The result is effectively a predictable but non-deep copy of the object. Implementing a deep snapshot is a future activity described in Chapter 8.

[3]Extending depth to the entire object graph is discussed in Chapter 8 as future work

6.4.1 Snapshot semantics

The snapshot implemented within HotSpot is a full-image contiguous snapshot that in the current research iteration is a two-layer copy, meaning the object and its children are copied, but not the entire n-deep object graph.

Initially the author considered implementing a deep and differential snapshot with the block size being the individual field members within the object while maintaining a bitmap and relationship between the original and the snapshot similar to Copy-on-Write or Redirect-on-Write snapshots of a storage system.

To narrow the block size to individual object members within HotSpot would be a large undertaking outside the scope of this paper. HotSpot deeply and generally considers object fields to be contiguously allocated on the heap and a pre-determined distance from the top of the object heap location. This memory model representation is frequently exposed or its abstraction pierced in pursuit of maximum performance. Consequently, implementation would require adapting bytecode re-writing, platform-specific fastpaths, and significant modifications to the c1 and c2 compilers and garbage collection. This is deemed an interesting topic for future research but not feasible for the author at this time.

6.4.2 Bytecode 0xcb asnap

The declaration of the 0xcb asnap bytecode is shown in Figures 6.18 and 6.19. Each bytecode supported by HotSpot's bytecode interpreter is represented by an implementation class descending from class Bytecode. The additional class representing asnap is shown in Figure 6.20 and is a minimal implementation.

Bytecode verification for 0xcb asnap is left for a future task. To enable evaluation for this research, an empty validate function was created and is shown in Figure 6.21.

```
1  class Bytecodes: AllStatic {
2   public:
3    enum Code {
4       _illegal   =  -1,
5
6       // Java bytecodes
7       _nop       =   0, // 0x00 [...]
8       _asnap     = 203, // 0xcb /* MCD asnap Opcode */
9       [...]
```

Figure 6.18: src/hotspot/share/interpreter/bytecodes.hpp

```
1  void Bytecodes::initialize() { [...]
2    // Java bytecodes
3    // bytecode   name    fmt   wide?   result    tp  stk  traps
4    def(_asnap, "asnap", "b", NULL, T_OBJECT, 0, false);/*MCD asnap*/
5    [...]
```

Figure 6.19: src/hotspot/share/interpreter/bytecodes.cpp

```
1  /* MCD Begin asnap */
2  class Bytecode_asnap: public Bytecode {
3   public:
4    Bytecode_asnap(Method* method, address bcp):
5        Bytecode(method, bcp) { verify(); }
6    void verify() const {
7        assert(java_code() == Bytecodes::_asnap, "check_asnap");
8    }
9    // Returns index
10   long index() const   { return get_index_u1(java_code()); };
11 };
12 /* MCD End asnap */
```

Figure 6.20: src/hotspot/share/interpreter/bytecode.hpp

```
1  void ClassVerifier::verify_method(const methodHandle& m, TRAPS) {
2    HandleMark hm(THREAD);
3    _method = m;    // initialize _method
4    log_info(verification)("Verifying method %s",
5             m->name_and_sig_as_C_string());
6    [...]
7    while (!bcs.is_last_bytecode()) {  [...]
8        switch (opcode) {
9          case Bytecodes::_nop :  [...]
10       /* MCD Begin Snap Opcodes */
11         case Bytecodes::_asnap :
12           verify_asnap(index, &current_frame, CHECK_VERIFY(this));
13           no_control_flow = false; break;
14       /* MCD End Snap Opcodes */  [...]
15 /* MCD Begin Snap Opcodes */
16 void ClassVerifier::verify_asnap(u2 index,
17   StackMapFrame* current_frame, TRAPS) {
18   // TODO MCD Ensure operand is reference type
19 }
20 /* NCD End Snap Opcodes */
```

Figure 6.21: src/hotspot/share/classfile/verifier.cpp

6.4.3 Bytecode Interpreter

HotSpot executes the stream of Java bytecodes within a method using a template-based bytecode interpreter when the method is not presently compiled. Consequently, interpretation is the baseline mode of operation within the HotSpot JVM and was originally its only mode of operation [22].

The interpreter maintains a table of native assembly code with a template entry corresponding to each Java bytecode. At startup, the interpreter and this table are loaded into memory by InterpreterGenerator. This approach is more performant than a switch-based interpreter due the fewer number of compare operations and the utilization of the native C stack to pass its arguments [37].

Figures 6.22, 6.23, and 6.24 convey the addition of asnap to the bytecode interpreter as well as its native template table.

```
1 BytecodeInterpreter::run(interpreterState istate) {
2   [...]
3   const static void* const opclabels_data[256] = {   [...]
4 /* 0xC8 */ &&opc_goto_w,&&opc_jsr_w,
5              &&opc_breakpoint, &&opc_asnap /* MCD Add asnap */
6         [...]
7         switch (opcode) { [...]
8           CASE(_asnap): UPDATE_PC_AND_CONTINUE(1); /* MCD */
9           [...]
```

Figure 6.22: src/hotspot/share/interpreter/bytecodeInterpreter.cpp

```
1 [...] static void asnap(); /* MCD */ [...]
```

Figure 6.23: src/hotspot/share/interpreter/templateTable.hpp

```
1 void TemplateTable::initialize() {
2   if (_is_initialized) return;   [...]
3   /* MCD Begin Snap Bytecodes */
4   // Java spec bcode     ubcp|disp|clvm|iswd in    out   gen    arg
5   def(Bytecodes::_asnap, ubcp|____|clvm|____, vtos, vtos, asnap, _);
6   /* MCD End Snap Bytecodes */ [...]
```

Figure 6.24: src/hotspot/share/interpreter/templateTable.cpp

The native x64 assembly code the asnap bytecode referenced in the table entry above is shown in Figure 6.25. This assembly code:

1. Pops the object reference off the operand stack
2. Calls into the VM to perform the object snapshot (Fig. 6.26, 6.27)
3. Pushes the new object reference back onto the operand stack

Given the routine is native to x64, future effort is required to port the code to other platforms or elevate it to a platform-independent implementation.

```
1  /* MCD Begin asnap */
2  void TemplateTable::asnap() {
3      transition(vtos, vtos);
4
5      // Pop the atos off the operand stack
6      __ pop(rax);
7
8      // Call VM to snap atos. Creates new atos returned in rcx
9      call_VM(rcx, CAST_FROM_FN_PTR(address,
10                                   InterpreterRuntime::asnap),
11                                   rax);
12
13     // Push the atos back onto the operand stack
14     __ push(rcx);
15 }
16 /* MCD End asnap */
```

Figure 6.25: src/hotspot/cpu/x86/templateTable_x86.cpp

At this point it is necessary to understand HotSpot internally refers to Java class type representations as klass objects – due to class being a reserved word in C++. The klass object contains the method vtable (behavior) and defines the in-memory layout of all object instances of that klass (layout).

The contents of each object instance is stored on the heap and its access is represented by an oops (ordinary object pointer) heap object. The klass is required to correctly interpret the Java class content layout of an oops on the heap. Each oops begins with a mark word and a klass word. The former is comprised of a series of flags and the latter is a pointer to the klass object describing the behavior and layout of the oops.

The call into the VM from Figure 6.25 is dispatched into the InterpreterRuntime where a new C++ routine implements the snap operation. The snap operation uses the object reference popped off the operand stack as input and works as follows. This rundown is organized using the line numbers shown in Figures 6.26 and 6.27.

Lines 5-7: Using the object reference provided as input, derive the Klass object pointer from the object reference's klass word. The klass is necessary to determine the layout of the oops object, including its length.

Lines 9-11: First ensure we are not about to try to instantiate an abstract class or snapshot using a klass definition that is not yet initialized.

Lines 13-14: Create a new oops object: using the klass derived from the source object, allocate a new oops on the heap. This ensures the new oops is of correct size.

Line 17: Calculate the size of the oops body (sans mark and klass words).

Lines 19-22: Copy the old oops body bytes to the new oops body.

Lines 24-29: To copy the first layer of object members, we need the layout of the members. Get the $InstanceKlass$ object, which will provide this layout. Instantiate a field descriptor, fd, which will be used to assess each field.

Line 32: Loop over each field in the object.

Lines 33-35: Determine the field type using the field descriptor, fd.

```
1  /* MCD Begin Snap Support */
2  IRT_ENTRY(void, InterpreterRuntime::asnap(JavaThread* thread,
       oopDesc* srcobj))
3    assert(oopDesc::is_oop(srcobj,true), "must be a valid oop");
4
5    // Get the klass object reference from the oopHeader & cast to
         instanceKlass
6    Klass* srck = srcobj->klass();
7    InstanceKlass* srcik = InstanceKlass::cast(srck);
8
9    // Make sure we are not instantiating an abstract klass & klass
         is initialized
10   srcik->check_valid_for_instantiation(true, CHECK);
11   srcik->initialize(CHECK);
12
13   // Allocate the new oop
14   oop dstobj = srcik->allocate_instance(CHECK);
15
16   // Calculate size in bytes of object body (size - mark word)
17   int oopbodysize = srcik->layout_helper()-HeapWordSize;
18
19   // Copy the oops field contents (no deep copy for now)
20   Copy::conjoint_jbytes( ((markOop*)srcobj)+1, // source
21                          ((markOop*)dstobj)+1, // dest
22                          oopbodysize);         // bdy sz (bytes)
23
24   // Get the klass object reference from the oopHeader & cast to
         instanceKlass
25   Klass* dstk = dstobj->klass();
26   InstanceKlass* dstik = InstanceKlass::cast(dstk);
27
28   fieldDescriptor fd;
29   int length = srcik->java_fields_count();
30
31 // Loop over the object fields
32   for (int i = 0; i < length; i += 1) {
33     // Get the field descriptor for the field
34     fd.reinitialize(srcik, i);
35     BasicType ft = fd.field_type();
```

Figure 6.26: src/hotspot/share/interpreter/interpreterRuntime.cpp (1 of 2)

Lines 38-39: Ignore static fields and fields that are neither object nor array.

Lines 40-42: Get the oop of the field member. Due to lines 38-39 it will either be an array or an object. From the oop's klass word get a reference to the klass.

Lines 45-50: If the field member is an object, get the layout via InstanceKlass, allocate the new oop on the heap, and calculate its body size based on its layout.

Lines 51-64: If the field member is an array: in this branch both object and type arrays must be considered. Object arrays are an array of objects and type arrays are an array of primitives. Get the layout via TypeArrayKlass or ObjArrayKlass, allocate the new oop on the heap, and calculate its body size based on its layout.

Lines 66-69: Copy the body of the object member oops to the new oops similar to lines 19-22.

Lines 70-71: Store the newly-created member object reference in the new object's member reference field.

Line 75: Return the pointer to the new object back to the assembler template routine.

```
37      // Only snap non-static obj arr and object instance fields
38      if (!fd.is_static()) {
39        if(ft == T_ARRAY || ft == T_OBJECT) {
40          // Get the instance/array object
41          oop srcfldobj = srcobj->obj_field(fd.offset());
42          Klass* srcfldk = srcfldobj->klass();
43          oop dstfldobj;
44          int oopfldbodysize;
45          if(ft==T_OBJECT) { // Object
46            InstanceKlass* srcfldik = InstanceKlass::cast(srcfldk);
47            // Allocate new oop
48            dstfldobj = srcfldik->allocate_instance(CHECK);
49            // size - mark word
50            oopfldbodysize=srcfldik->layout_helper()-HeapWordSize;
51          } else { // T_ARRAY
52            ArrayKlass* srcfldak;
53            if(srcfldk->is_typeArray_klass()) {  // Type Array
54              srcfldak = TypeArrayKlass::cast(srcfldk);
55            } else { // Object Array
56              srcfldak = ObjArrayKlass::cast(srcfldk);
57            }
58            // Allocate new oop
59            dstfldobj = srcfldak->allocate_arrayArray(1,
60                          ((arrayOop)srcfldobj)->length(),CHECK);
61            // header + element_size * num_elements
62            oopfldbodysize = srcfldak->array_header_in_bytes()
63                           + ((1 << srcfldak->log2_element_size())
64                           * ((arrayOop)srcfldobj)->length());
65          }
66          // Copy the oops field contents (no deep copy for now)
67          Copy::conjoint_jbytes(((markOop*)srcfldobj)+1,//src af.mw
68              ((markOop*)dstfldobj)+1,  // dest after mark word
69              oopfldbodysize);          // body size in bytes
70          // Update the field reference
71          dstobj->obj_field_put(fd.offset(),dstfldobj);
72        }
73      }
74    }
75    thread->set_vm_result(dstobj);  // Return the new oops
76  IRT_END
77  /* MCD End Snap Support */
```

Figure 6.27: src/hotspot/share/interpreter/interpreterRuntime.cpp (2 of 2)

Figure 6.28 adds asnap support to the parser used by the optimization engine, which requires all bytecodes encountered be defined.

```
1 void Parse::do_one_bytecode() { [...]
2   switch (bc()) {
3   case Bytecodes::_asnap:   // MCD Fall-through; Snap Support
4   case Bytecodes::_nop:
5     // do nothing
6     break; [...]
```

Figure 6.28: src/hotspot/share/opto/parse2.cpp

Interpreted methods are mapped during garbage collection with the maps stored in the OopMapCache. HotSpot generates these maps during garbage collection as it is walking the thread stacks to determine which objects are in use on the stack. Figure 6.29 shows adding asnap to the OopMap generator.

```
1 // Sets the current state to be the state after executing the
2 // current instruction, starting in the current state.
3 void GenerateOopMap::interp1(BytecodeStream *itr) { [...]
4   // abstract interpretation of current opcode
5   switch(itr->code()) { [...]
6     case Bytecodes::_asnap:
7       break; /* MCD ASnap Support */ [...]
```

Figure 6.29: src/hotspot/share/oops/generateOopMap.cpp

HotSpot's BCEscapeAnalyzer class conservatively analyzes code blocks at the bytecode level to determine the escape state of objects used by a code block under analysis. The determination algorithm [38] allows HotSpot to perform optimizations such as elimination of synchronization locks for thread-local objects for which there can be no contention [23, § Escape Analysis]. Escape states are computed as follows:

1. **None**: The object cannot be used outside the scope

2. **Arg**: The object is passed as an argument but is otherwise unobservable outside the scope

3. **Global**: The object escapes the method: it is either returned, stored in a static field, or stored in an object that escapes the method.

In future research, the intent for non-shared state conveyed by **__snap__** could conceivably be combined with the escape analysis to shed the additional asnap/astore/aload bytecode pattern in situations where the input object's escape state is such that the snap operation has no net benefit. This would allow intent-based defensive programming with a minimum performance penalty and would need to address the fact that the escape analyzer is executed lazily and its analysis is not always available to the runtime [23, § Escape Analysis].

Bytecode asnap is added in Figure 6.30 as all bytecodes must be represented in the switch statement, but the implementation is minimal and further work is needed to properly analyze the input and output of the asnap bytecode.

```
1  void BCEscapeAnalyzer::iterate_one_block(ciBlock *blk,
2       StateInfo &state, GrowableArray<ciBlock *> &successors) {
3   [...]
4   while (s.next() != ciBytecodeStream::EOBC()
5      && s.cur_bci() < limit_bci) {
6      fall_through = true;
7      switch (s.cur_bc()) {
8        case Bytecodes::_nop:
9        case Bytecodes::_asnap:   // MCD Snap Support
10         break;   [...]
```

Figure 6.30: src/hotspot/share/ci/bcEscapeAnalyzer.cpp

The type flow analyzer requires all bytecodes be identified; consequently, asnap is added to the analyzer switch statement in Figure 6.31.

```
1 bool ciTypeFlow::StateVector::apply_one_bytecode( [...]
2   switch(str->cur_bc()) { [...]
3   case Bytecodes::_asnap:  // MCD Snap Support
4   case Bytecodes::_return:
5     { break; }  // do nothing. [...]
```

Figure 6.31: src/hotspot/share/ci/ciTypeFlow.cpp

The bytecode assembler is modified in Figure 6.32 to insert support for asnap. This class is used by the VM to create new methods such as during default method analysis when new overpass methods are generated.

```
1 /* MCD Begin Snap Support */
2 void BytecodeAssembler::asnap() {
3   _code->append(Bytecodes::_asnap);
4 }
5 /* MCD End Snap Support */
```

Figure 6.32: src/hotspot/share/classfile/bytecodeAssembler.cpp

6.4.4 c1 and c2 Just-in-time (JIT) compilers

Intuitively, when HotSpot determines[4] a method is "hot" and meets specific criteria such as method size, the method is compiled into native code and the bytecode interpreter is bypassed for subsequent executions. The Just-in-time (JIT) compilers implemented in many JVMs perform this compilation asynchronously and compile heavily-used methods into native code for more direct and faster execution.

HotSpot's c1 compiler is intended as a client compiler that has a fast warm-up and contains intermediate and less-costly optimizations [22]. The idea is that clients have fewer users, tend to run applications for shorter periods, and will exercise the code less intently than a multi-user server; consequently, this balance of fast compilation

[4]The rules and mechanism for this determination are outside the scope of this paper

and fast startup makes sense for clients.

HotSpot's c2 compiler is intended as a server compiler that has a longer warm-up and contains advanced and more-costly optimizations [22]. The idea is a server has a larger number of users, may run for a longer time, and will have methods that are more heavily used than in a client situation; thus a longer warm-up and more-costly optimizations may be a good trade-off for servers.

In reality, servers and clients use both of these compilers in HotSpot's current implementation, which employs a tiered compilation strategy to provide an optimal and dynamic level of optimization regardless of whether the host computer is a client or a server [23, § Tiered Compilation].

This paper does not alter the c1 and c2 compilers to compile the 0xcb asnap bytecode due to time constraints. When encountering an unknown bytecode, the compiler will fail. For expected results as described in this chapter and in the evaluation, the c1 and c2 compilers must be suppressed via command line to ensure the runtime executes only with the bytecode interpreter. Adapting c1 and c2 is future work described in Chapter 8.

6.5 Summary

To implement object-level snapshots directly in OpenJDK 10, several changes were needed. First, the javac compiler was modified to declare the **__snap__** keyword and 0xcb asnap bytecode, which takes zero parameters. The parse step and abstract syntax tree within javac were modified to map the new keyword within a method's formal parameter declaration into a new flag within the abstract syntax tree.

Second, within the javac generate step the snap flag, if set in the abstract syntax tree, triggers the output of an asnap, astore, and aload bytecodes upon the first aload of a snapshot parameter subsequent to method invocation.

This sequence, when executed within the JVM, takes the aloaded object reference off the top of the operand stack, snaps it, pushes the new snapshot reference back onto the top of the operand stack, and then astores and aloads it again to update the object reference on the method stack and put the new object reference back on top of the operand stack. Recall this activity occurs during an aload and this set of operations results in a state similar to an aload, except the object now on the method and operand stacks is the snapped copy of the object.

Within the HotSpot JVM, declarations for bytecode 0xcb asnap are added but bytecode verification and escape analysis are left for future work. The bytecode interpreter is modified to accept the 0xcb bytecode and x64 assembler code is inserted into the template table to handle popping the object off the operand stack, calling into the VM using the object reference, and pushing the new returned snapshot object back onto the stack. Within the VM, the klass type of the object reference is determined and another object of the same type and size is created on the heap. The contents of the initial object are copied to the new object. For the new object, direct members are also copied to new objects. A deep copy facility is left as future work.

During the course of the implementation, various other support areas were minimally updated as well as the exit analyzer and type flow analyzer. The c1 and c2 compilers were not adapted and this task is left for future research as described in Chapter 8. Consequently, during evaluation, the JIT compilers must be disabled.

The outcome of this activity is an object-level snapshot facility that produces objects with a two-layer deep snapshot that can be triggered using the keyword __**snap**__ on the formal method declaration. This is a simple and natural means by which to unshare the state of an input object where shared state is not desired. This facility is sufficient to evaluate the prediction set out in Chapter 1. This evaluation is carried out in Chapter 7.

Chapter 7

Evaluation

"It is noticeably hard to predict the effect of optimization strategies in Java without implementing them."
- Michael J. Steindorfer and Jurgen J. Vinju

This chapter evaluates the experimental implementation described in Chapter 6 to determine its characteristics relative to the prediction set out in Chapter 1. It outlines the pertinent research questions relevant to a software engineering practitioner, outlines the process by which to evaluate these characteristics, then performs and summarizes the evaluation.

7.1 Key Research Questions

The four questions raised in Chapter 1 relative to the prediction are detailed below. Each question is described within the context of the current Java development environment as well as its criteria for evaluation.

1. Is the behavior predictable?
2. Is the operation universal?
3. What is the performance relative to other methods?
4. Is the operation native to the JDK?

7.1.1 Is the Behavior Predictable?

Software Engineers value predictable behavior that reduces the testing required to verify correctness relative to a specification. A key question: is the behavior of the operation predictable without understanding the type's underlying implementation?

The evaluation of this characteristic simply pertains to whether the operation produces a consistent outcome discernible in advance without a requirement to understand the concrete implementation of the type upon which the operation is performed. Within Chapter 8, a user study is suggested as future work to confirm whether software engineers generally find the behavior more predictable.

7.1.2 Is the Operation Universal?

A second limitation of the native facilities available relates to their non-universal nature of operation. When the method designer intends to unshare the state of a shared mutable object, a navigation of the available options by concrete type should not be necessary.

The evaluation of this characteristic pertains to whether it applies to all object instances regardless of concrete type – that is, its operation is universal.

7.1.3 What is the Performance Relative to Other Methods?

To some extent, the optimizations within the HotSpot JVM to fastpath clone() is a reflection of clone()'s frequent use. This frequency itself is a reflection of its value to software engineers who need to fix the state of a shared mutable object.

An evaluation of relative performance vis-á-vis existing native approaches is undertaken to assess the relative cost or benefit of using the snapshot method based on a simple benchmark suite.

The reader should clearly note this performance comparison is between:

1. A minimal and un-optimized implementation developed by one person after work and over some weekends, and

2. Native methods heavily optimized over an extended period of time by several software engineers.

Multiple articles explain the difficulty of evaluating Java performance due to the non-deterministic nature of JVM operation: from thread-scheduling to Just-In-Time Compilation to Garbage Collection to other JVM operations [39, 40]. Horkỳ et al. [41] point out multiple technical details confuse even simple scenarios and produce "tricky results."

The performance evaluation focuses on steady state execution rather than startup. The system under evaluation is the custom HotSpot 10 on x64, which is the target platform/JVM combination discussed in Chapter 6. Implementation and performance evaluations on other platforms and JVMs is not undertaken at this time as the functionality is currently implemented experimentally for one JVM and solely on x64.

At the outset of the performance evaluation, a warm-up cycle of the benchmarking is executed and its results discarded. This is appropriate for steady-state performance evaluation. Replay compilation is not considered as evaluation is performed with the bytecode interpreter only. Garbage collection is requested in between measurable activities to minimize some aspects of non-determinism.

Benchmark Suite Selection

The benchmark suite is a simple evaluation of relative performance. Given the implementation in Chapter 6 is a minimal and un-optimized research prototype, there are limits to the conclusions that may be drawn about performance. Still, it is desirable to understand the relative performance attributes.

With that in mind, two objects were created that model a small and simple object as well as a large and relatively complex object. The objects include a mix of member objects, primitive data types, object arrays, and primitive arrays.

The Benchmark Suite

The first object in the suite is the small object, which is defined in Figure 7.1 and consists of members: Integer[] (array), int, and String. This object overrides several methods inherited from its parent.

The second object in the suite is the large object, which is defined in Figure 7.2 and consists of members: int[], Object[], and String[] (all of array size 10000). This object overrides several methods inherited from its parent.

```
1  public class SmallObject implements Serializable, Cloneable {
2      public SmallObject(SmallObject cpySource) {// Cpy Constructor
3          this.int1         = cpySource.int1;
4          this.str1         = cpySource.str1;
5          this.intShortList = cpySource.intShortList.clone();
6      }
7      public SmallObject(int inInt) {           // Constructor
8          for(int i = 0; i < intShortList.length; i++) {
9              intShortList[i] = inInt;
10         }
11         int1 = inInt;
12         str1 = (Integer.valueOf(int1)).toString();
13     }
14     private Integer[] intShortList = new Integer[10];
15     private int int1;
16     private String str1;  [...]
17     // Override Methods
18     public Object clone() throws CloneNotSupportedException {
19         return super.clone();
20     }
21     public boolean equals(Object o) {
22         if (o == this) { return true; }                  // Ident
23         if (!(o instanceof SmallObject)) {return false;}// Type
24         SmallObject so = (SmallObject) o;
25         return (this.int1 == so.int1
26             && this.str1.equals(so.str1)
27             && Arrays.equals(this.intShortList,so.intShortList));
28     }
29     public int hashCode() {
30         int hash = 3;
31         hash = 17 * hash+Arrays.deepHashCode(this.intShortList);
32         hash = 17 * hash+this.int1;
33         return 17 * hash+Objects.hashCode(this.str1);
34     }
35 }
```

Figure 7.1: SmallObject.java

```
1  public class LargeObject extends SmallObject {
2      public LargeObject(LargeObject cpySource) {// Cpy Constructor
3          super(cpySource);
4          this.intList = cpySource.intList.clone();
5          this.objList = cpySource.objList.clone();
6          this.strList = cpySource.strList.clone();
7      }
8      public LargeObject(int inInt) {           // Constructor
9          super(inInt);
10         for(int i = 0; i < intList.length; i++) {
11             intList[i] = inInt;
12             objList[i] = intList[i];
13             strList[i] = objList[i].toString();
14         }
15     }
16     private int[]    intList = new int[10000];
17     private Object[] objList = new Object[10000];
18     private String[] strList = new String[10000];
19     // Override Methods
20     public Object clone() throws CloneNotSupportedException {
21         return super.clone();
22     }
23     public boolean equals(Object o) {
24         if (o == this) { return true; }                  // Ident
25         if (!(o instanceof LargeObject)) {return false;}// Type
26         if (!super.equals(o)) { return false; }          // Super
27         LargeObject so = (LargeObject) o;
28         return (Arrays.equals(this.intList, so.intList)
29              && Arrays.equals(this.objList, so.objList)
30              && Arrays.equals(this.strList, so.strList));
31     }
32     public int hashCode() {
33             int hash = super.hashCode();
34             hash = 37 * hash + Arrays.hashCode(this.intList);
35             hash = 37 * hash + Arrays.deepHashCode(this.objList);
36             return 37 * hash + Arrays.deepHashCode(this.strList);
37     }
38 }
```

Figure 7.2: LargeObject.java

The test harness accepts as parameters the number of repetitions and the number of objects to work with for each repetition. All actions beneath (a) and (b) calculate runtime. Garbage collection is requested before each measure operation.

1. Perform a self-check confirming the snap facility is present and operating
2. Perform the following steps for the number of repetitions desired
    (a) Create the desired number of small objects and store them in an ArrayList
        i. Loop over the objects array list twice (warm-up)
        ii. Clone each object in the array list
        iii. Snap each object in the array list
        iv. Serialize/De-serialize each object in the array list
        v. Copy each object in the array list via copy constructor
    (b) Create the desired number of large objects and store them in an ArrayList
        i. Loop over the objects array list twice (warm-up)
        ii. Clone each object in the array list
        iii. Snap each object in the array list
        iv. Serialize/De-serialize each object in the array list
        v. Copy each object in the array list via copy constructor
3. Output the results of the comparison

As previously described, all evaluations are performed using the bytecode interpreter: no evaluations are performed using the c1 or c2 JIT compilers as they are not modified by this paper. This suppression was effected by using the $-Xint$ command line option, which puts HotSpot into interpreted-only mode.

```
>java.exe -Xint edu.ecu.seng7000.ecumatt.ex2.Main 60 2

Self-Check Phase
--------------------
- main  PRE : myList.count=1 (2761399) [Yogi]
  - testNorm PRE : incoll.count=1 (2761399) [Yogi]
  - testNorm POST: incoll.count=3 (-1563221239) [Yogi,Norm,Stan]
- main  MID : myList.count=3 (-1563221239) [Yogi,Norm,Stan]
  - testSnap PRE : incoll.count=3 (-1563221239) [Yogi,Norm,Stan]
  - testSnap POST: incoll.count=1 (2581513) [Snap]
- main  POST: myList.count=3 (-1563221239) [Yogi,Norm,Stan]

Benchmarking Phase
--------------------
[...]
- Building 60 Small Objects -> 135,000 ns
- 60 Noop (warmup loop #1)  -> IDENT
- 60 Noop (warmup loop #2)  -> IDENT
- 60 Clones                 -> EQUIV -> 1,573,700 ns
- 60 Snaps                  -> EQUIV -> 2,232,700 ns
- 60 Serialize/Unserializes -> EQUIV -> 49,488,300 ns
- 60 Copy Constructors      -> EQUIV -> 2,911,600 ns

- Building 60 Large Objects -> 1,086,203,600 ns
- 60 Noop (warmup loop #1)  -> IDENT
- 60 Noop (warmup loop #2)  -> IDENT
- 60 Clones                 -> EQUIV -> 2,241,800 ns
- 60 Snaps                  -> EQUIV -> 5,348,300 ns
- 60 Serialize/Unserializes -> EQUIV -> 17,990,479,400 ns
- 60 Copy Constructors      -> EQUIV -> 7,118,100 ns
[...]
```

Figure 7.3: Example Benchmark Run

An example run is shown in Figure 7.3 and the snap and clone benchmark code is shown in Figures 7.4 and 7.5. The benchmark program outputs run data as matrices for plotting and analysis in GNU Octave or other tools. A violin plot of the run data is shown in the Evaluation Results section (Figure 7.7).

```
1  /********************** Snap ***************************/
2  System.out.printf("- %d Snaps                  ",cnt);
3  int i = 0;
4  objOutput    = new ArrayList<>();
5  elapsed       = 0;
6  for(T obj : objInput) {
7       System.gc();
8       start = System.nanoTime();
9       newObj = snap(obj);
10      end = System.nanoTime();
11      elapsed += end - start;
12      objOutput.add(newObj);
13      objTimeSnap.add(end - start);
14 }
15 System.out.printf(" -> ");System.out.printf("%s",padRight(
       evalEquiv(objInput,objOutput).toString(),10).substring(0, 5));
16 System.out.printf(" -> ");System.out.printf("%s ns\r\n",
       NumberFormat.getIntegerInstance().format(elapsed));
```

Figure 7.4: Benchmark Harness - Snap

```
1  /********************** Clone ***************************/
2  System.out.printf("- %d Clones                 ",cnt);
3  objOutput     = new ArrayList<>();
4  elapsed       = 0;
5  for(T obj : objInput) {
6       System.gc();
7       start = System.nanoTime();
8       newObj = (T) obj.clone();
9       end = System.nanoTime();
10      elapsed += end - start;
11      objOutput.add(newObj);
12      objTimeClone.add(end - start);
13 }
14 System.out.printf(" -> ");System.out.printf("%s",padRight(
       evalEquiv(objInput,objOutput).toString(),10).substring(0, 5));
15 System.out.printf(" -> ");System.out.printf("%s ns\r\n",
       NumberFormat.getIntegerInstance().format(elapsed));
```

Figure 7.5: Benchmark Harness - Clone

7.1.4 Is the Operation Provided by the JDK?

The JDK is a standalone, cross-platform development environment. While third-party externalities are important, the author's personal opinion is that third party tools should not be required to manage sharing of state among software modules.

This question is evaluated by assessing whether the operation may be performed using only the JDK and standard library.

7.2 Evaluation Process

The evaluation considers each research question individually and its evaluation process is described below.

Is the Behavior Predictable?[1] In this evaluation we evaluate whether the behavior of the operation may be predicted without knowledge of the underlying concrete type – for instance, an object instance of apparent type List. If the operation behavior may be predicted, then this binary evaluation is true; otherwise, false.

Is the Operation Universal? This evaluation considers whether all object instances may have the operation applied. If the operation may be applied without restriction, then this binary evaluation is true; otherwise, false.

What is the Performance Relative to Other Methods? This non-binary evaluation uses the benchmark suite described above to quantify performance of the operation relative to the other operations available.

Is the Operation Provided by the JDK? In this evaluation we evaluate whether the operation is natively implemented within the JDK or if additional externalities are required for the operation to succeed. If the operation may be executed using only the JDK, then this binary evaluation is true; otherwise, false.

[1] Chapter 8 suggests a future user study to assess further

7.2.1 System Under Evaluation

The system is the head of the OpenJDK 10 master repository as of July 1, 2018 built using the fastdebug target by Microsoft Visual C++ 2010 and containing the javac compiler and HotSpot JVM modifications described in the previous chapter.

7.2.2 Evaluation Environment

The evaluation was carried out using hardware described in Figure 7.6 in an unloaded and otherwise unburdened state. This hardware is a typical example of x64-based client hardware readily available at the time of the article's writing with no special or noteworthy abilities beyond its compact size.

Operating System:	Microsoft Windows 10 Enterprise 10.0.17134 Build 17134
Architecture:	x64-based PC
Processor:	Intel® Core™ i5-6300U CPU @ 2.40GHz, 2 Cores
Memory:	8 GB
Storage:	235.48 GB NVMe THNSN5256GPU7 TO w/BitLocker
Make:	Microsoft Corporation
Model:	Surface Pro 4
BIOS:	Microsoft Corporation 108.2318.769, 2018-08-14
Other:	Virtualization-based security Running

Figure 7.6: Evaluation Environment

When evaluating the performance of Java code, the literature emphasizes the need to evaluate the code on multiple platforms and multiple JVMs to provide a clearer picture of performance [40, 39]. In this paper, the evaluation is limited to HotSpot 10 on x64 as that is the platform of this experimental implementation.

Future research may expand the implementation to further platforms and discover noteworthy cross-platform and cross-VM performance variances. For the purposes of this paper, the findings are relevant only to HotSpot 10 on x64.

7.3 Evaluation Results

This section summarizes the evaluation results of each research question.

7.3.1 Is the Behavior Predictable?

While more work is suggested[2] to quantify how software engineers assess predictability, a straightforward test is used: regardless of the concrete type, is the operation's behavior predictable?

- **Clone()**

  False. This operation is under-specified and the API documentation states its behavior is type dependent [6, § Object.clone()]. These problems are discussed within the literature [7, 8]. Without evaluating the underlying type implementation, the behavior of this operation may not be consistently predicted.

- **Serialize**

  False. This operation is predictable to a larger extent than clone() in that it predictably replicates the entire object graph. A serialize operation may throw a NotSerializableException if any concrete type in the input object graph lacks the Serializable marker interface as described within the Java API documentation [6, § java.io.Serializable]. As it is necessary to consider the type of all objects within the object graph, this operation is not be evaluated as true.

- **Snap**

  True. In this proposal, the snap operation is specified with consistent behavior across types based on the method designer's intent and an evaluation of the actual type passed as input to the method (i.e., snapping immutable enums is

[2] See Chapter 8, future work

unnecessary). Chapter 4 and 6 discuss the specifics of this specification.

- **Copy Constructor**

  False. This operation is not provided natively by the JDK as evidenced by a search of Java API documentation [6] and is implemented by various designers in a non-universal type-specific manner.

7.3.2 Is the Operation Universal?

This research question is answered below for each operation under evaluation.

- **Clone()**

  False. This operation is not universally-supported for all concrete types as evidenced by the Java API documentation's description of the Cloneable marker interface [6, § java.lang.Cloneable].

- **Serialize**

  False. This operation is not universally supported for all concrete types as evidenced by the description of the Serializable marker interface within the Java API documentation [6, § java.io.Serializable].

- **Snap**

  True. In this proposal, the snap operation is specified as applicable to any concrete object instance as the method designer is specifying intent. The runtime may use its knowledge of the actual runtime type or escape analysis to omit the snapshot operation when there is no need to do so – i.e., enums. But the intent-based mechanism described in Chapter 4 and 6 remains intact and applies to all object instances.

- **Copy Constructor**

  False. This operation is not provided natively by the JDK as evidenced by a search of Java API documentation [6] and is implemented by various designers in a non-universal type-specific manner.

7.3.3 What is the Performance Relative to Other Methods?

The benchmark suite violin plots immediately show the performance penalty incurred by serialization relative to clone(), snap, and copy constructor. Perhaps due to the expensive round-trip operation, this penalty nearly ruins the plot scale (Figure 7.7) and pushes other methods to the bottom of the plot.

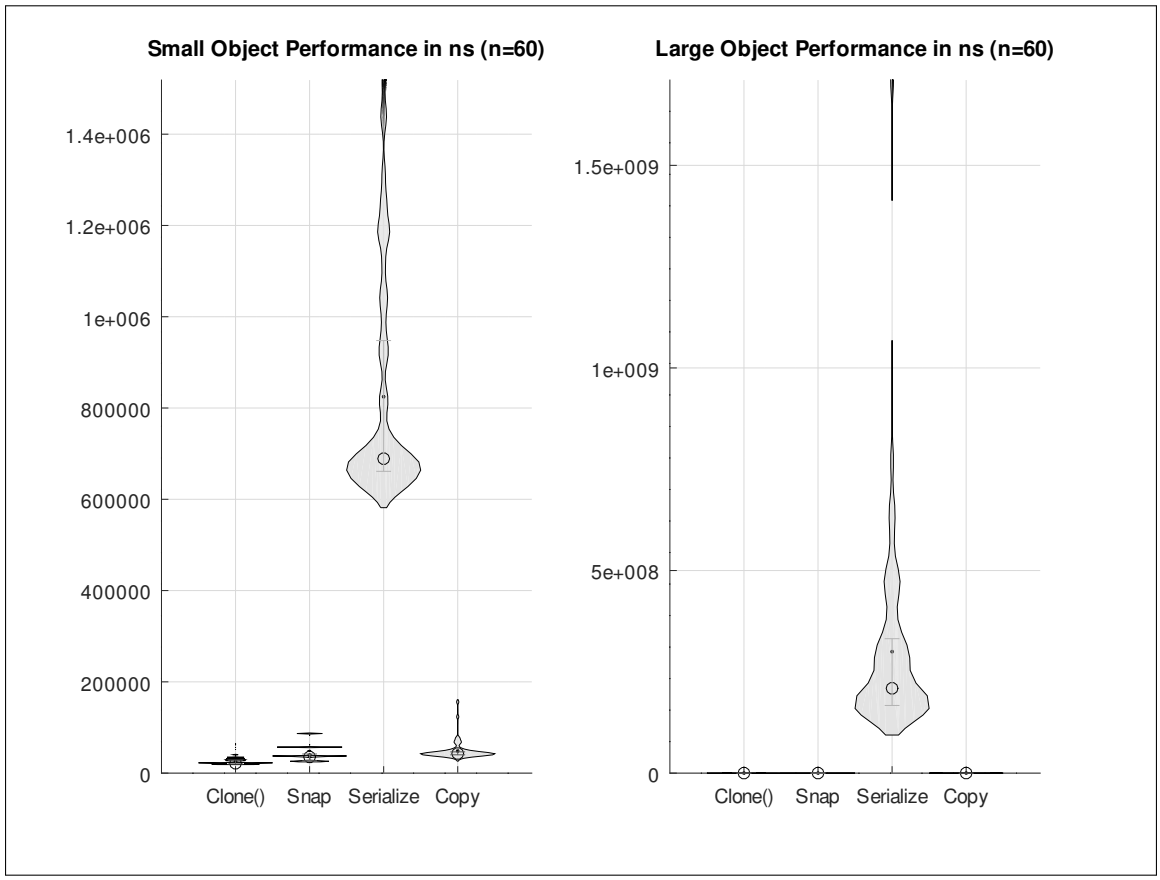

Figure 7.7: Benchmarking Results - Violin Plot

To improve the scale of the plot, Figure 7.8 removes Serialization and the scale significantly narrows. This plot clearly shows Copy Constructor takes a penalty (30% at the mean) relative to Snap. Snap clearly takes a penalty (42% at the mean) relative to clone() and is rarely faster.

It is notable that clone() in HotSpot is heavily optimized by many talented software engineers across many years of active development. Snap is not optimized; however, it benchmarks as more performant than two extant methods: copy constructor and serialization. Only the heavily-optimized clone() is consistently faster.

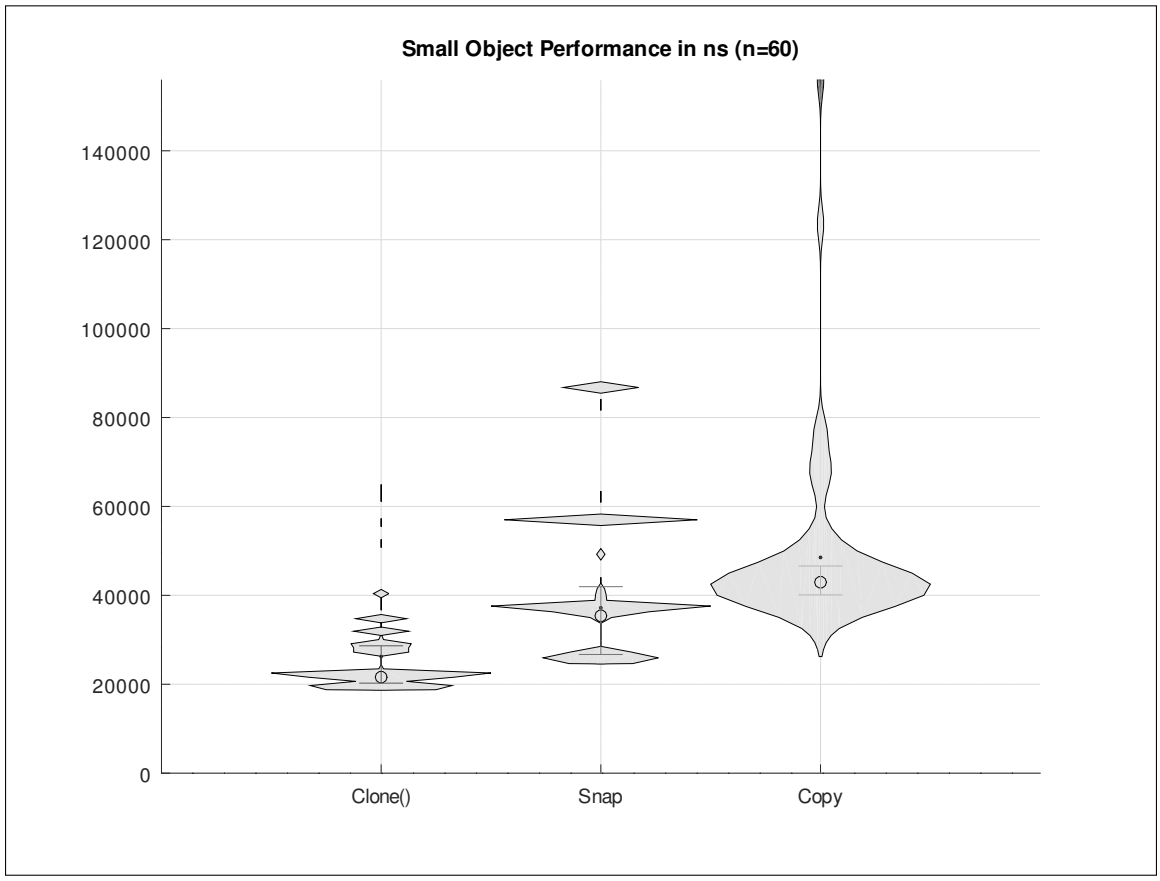

Figure 7.8: Benchmarking Results - Violin Plot - Small Object w/o Serialize

Figure 7.9 visualizes the benchmark results for the large object scenario without Serialization. This plot clearly shows Copy Constructor takes a penalty (31% at the mean) relative to Snap. Snap clearly takes a penalty (140% at the mean) relative to clone() and is rarely faster.

Again, clone() in HotSpot is heavily optimized and snap is not optimized; however, snap again appears more performant than two extant methods: copy constructor and serialization. Only the heavily-optimized clone() is consistently faster.

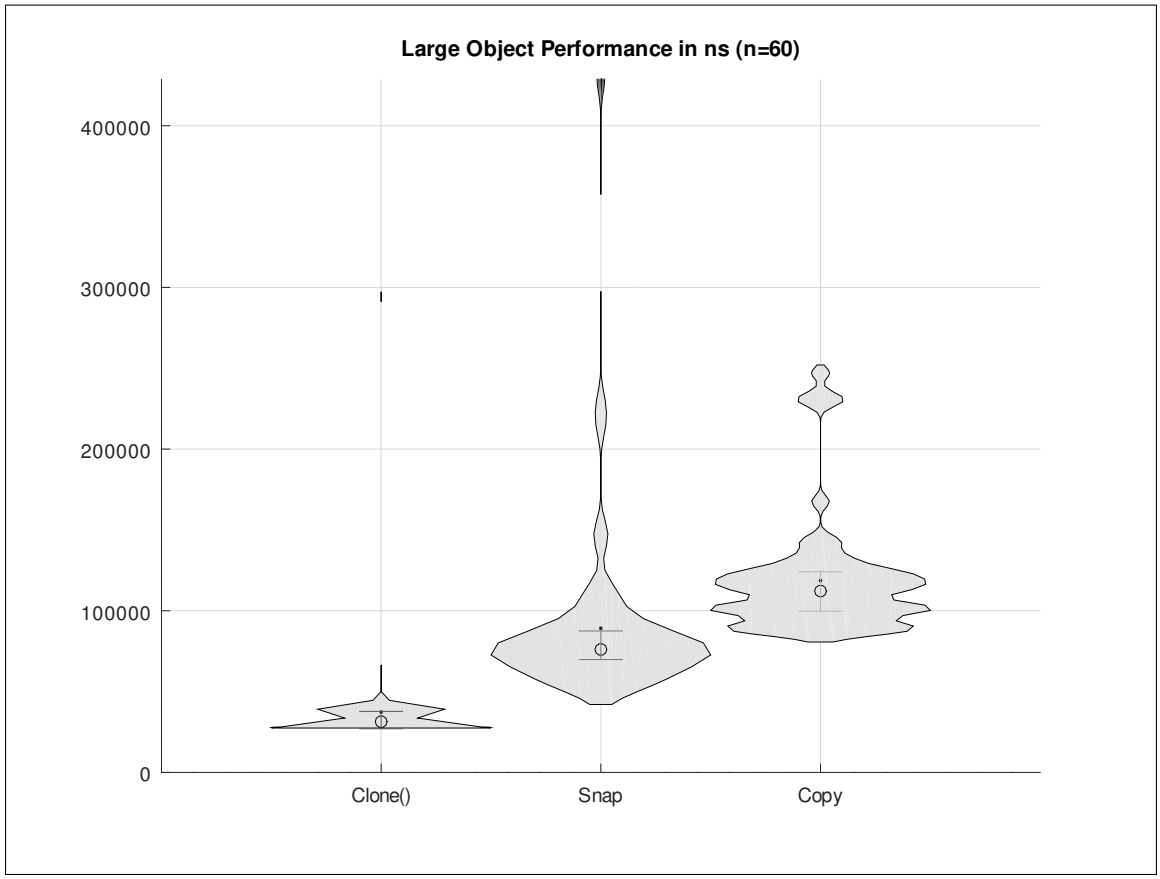

Figure 7.9: Benchmarking Results - Violin Plot - Large Object w/o Serialize

7.3.4 Is the Operation Provided by the JDK?

This research question is answered below for each operation under evaluation.

- **Clone()**

  True. This operation is provided natively by the JDK as evidenced by the Java API documentation [6, § Object.clone()].

- **Serialize**

  True. This operation is provided natively by the JDK as evidenced by the Java API documentation [6, § java.io.Serializable].

- **Snap**

  True. In this proposal, the snap operation is specified in a revised Java Language Specification to be a modifier on the method formal parameter declaration. Further details are provided in Chapter 6.

- **Copy Constructor**

  False. This operation is not provided natively by the JDK as evidenced by a search of Java API documentation [6]. If desired, the type designer must implement a copy constructor facility for each type undergoing design and implementation. It is true that no externalities are generally required to implement a copy constructor, but unlike clone() or serialization, it is not a native facility.

7.4 Summary

The evaluation proposed a method for answering the following research questions and implemented that method to arrive at the answers summarized in Table 7.1:

Is the Behavior Predictable?[3] In this evaluation we evaluate whether the operation behavior may be predicted without knowledge of the underlying concrete type – for instance, an object instance of apparent type List. If the operation may be predicted, then this binary evaluation is true; otherwise false.

Is the Operation Universal? This evaluation considers whether all object instances may have the operation applied. If the operation may be applied without restriction, then this binary evaluation is true; otherwise false.

What is the Performance Relative to Other Methods? This non-binary evaluation uses the described benchmark suite to quantify relative performance.

Is the Operation Provided by the JDK? This evaluation considers whether the operation is natively available within the JDK – that is, no externalities are required for the operation to succeed. If the operation may be executed with only the JDK under test, then this binary evaluation is true; otherwise false.

Table 7.1 summarizes the observations from this evaluation.

Research Question	Clone()	Snap	Serialize	Copy Constructor
Predictable?	False	True	False	False
Universal?	False	True	False	False
Provided by the JDK?	True	True	True	False
Performance Rank	1	2	4	3

Table 7.1: Summary of Evaluation Observations

[3]Chapter 8 suggests a future user study to assess further

Chapter 8

Future Work

Throughout this research and as described in the preceding chapters, areas of future work are apparent, some of which appear to be quite interesting. This chapter provides a suggested list of future work organized by approach with notes regarding potential issues and solutions, where known and as appropriate.

8.1 Direct JDK Implementation Approach

This section pertains to the direct JDK implementation described in Chapter 6, which utilizes the new 0xcb asnap bytecode to pop an actual parameter object reference off the operand stack after aload, create a point-in-time copy of a portion of the object graph, and replace the frame and operand on-stack reference.

8.1.1 Deep Snapshots

The abstraction described in Chapter 4 and the direct implementation in Chapter 6 is intended for research and is limited to snapshotting two layers of the object graph, similar to some implementations of clone(). Due to this limitation, which candidly is due to time constraints, its immediate utility may be less than desired.

As future work, the author suggests an empirical study to determine the distribution of typical object graph depths. In the author's view, snapshotting the entire

graph appears to be the more-compelling approach as it maximally unshares the object, but more analysis would refute or confirm this view. It is observed solutions such as serialize, GSON [35], and cloning libraries [36] choose to copy the entire graph.

A further option is to let the designer decide by additionally providing a __**deepsnap**__ keyword and corresponding 0xcc adsnap opcode that specifically performs a deep snap as a further alternative. In this scenario one must question how the method designer would intelligently select the "correct" keyword as the choice requires knowledge of type-specific behavior similar to the conundrum present today in standard Java, albeit without the limitations on universality or errant CloneNotSupported and NotSerializable exceptions.

8.1.2 Platform Independence

The direct implementation described in Chapter 6 utilizes one four line native x64 routine (Figure 6.25) that could be avoided as its operation is simply to pop the operand stack, call into the VM's platform-neutral code, then push the result back onto the operand stack. Eliminating this x64-specific masm would make the solution platform independent. An alternative to this approach is to implement the routine for other platforms.

8.1.3 Differential Snapshots

The implementation described in Chapter 6 is a compromise based on feasibility relative to a time limitation. The author's initial intent was to explore differential snapshots in HotSpot using a block size smaller than the object heap bucket. This change in approach was necessitated by a fuller understanding of HotSpot's pursuit of maximal performance through platform-specific fastpaths and shortcuts that bypass the fig leaf abstraction responsible for protecting the internal object representation.

In HotSpot, implementing differential snapshots would be quite an undertaking. Nearly all object member access is calculated as an offset from the top of the heap object based on an assumption of contiguous allocation and read directly by a dizzying array of routines, native and otherwise. Implementing differential snapshots would require a level of indirection: that these routines would need to check a data structure to determine the actual location of the object member block on the heap as it may reside in a different bucket. Recall differential snapshots store block data in multiple areas and the meaning of each area varies whether the differential strategy is Copy-on-Write or Redirect-on-Write.

What follows is a discussion for conceptual illustration and not a formal definition of a differential snapshot implementation within HotSpot. Within this section the operation of the snapshot area is not described.

Let a concrete type, $MyType$, descend directly from type $Object$ and possess two members, public int $member1$ and public int $member2$. Let $v1$ be a valid instantiation of $MyType$. In this instantiation, access to member $member1$ and $member2$ within the JVM is straightforward as it is a simple offset calculated from the top of the heap object. Consequently, accessing or mutating $v1.member1$ or $v1.member2$ uses basic pointer arithmetic and is unaltered from the current scenario.

Now let us introduce a differential snapshot. For the purposes of our example, we will utilize a redirect-on-write methodology to 0xcb asnap $MyType$ instance $v1$ and store the point-in-time snapshot as $v2$. When the block size is the object as in the Chapter 6 implementation, writes may continue across $v1$ and $v2$ as they operate entirely in separate spaces on the heap. But in a differential snapshot scenario, both versions of the object state – $v1$ and $v2$ – occupy the same memory locations at the outset: $v1.member1$ accesses the same physical address on the heap as $v2.member1$.

When a write to $v1.member1$ occurs, that write must not affect $v2.member1$ and

vice versa. Consequently, during the asnap operation an indirection bitmap must be activated for $v1$ and for $v2$. This bitmap indicates whether the member value is stored "here" (on this object) or "there" (at some other location) and whether the value is "shared" or "private." The bitmap for $v1$ is initialized as "here/shared." Further, $v2$ is created on the heap simply as an object header and bitmap. Its bitmap is initially a copy of $v1$'s before all "here/shared" entries are changed to "there/shared" with a pointer to $v1$. The current state of the system is as follows:

- Object instance $v1$:
  $member1$: "here/shared" $\rightarrow$ (local)
  $member2$: "here/shared" $\rightarrow$ (local)
- Object instance $v2$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$

In this initial state, a read of $v1.member1$ evaluates $v1$'s bitmap and as the $member1$ value is "here" (on this object) it therefore reads the value directly from $v1$'s object representation on the heap. Similarly, a read of $v2.member1$ evaluates $v2$'s bitmap and sees the value is "there" (on some other object) with a pointer to $v1$ and reads the same value ($v1.member1$) directly from $v1$'s object representation on the heap.

Writes are more interesting. A write to $v1.member1$, which is "here/shared" must be redirected to prevent affecting the value of $v2.member1$. Thus, $v1$'s bitmap for $member1$ is changed from "here/shared" to "there/private" with a pointer into a snapshot area on the heap to which the write is redirected. Subsequent reads and writes of $v1.member1$ are directed by the bitmap to the private snapshot block so as not to disturb $v2.member1$. The current state of the system is as follows:

- Object instance $v1$:
  $member1$: "there/private" $\rightarrow$ Snapshot area
  $member2$: "here/shared" $\rightarrow$ (local)
- Object instance $v2$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$

A write to $v2.member1$ is similar but slightly altered. As the bitmap value for the member is "there/shared," the write must be redirected. The bitmap value for the member is changed to "there/private" with a pointer into a snapshot area on the heap to which the write is redirected. Similar to the previous example, subsequent reads and writes of $v2.member1$ are redirected to the private snapshot block so as not to disturb the original block value, which is no longer referenced by anyone. The current state of the system is as follows:

- Object instance $v1$:
  $member1$: "there/private" $\rightarrow$ Snapshot area
  $member2$: "here/shared" $\rightarrow$ (local)
- Object instance $v2$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$

Let us now assume a second snapshot of $v1$ is taken, $v3$. When asnap executes, similar to $v2$ it creates a header-only object representation on the heap. Unlike $v2$ where all $v1$ bitmap entries were "here/shared," $v1$ currently has entries flagged "there/private." In this case and prior to bitmap copy, $v1$'s "there/private" flags are changed to "there/shared." $v1$'s bitmap is then copied to $v3$'s and in $v3$'s bitmap all

"here/shared" entries are changed to "there/shared" with a pointer to $v1$. [1] The outcome of this operation is as follows:

- Object instance $v1$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "here/shared" $\rightarrow$ (local)
- Object instance $v2$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$
- Object instance $v3$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "there/shared" $\rightarrow v1$

Now let us asnap $v2$ to a new object, $v2.1$. The process follows the same pattern as with $v2$ and $v3$. Note that the $v2$ and $v2.1$ references to $v1$ are to $v1$'s original values. For instance, $v2.member2$ clearly points to the same memory area as $v1.member2$ but it may not be clear without emphasis that $v2.member1$ continues to point to the *original* value of $v1.member$, to which $v1$ has lost visibility but to which neither $v2$ nor $v2.1$ have lost visibility. This value remains in $v1$'s original object representation on the heap, which is obscured to $v1$'s own view by means of its bitmap, which redirects reads and writes from abstract clients to $v1.member1$ onto the shared snapshot area. The outcome of this operation is as follows:

- Object instance $v1$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "here/shared" $\rightarrow$ (local)

[1] For clarity some steps were not discussed in the creation of v2 as they were not relevant.

- Object instance $v2$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$
- Object instance $v2.1$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$
- Object instance $v3$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "there/shared" $\rightarrow v1$

Now let us write to $v2.member1$, which requires redirecting to a non-shared write. The outcome of this operation is as follows:

- Object instance $v1$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "here/shared" $\rightarrow$ (local)
- Object instance $v2$:
  $member1$: "there/private" $\rightarrow$ Snapshot area
  $member2$: "there/shared" $\rightarrow v1$
- Object instance $v2.1$:
  $member1$: "there/shared" $\rightarrow v1$
  $member2$: "there/shared" $\rightarrow v1$
- Object instance $v3$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "there/shared" $\rightarrow v1$

It may be observed from the simplistic examples above that the root of any snapshot tree is a traditional oops object plus a read/write redirection bitmap. Subsequent snapshots are represented as an object header plus a redirection bitmap. Each object

in the snapshot tree may source its data from up to two places: the oops object at the top of the snapshot tree or from the snapshot area.

Let us pause for a moment as there are obvious arguments against differential in-memory snapshots using the object member as the block size. For instance,

1. **Saving memory is not worth the effort.** One may argue saving memory is not important in modern systems. A review of compressed oops as implemented within HotSpot demonstrates a presence of will to reduce memory consumption at the trade-off of significant complexity. The rationale provided is that while memory is cheap, bandwidth and cache are in "short supply" [23, § Compressed Ordinary Object Pointer]. While evaluation of this trade-off is beyond the scope of this paper, the effort to bring it into existence is notable.

2. **There won't be any real memory savings:** the block size of object members is too small. In storage snapshots, the block size is large relative to the pointer size, which provides the necessary economy for snapshots as the low cost of additional pointers is offset by eliding large data blocks that would otherwise be redundant. In HotSpot, object members are a sequence of references or primitives, both of which occupy relatively little memory. It would be a false economy to use two 32 or 64-bit pointers to "economize" an 8-bit "byte" member. Consequently, if differential snapshots are to be pursued some consideration should be applied to determine when to duplicate the member value, and when not to, based on the overhead of referencing relative to the projected savings. And even then, future analysis is needed to understand options and trade-offs.

3. **Performance will suffer.** This may be true due to poorer locality and additional indirection. Future research is needed to ascertain relative performance.

4. **Garbage could accumulate.** As member values within snapshots are updated, the approach described informally above will accumulate some garbage. Let us assume $v1$ has $member1$ and is snapped as $v2$. member1 is marked here/shared. If both $v1$ and $v2$ write to $member1$, the "here" value for $v1.member1$ is present but not in use. Further, if $v1$ is snapped again as $v3$, $v1$'s "there/private" pointer into the snapshot area is marked "there/shared." If both $v1$ and $v3$ write to $member1$, then the original "there/private" value is now garbage. Depending on the design of the snapshot area and garbage collection, it might be some time before the garbage value is collected.

Returning to the simplified example, removing non-root snapshots is trivial as no snapshot in the tree directly relies upon any non-root node value. Let us remove $v2$ from the middle of the tree. The outcome of this operation is as follows:

- Object instance $v1$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "here/shared" $\rightarrow$ (local)
- ~~Object instance $v2$:~~
  ~~$member1$: "there/private" $\rightarrow$ Snapshot area~~
  ~~$member2$: "there/shared" $\rightarrow$ $v1$~~
- Object instance $v2.1$:
  $member1$: "there/shared" $\rightarrow$ $v1$
  $member2$: "there/shared" $\rightarrow$ $v1$
- Object instance $v3$:
  $member1$: "there/shared" $\rightarrow$ Snapshot area
  $member2$: "there/shared" $\rightarrow$ $v1$

Removing the root node is possible when no dependent snapshots remain and is delegated to garbage collection via enhancement to include the root and snapshot area

pointers (if present) in the object graph. Within HotSpot, default garbage collection is generational and intuitively functions by walking the thread stacks and object graph to determine reachable oops instances. Found objects are moved to a survivor area and pointers updated. The previous area is subsequently re-used and the memory occupied by the inaccessible objects is re-used.

8.1.4 Simplify Snapshot Variable Load

The javac implementation described in Chapter 6 utilizes the 0xcb asnap bytecode to snap the object upon its first aload and subsequently emits an astore and aload to update the stack frame and push the snapped object reference back atop the operand stack as demonstrated by the disassembler output in Figure 8.1.

```
public T snap(T);
  descriptor: (Ledu/ecu/seng7000/ecumatt/ex2/SmallObject;)
     Ledu/ecu/seng7000/ecumatt/ex2/SmallObject;
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=1, locals=2, args_size=2
       0: 0x2b aload_1
       1: 0xcb asnap
       2: 0x4c astore_1
       3: 0x2b aload_1
       4: 0xb0 areturn
  LineNumberTable:
     line 103: 0
  Signature: #74                          // (TT;)TT;
```

Figure 8.1: Method w/asnap - javap output

It appears possible to combine the operation of these four steps into one asnap at the cost of creating the corresponding asnap_1, asnap_2, ... bytecodes and amending the base asnap to accept a stack frame location parameter (similar to base aload) from which it would retrieve and snap the object, return the value back to the stack

frame location, and leave the snapped reference atop the operand stack.

Intuitively, this would be a faster approach at the expense of generalization. Further work is required to determine the extent of any difference.

8.1.5 Type Exception

The JLS specification [13, § 8.4.1] adaptation described in Chapter 4 includes an operative exception for enum types that is not implemented in Chapter 6. This activity is left as future work as well as a mechanism for immutable and singleton types to exclude themselves from the snapshot process, only if consistent with design intent described in Chapter 4. An example of such a situation may be an immutable object where the state, as it cannot change, is effectively non-shared – there is effectively no value to snapping the object instance.

8.1.6 Bytecode Verification

As described in Chapter 6, the bytecode verification specified in Figure 6.2 is not implemented in Figure 6.21 and its implementation is left for future work.

8.1.7 Garbage Collection

The current minimal implementation can fail when garbage collection is triggered while the snap operation is running. This did not impede benchmarking as the benchmarking suite requests garbage collection before each snap operation, but this problem should be addressed as future work.

8.1.8 Supporting Tools

Supporting tools within OpenJDK not necessary to complete this research are not adapted and an attempt was not made to enumerate or explore this area. Tools that contain a switch statement based on bytecode that expect an entry for every known bytecode may be expected to fail when encountering 0xcb asnap. Similarly, tools that require an understanding of every known bytecode will likely behave in a similar manner. Enumerating and adapting the constellation of tools surrounding OpenJDK is beyond the scope of this research.

8.1.9 Escape Analysis Optimization

An escape analysis of formal parameters bearing the **__snap__** modifier may allow omission at runtime of the asnap/astore/aload pattern when the variable is fully captured within the method. While it is easy to claim a practitioner should not declare effort is needed to fix object state when that is not the case, **__snap__** is a declaration of intent and may be one element of a defensive strategy in the face of uncertainty or to protect against future adaptations to the calling code. One restriction with escape analysis within the current HotSpot implementation is that it is performed lazily [23, § Escape Analysis] and its output is not always available; hence, this optimization may not be attainable for all invocations. As future work, an empirical analysis of publicly-available source code may provide useful indicators regarding the optimization's potential degree of benefit.

8.1.10 JIT Compiler Support

As described in Chapter 6, the bytecode interpreter is adapted, but the c1 and c2 JIT compilers within HotSpot are not modified and were disabled in this paper

to force bytecode-interpreter only execution for evaluation in Chapter 7. For optimal performance and to ensure special command line parameters are not required, c1 and c2 JIT support is an item of future work.

8.1.11 Evaluate Predictable Behavior

In Chapter 7, predictable behavior is evaluated along narrow lines. It would be interesting to determine via a user study whether software engineers in practice find the __**snap**__ approach more or less predictable than the currently-standard approaches. This is suggested as future work.

8.2 Transformation Approach

A logical next step for transformation exploration involves incorporating a non-native mechanism such as the GSON [35] or cloning [36] libraries. These involve an adjustment of the snapshot depth guarantee in Chapter 4 but otherwise appear to warrant research. The trade off with these approaches is expected to be runtime performance, but the extent of the penalty is not known and the benefit is eliminating the need to adapt HotSpot and other JVMs to accommodate the additional bytecode.

8.2.1 Identity Relationship

The Java Language Specification defines the == identity operator as follows:

> At run time, the result of == is true if the operand values are both null or both refer to the same object or array; otherwise, the result is false [13, § 15.21.3].

Using a helper class to maintain a tree of snapshots in the transformation approach may provide the ability to identify multiple points in time of the same object. This

raises interesting questions:

1. Let mutable object type T be instantiated as A and let the state of A be modified using a mutator method giving A'. Did A's identity change due to modification?

   → *The answer to this, obviously, is 'no.'*

2. Let A be snapped as B. Do A and B have the same identity?

   → *The answer, as implemented in Chapter 4, 5, and 6, is obviously 'no'*

3. Should A and B have the same identity?

   → *Why or why not?*

4. Should A' and B have the same identity?

   → *Why or why not?*

5. What are the implications if A shared the same identity as A', and B as A?

To be clear, the present definition of identity is well-understood and modification thereof should not to be taken lightly. But it is a question similar to the Ship of Theseus: at what point does A stop being A? It is clear and accepted that $A == A'$ meaning mutation operations upon a mutable object do not alter identity. But if A and A' are the same identity and if B is a point-in-time of A, do both not meet the definition of identity equality specified in the Java Language Specification [13, § 15.21.3]?

In common practice an additional consideration seems apparent that is unclear from the Java Language Specification. While a particular object may change and mutate over time and retain its identity, only one object at any time may have that particular identity. Arguing whether that is appropriate or not in a world where the same object simultaneously exists in multiple states is beyond the scope of this paper.

Instead, it may be more useful to sidestep and consider a facility for reasoning about the identity relationship without upsetting the == apple cart. This may be accommodated by defining and transforming a new operator, perhaps @=, into a call to the snapshot helper object that returns true if the objects are == or both share the same snapshot root – and false otherwise.

Further consideration of this topic is suggested as future work.

8.2.2 Reasoning about Object State over Time

With multiple states of an object present simultaneously, it is feasible to determine which objects share a common ancestor state and navigate mutable object state similar to a tree. Further, recording the sequence of states selectively for specific objects at specific times or events may provide a foundation to check runtime conformance with a formal behavioral model. Given the programmatic nature of the functionality, it may be possible to trigger snapshots on state transitions to address problems of time scale [42, 43] encountered by other researchers.

8.3 Future Research Questions

As of this writing, the relative frequency of the existing methods – clone(), serialize, copy constructor, third party library – does not appear to be well-understood. Future work is suggested to analyze publicly-available repositories to determine and draw conclusions regarding the frequency of usage.

8.4 Summary

This area of mutable memory snapshots and runtime-managed unsharing of state by an intent declaration provides numerous accessible avenues for future researchers interested in providing better tools and solutions for software engineers. These solutions may have a positive impact on real-world development of large systems where problems of mutable object state sharing become most apparent.

Chapter 9

Conclusion

This chapter draws conclusions relative to the observations in **Chapter 7** regarding the experiment in **Chapter 6** and the hypothesis and prediction in **Chapter 1**. A summary is provided of the motivation, key research questions, and contribution.

9.1 Motivation

The motivation for this research is the premise Java should offer a universal and predictable manner to declare a need to unshare the state of a mutable object. For almost two decades software engineers have continued to use the same broken array of tools in Java or worked around the limitations with third-party libraries.

Similar to past efforts to improve Java, notably Pizza's parametric abstraction enhancements [12], Java's present limitations are not necessarily a prediction of its future. Offering an alternative path forward is the motivation for this research.

9.2 Hypothesis

The hypothesis of the research is described in **Chapter 1**: namely, that a universal Java mechanism is feasible that allows a method designer to universally declare when a method's actual parameter should be guaranteed to present non-shared state with predictable semantics.

9.3 Prediction

To test the hypothesis, **Chapter 1** predicted a mechanism to unshare state may be implemented in Java that is: universal (applies to all object instances), predictable (behavior is consistent across types), self-contained (does not require external libraries), and has reasonable performance (relative to existing methods).

9.4 Alternatives

After an overview of Java in **Chapter 2** and Snapshot concepts in **Chapter 3**, an abstract solution is proposed in **Chapter 4** with specific changes proposed to the Java Language Specification [13] in support of the proposal.

The basic idea is to add a new keyword, **__snap__**, as a formal method parameter modifier as shown in Figure 9.1. This modifier declares the method designer's intent to unshare the state of the actual method parameter, which involves transparently taking a point-in-time snapshot of the object before its use within the method body.

```
1 // Snap/unshare objInput state prior to iterating
2 public void addAll(__snap__ ArrayList<String> objInput) {
3     for(String str : objInput) {
4         this.add(str);
5     }
6 }
```

Figure 9.1: __snap__ example

Two alternative implementations were considered:

Chapter 5 discusses a transformation approach that encountered seemingly-fatal limitations that may have workarounds requiring the use of third-party libraries. This research is attempting to avoid OpenJDK externalities.

Chapter 6 describes a direct minimal implementation in OpenJDK's javac compiler and HotSpot JVM. The basic idea is to create a new Java bytecode, 0xcb asnap, which creates a point-in-time writable copy of an object and returns the point-in-time copy of the object in its place on the operand stack. This action is triggered by javac on the first load of a method parameter bearing the **__snap__** modifier keyword.

9.5 Evaluation

Chapter 7 evaluated the method proposed in **Chapter 6** against the research questions posed in **Chapter 1** and predicted could be true:

1. Is the operation universal?
2. Is the operation's outcome predictable?
3. Are third-party externalities required, or does the JDK stand on its own?
4. What is the performance relative to the extant native methods?

The observations from the evaluation in Table 9.1 indicate the Chapter 1 prediction is true and indicates the hypothesis is likely correct. A direct implementation of snapshots with the desired attributes appears feasible and is characterized by reasonable performance with improvements available (see Chapter 8).

Research Question	**Clone()**	**Snap**	**Serialize**	**Copy Constructor**
Predictable?	False	True	False	False
Universal?	False	True	False	False
Provided by the JDK?	True	True	True	False
Relative Performance?	1	2	4	3

Table 9.1: Summary of Evaluation Observations

9.6 Contribution

A novel mechanism is proposed in this research to enable a method designer to declare intent to unshare mutable object state and for the runtime to manage to that declared intent.

Modifications to the Java Language Specification [13] are proposed as a concrete example. These are set out in **Chapter 4** whereby method designers may clearly declare intent to unshare mutable object state using the **__snap__** modifier on a formal method parameter declaration. This proposal is followed by two alternative implementation strategies in **Chapters 5** and **6**.

An initial direct implementation in OpenJDK 10 is described more or less step-by-step in **Chapter 6** along with proposed modifications to the Java Virtual Machine Specification [15] centered around a new bytecode, 0xcb asnap. This bytecode is implemented in both javac and the HotSpot JVM's bytecode interpreter.

This paper may serve as a rough guide for future researchers undertaking similar modifications to OpenJDK. Within this paper these tasks are laid out step-by-step. The initial JDK toolchain setup is detailed in the OpenJDK repository [44]. Scant documentation exists relative to HotSpot internals aside from the actual sources, which bear few comments except in certain areas. Dissecting the code's operation required extended effort – usually on weekends or nights after work – to explore and determine how to adapt the codebase. Hopefully this paper will save some researchers time in the future..

Multiple paths for future work are proposed for researchers interested in providing simpler and improved facilities over what presently exists in Java and OpenJDK.

9.7 Future Work

To be clear, the implementation in **Chapter 6** is a minimal implementation and not yet suitable for insertion into the Java Language Specification, Java Virtual Machine Specification, and OpenJDK. The author has no illusions about this fact and details the future work remaining in **Chapter 8**:

1. Direct JDK Implementation Approach
    (a) Deep Snapshots
    (b) Platform Independence
    (c) Differential Snapshots
    (d) Simplify snapshot variable load
    (e) Type exception
    (f) Bytecode verification
    (g) Supporting tools
    (h) Escape Analysis Optimization
    (i) JIT Compiler Support
    (j) Evaluate Predictable Behavior
2. Transformation Approach
    (a) Identity Relationship
    (b) Reasoning re: object state over time
3. Evaluate clone(), serialize, and object constructor use

9.8 Conclusion

This paper sets out an experimental implementation and a plan for future work that aims to solve a common problems for software engineers – universally unsharing mutable object state without knowledge of the underlying concrete type. As part of this work, a novel intent-based mechanism, formal parameter modifier **__snap__**, is proposed for guaranteeing non-shared state of objects input into a software module.

The evaluation of the novel, intent-based **__snap__** method shows a promising approach to solve a long-lived and troublesome problem for software engineers in a simple and performant manner.

BIBLIOGRAPHY

[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] M. J. Steindorfer, "Efficient immutable collections," Ph.D. dissertation, University of Amsterdam, 2017.

[4] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

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

[6] "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

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

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

[9] "EsotericSoftware/kryo: Java binary serialization and cloning: fast, efficient, automatic," 2018, Accessed: 2018-09-22. [Online]. Available: https://github.com/EsotericSoftware/kryo#copyingcloning

[10] "kostaskougios/cloning: deep clone java objects," 2018, Accessed: 2018-09-22. [Online]. Available: https://github.com/kostaskougios/cloning

[11] "BeanUtils (Commons BeanUtils 1.8.3 API)," 2010, Accessed: 2018-09-22. [Online]. Available: https://commons.apache.org/proper/commons-beanutils/javadocs/v1.8.3/apidocs/org/apache/commons/beanutils/BeanUtils.html

[12] 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.

[13] 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

[14] "JEP 286: Local-Variable Type Inference," Accessed: 2018-09-16. [Online]. Available: http://openjdk.java.net/jeps/286

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, "The Java Virtual Machine Specification: Java SE 10 Edition," 2018, Accessed: 2018-09-16. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se10/html/index.html

[16] "JSR 335: Lambda Expressions for the Java Programming Language," Accessed: 2018-09-16. [Online]. Available: https://www.jcp.org/en/jsr/detail?id=335

[17] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java Language Specification: Java SE 7 Edition," 2015, Accessed: 2018-10-21. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se7/html/index.html

[18] "The Java Language Specification: Java SE 8 Edition," Accessed: 2018-09-22. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se7/html/index.html

[19] D. Erni and A. Kuhn, "The Hackers Guide to javac," *University of Bern, Bachelor's thesis, supplementary documentation*, 2008.

[20] H. McGhan and M. O'Connor, "PicoJava: a direct execution engine for Java bytecode," *Computer*, vol. 31, no. 10, pp. 22–30, Oct 1998.

[21] "JEP 318: Epsilon: A No-Op Garbage Collector," Accessed: 2018-09-16. [Online]. Available: http://openjdk.java.net/jeps/318

[22] M. Paleczny, C. Vick, and C. Click, "The Java Hotspot TM Server Compiler," in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, vol. 1, no. S 1, 2001.

[23] "5 Java HotSpot Virtual Machine Performance Enhancements," Accessed: 2018-10-13. [Online]. Available: https://docs.oracle.com/javase/10/vm/java-hotspot-virtual-machine-performance-enhancements.htm#JSJVM-GUID-3BB4C26F-6DE7-4299-9329-A3E02620D50A

[24] G. Duzy, "Match snaps to apps," *Storage, Special Issue on Managing the information that drives the enterprise*, pp. 46–52, 2005.

[25] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, R. N. Sidebotham *et al.*, "The Episode file system," in *Proceedings of the USENIX Winter 1992 Technical Conference*. San Fransisco, CA, USA, 1992, pp. 43–60.

[26] D. Hitz, J. Lau, and M. A. Malcolm, "File System Design for an NFS File Server Appliance," in *USENIX winter*, vol. 94, 1994.

[27] W. Xiao, Y. Liu, Q. Yang, J. Ren, and C. Xie, "Implementation and performance evaluation of two snapshot methods on iSCSI target storages," in *Proc. of NASA/IEEE Conference on Mass Storage Systems and Technologies*, 2006.

[28] W. Xiao, Q. Yang, J. Ren, C. Xie, and H. Li, "Design and analysis of block-level snapshots for data protection and recovery," *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1615–1625, 2009.

[29] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, 1983. [Online]. Available: https://doi.org/10.1145/289.291

[30] J. R. Larus and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 1, no. 1, pp. 1–226, 2007.

[31] X. Zhang, L. Peng, and L. Xie, "A Lightweight Snapshot-based Algorithm for Software Transactional Memory," in *Proceedings of the 9th International Conference for Young Computer Scientists, ICYCS 2008, Zhang Jia Jie, Hunan, China, November 18-21, 2008*, 2008, pp. 1254–1259. [Online]. Available: https://doi.org/10.1109/ICYCS.2008.180

[32] J. Li, H. Liu, L. Cui, B. Li, and T. Wo, "iROW: An Efficient Live Snapshot System for Virtual Machine Disk," in *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19, 2012*, 2012, pp. 376–383. [Online]. Available: https://doi.org/10.1109/ICPADS.2012.59

[33] A. Osuna and C. Carlane, *IBM System Storage N Series SnapMirror*. IBM, International Technical Support Organization, 2006.

[34] W. Xiao, Y. Liu, Q. Yang, J. Ren, and C. Xie, "Implementation and performance evaluation of two snapshot methods on iSCSI target storages," in *Proc. of NASA/IEEE Conference on Mass Storage Systems and Technologies*, 2006.

[35] "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

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

[37] "OpenJDK HotSpot Runtime Overview," Accessed: 2018-10-13. [Online]. Available: http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html

[38] J. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Escape Analysis for Java," in *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, 1999, pp. 1–19. [Online]. Available: https://doi.org/10.1145/320384.320386

[39] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: java benchmarking development and analysis," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 2006, pp. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[40] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, 2007, pp. 57–76. [Online]. Available: https://doi.org/10.1145/1297027.1297033

[41] V. Horký, P. Libic, A. Steinhauser, and P. Tuma, "DOs and DON'Ts of Conducting Performance Measurements in Java," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*, 2015, pp. 337–340. [Online]. Available: https://doi.org/10.1145/2668930.2688820

[42] P. Parizek, F. Plasil, and J. Kofron, "Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker," in *30th*

Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA, 2006, pp. 133–141. [Online]. Available: https://doi.org/10.1109/SEW.2006.23

[43] N. Rivierre, F. Horn, and F. D. Tran, "On monitoring concurrent systems with TLA: an example," in *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*, 2005, pp. 36–45. [Online]. Available: https://doi.org/10.1109/ACSD.2005.29

[44] "Building OpenJDK," Accessed: 2018-10-19. [Online]. Available: http://hg.openjdk.java.net/jdk10/jdk10/raw-file/tip/common/doc/building.html