Interacting with Dead Objects

Robin Salkeld  Gregor Kiczales

University of British Columbia, Vancouver, BC, Canada
{rsalkeld, gregor}@cs.ubc.ca

Abstract

Debugging and analyzing a snapshot of a crashed program’s memory is far more difficult than working with a live program, because debuggers can no longer execute code to help make sense of the program state. We present an architecture that supports the restricted execution of ordinary code starting from the snapshot, as if the dead objects within it had been restored, but without access to their original external environment. We demonstrate the feasibility of this approach via an implementation for Java that does not require a custom virtual machine, show that it performs competitively with live execution, and use it to diagnose an unresolved memory leak in a mature mainstream application.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Run-time environments

Keywords programming languages, meta-programming, reflection, runtime analysis, Java, virtualization

1. Introduction

When systems fail, sometimes the clues as to what happened are only found within the logs, event databases, stack traces, and various other remnants left behind by the system. Because it offers high fidelity, a snapshot of a process at the time of a failure is a nearly omnipresent feature, and for many modern languages this translates into a heap dump file that contains the state of every live object and its connections to other objects. A number of tools can parse this file and present the developer with an interactive, browsable tree of values. This interface is familiar and useful for developers, as it parallels how a debugger models the state of a live system.

Unlike live debugging, however, the developer cannot execute any of their own code in the context of the failure, which is a critical piece of functionality. The sheer size and complexity of the object graph makes navigating and understanding it without any higher-level, application-specific computation painful and time-consuming. Even obtaining a simple, human-readable representation of an object using utility code is not generally possible without a live process to evaluate on. Heap dump analysis tools attempt to imitate this feature, but require the application developers to re-implement the relevant application code using the meta-level interface for reading the heap dump.

To address this issue, this paper presents an architecture that enables additional execution on the objects, as if a live, debuggable process had been restored from the snapshot. This allows developers to invoke any code present in the original process, or even to load new analysis code into the emulated process, with no need for meta-programming. This functionality is implemented as an ordinary library, and does not require a custom language runtime. The execution is made sound by forbidding the recorded objects from accessing state external to the snapshot, since the original environment has been lost and cannot be accurately emulated. We hence refer to these objects as holographic objects: accurate recreations that cannot interact directly with the outside world.

The main contributions of this paper are:

• a summary of the shortcomings of analyzing raw machine state such as heap dumps via reflective APIs;
• an architecture for holographic objects, which enable restricted execution starting from the state captured in a heap dump;
• a description of how holographic execution must be restricted to ensure safety;
• evidence that this architecture can be efficiently implemented in a statically-typed language on an unmodified commodity language virtual machine;
• and evidence for the utility of holographic objects by using them to diagnose an unsolved memory leak in a mature mainstream application.

The implementation discussed here supports the Java programming language and runs on the Java Virtual Machine (JVM). The general approach based on emulating
language semantics, however, is applicable to other language runtimes as well. In fact, our implementation operates on JVM bytecode and is hence not limited to Java. Our implementation is available at https://github.com/robinsalkeld/retrospect/

2. Motivation

We first describe the approach many heap dump analysis tools use to work around the inability to execute code, which is to force developers to program analysis using the reflective heap dump model, and examine the problems we intend to solve with our architecture.

Consider a Java developer browsing the object graph in a heap dump, who has identified an object that implements the built-in java.util.Map interface. This implies that the object implements an associative mapping from keys to values. Suppose the developer needs to know what value a certain key is mapped to. When debugging a live process this is as simple as evaluating the expression `map.get(key)`, but the task is surprisingly involved with a dead process. Even if the concrete type of the map was a well-known class such as java.util.HashMap, an implementation backed by a straightforward hash table, finding the right key-value pair involves manually searching through the internal representation of a potentially massive hash table to find the matching entry.

Not being able to execute code also means there is also no way to invoke the method `Object.toString`. This method is defined at the top of the Java class hierarchy with a default implementation and therefore callable on any object, and is used to create human-readable text describing the object. The Eclipse debugging perspective, for example, includes a window dedicated to displaying the result of calling `toString` on objects in other windows, and it is exactly this basic functionality that is missing in a heap dump browser.

The tools used to interact with heap dumps attempt to address this issue by providing utilities to automate these tasks. The Eclipse Memory Analyzer Tool (MAT) [2], which we use to represent the current state-of-the-art, includes several dozen actions for navigating the object graph, and “Extract Hash Entries” is one such utility. To implement this utility, however, the tool developer must include handler code for every single subclass of java.util.Map that might be encountered in the object graph.

Since the tool cannot possibly handle all possible types in arbitrary application code, the utility must be pluggable so that developers wishing to analyze their heap dumps can contribute handlers for the types of objects in them. Ultimately this means that those developers are faced with the job of re-implementing the semantics of a non-trivial portion of application code using the meta-level represented by the heap dump model.

Listing 1. An example of working with an associative mapping using application code.

```
public String getProperty(
    ServiceProperties map, String key) {
    return (String)map.get(key);
}
```

Listing 2. The same example as in Listing 1 but using the meta-level interface of a heap dump model.

```
public String getPropertyMeta(
    IObject map, String key) {
    String[] keys = null;
    String[] values = null;

    IObject object = map.resolveValue("headers");
    IObjectArray array = (IObjectArray)headers;
    if (array != null) {
        long[] keyAddrs = array.getReferenceArray();
        if (keyAddrs != null) {
            // Call helper method to dereference
            // addresses to String objects
            keys = getServiceProperties(keyAddrs);
        }
    }
    /* Similarly for "values" field */
    if (keys == null || values == null)
        return null;
    for (int i = 0; i < keys.length; i++) {
        if (keys[i].equals(key)) {
            return values[i];
        }
    }
    return null;
}
```

Listing 1 contains an example of base code that retrieves the value a key is mapped to in one particular type of map, and Listing 2 contains an example of MAT source code that achieves same thing using the reflective heap dump API. This snippet is part of a large submodule responsible for reconstructing the state of the Equinox OSGi Java module system [9] from a heap dump; this code is only concerned with the configuration properties, a mapping from strings to strings, for a particular service. The configuration is stored as a pair of parallel arrays, and this code first resolves the object references contained in each, then iterates through the list of keys until it finds a match. This example illustrates a number of disadvantages inherent to this approach.

Complex: The complexity inherent in reproducing the behaviour of a polymorphic method such as Map.get is high. Meta-level code is inevitably much more verbose than the equivalent base-level code, and reproducing higher-level

---

1 The standard library alone includes several dozen, such as ConcurrentHashMap, LinkedHashMap, TreeMap, and so on.
language features such as polymorphic method dispatch and field shadowing correctly at this level is challenging.

**Type-unsafe:** If the analysis code makes a type error, it will not cause an error until further downstream reads fail to find the fields they are expecting, if an error occurs at all. This makes tracking down coding errors much more difficult. In Listing 2, the meta-level code assumes that the property values are strings when in fact they can be other types, and in the base-level code a cast is necessary to make the code compile. Guarding against such errors through explicit meta-level type checks requires programmer discipline, contributes greatly to code bulk, and as shown in the example are often omitted.

**Brittle:** In Listing 2, the code expects the properties to be stored in two parallel arrays. In more recent versions of Equinox, however, these fields no longer exist, and the properties are instead stored in a single special-purpose ServiceProperties object. Because of the various null checks in this code, likely present to prevent this exact problem from crashing the overall analysis, no error is raised, and the service object appears to have no properties. The fact that meta-level analysis accesses datatype internals directly means that the analysis can very easily break, or even worse silently produce incorrect results, when run against a dump produced by a newer version of the target code. Since this approach involves a second, redundant implementation of the functionality in the application, it introduces an ongoing maintenance burden.

**Insecure:** The access control mechanisms built into a language, such as declaring types or fields private or protected in Java, are generally more easily circumvented within its own reflection facilities. They are lost entirely, however, when traversing a heap dump model; in the example above the code reads from two private fields with no extra difficulty. This contributes towards the brittleness of analysis code as above, since it is easy to refer to internal details that are likely to change in the future, but it also means that sensitive data is easy to read, whether by accident or on purpose.

**Unfamiliar:** Developers familiar with base-level (ordinary) programming in a language must learn an additional paradigm to understand and effectively work with the meta-level object model. In addition, because application values frequently refer to datatypes from libraries, reimplementing application-level code requires understanding and reimplementing library code as well, since the encapsulation that normally hides those details from clients has been lost. Tool developers can alleviate some of this burden by handling the most common functionality, but handling even a small percentage of the code a user is likely to encounter is impractical.

Most debugging and analysis tools that are capable of reading snapshots or core dumps share these limitations. The GNU Project Debugger (GDB), for example, supports automated debugging through Python scripts, but these scripts operate on a similar meta-level representation of program state.

### 3. Holographic Virtual Machines

Ultimately all of these difficulties would be resolved if a heap dump analysis tool could execute code on the objects in the dead process, as if the execution occurred on the live process immediately after the snapshot was taken. We aim to provide this functionality through holographic objects, which are virtual objects that reflect the state and behaviour of the dead objects recorded in the snapshot. This section describes the high-level architecture we have used to make this possible.

To implement holographic objects, we require a reflective API for accessing the state of the dead objects, and an execution environment that implements the semantics of normal execution with respect to the reflective API instead of in-memory native objects. For example, an instruction that accesses a field of an object should have the effect of accessing that field from the holographic object in the heap dump via our reflective API.

Holographic objects should behave like the dead objects they imitate, or else any analysis performed on them may produce incorrect results. Any code in the control flow of holographic execution that cannot be exactly reproduced based on the information in the heap dump must result in an explicit exception. This includes any attempt to access or mutate the external environment of the original process, such as other processes, the file system, the network, and so on. This also implies that holographic objects must be completely sandboxed: it must be impossible for them to obtain references to any objects in the host VM, or vice versa. Holographic objects are hence encapsulated inside a holographic virtual machine. Values may only be passed between the guest and host VMs using explicit reflective methods, and only primitive values are permitted to avoid leaking references. Figure 1 contrasts an ordinary VM with a holographic VM running inside another ordinary VM.

#### 3.1 Mirrors

To support creating holographic objects on top of multiple snapshot formats, or indeed to other sources of object state, we define our reflective API using an independent set of reflective interfaces. The core functionality of the reflective access we need is apparent if we compare the heap dump model to existing reflective APIs. The Java platform includes two such APIs: a set of built-in reflective methods such as Class.getFields, and the Java Debugging Interface (JDI) provided by the Java Platform Debugger Architecture [16], on which remote Java debuggers are built. Each API provides similar functionality backed by different state: the built-in Java reflection methods reflect on the state of the current

---

2 In the case of the JVM, String values are permitted despite being objects, as they are a core type whose implementation must be immutable.
VM, the JDI reflects on the state of a separate VM being debugged, and the heap dump model reflects on the past instantaneous state of another VM.

Bracha and Ungar \cite{Bracha2001} label such pluggable, independent reflection interfaces as mirrors, and we adopt their terminology here. We define a central interface named VirtualMachineMirror, which encapsulates an entire object graph, including all loaded classes and hence all executable code in the system. Other interfaces represent objects, classes, fields, methods, arrays, and so on.

A holographic VM is then represented as a VirtualMachineHolograph wrapper that refers to an underlying VirtualMachineMirror instance and implements that interface itself. This achieves our goal of making holographic objects a general-purpose library, as the wrapper can be applied to any representation of VM state that can be used to implement the generic mirrors API. The library provides similar holographic wrappers for the other related mirror interfaces. Most importantly, the holographic implementation of MethodMirror.invoke does not delegate to the wrapped method mirror, but instead emulates the semantics of that method’s definition as described above.

### 3.2 Mutations

Although holographic execution is not allowed to read from or write to the state of the outside world, many useful expressions that are semantically functional will have internal side-effects that attempt to modify the internal object graph. For example, looking up a string key in a HashMap as described earlier requires calculating the string’s hash code, and the implementation of string hashing caches the result in an instance field of the string the first time it is calculated. This means if the method was not previously called on a string in the original VM, or if the string was newly-created as part of holographic execution, the method will attempt to set a new value on the mirror. The heap dump model is read-only, as many potential mirror implementations will be, and hence this will trigger an exception.

To support mutation in holographic execution, the holographic adapters superimpose a mutable mirror graph over the wrapped, potentially immutable graph. Each wrapper is initially empty, exposing state identical to that of the wrapped object. As values are written to the holographic objects they are stored in the wrapping object, and future reads will return those values. The same approach applies for other more subtle side-effects on the mirrors API, such as expanding the set of classes loaded by a class loader by defining a new class.

Maintaining mutations on the object graph independently in this way also offers flexibility in the semantics of multiple successive sessions of holographic execution. If the same holographic VM instance is used for each, the side-effects of prior executions will potentially affect future executions. This is consistent with a developer’s experience when evaluating expressions in a normal debugging client. Alternatively, a new holographic VM can be instantiated for each evaluation, to restore the same pristine initial snapshot state. This is equivalent to experimenting with a live process by repeatedly forking a new sandboxed process that can be perturbed in arbitrary ways and then discarded.
public class Employee {
    private int age;

    public static Set<Employee> over40(Employee[] input) {
        Set<Employee> result = new HashSet<Employee>();
        for (Employee e : input) {
            if (e.age > 40) {
                result.add(e);
            }
        }
        return result;
    }
}

public class hologram/Employee
    extends ObjectHologram {
    // Inherited from ObjectHologram:
    // public LObjectMirror; mirror
    public final static LClassMirror; classMirror

    public static over40(Lhologramarray1/Employee;)
        Lhologram/Set;
    L0
    LINENUMBER 17 L0
    NEW hologram/HashSet
    DUP
    1)| INVOKEESPECIAL HashSet.<init> ()V
    ASTORE 1
    L1
    ALOAD 0
    DUP
    ASTORE 5
    2)| INVOKEINTERFACE ArrayMirror.length ()I
    ISTORE 4
    ICONST_0
    ISTORE 3
    GOTO L2

    L3
    ALOAD 5
    ILOAD 3
    3)| INVOKESTATIC ObjectArrayHologram.getHologram
        (LObjectArrayMirror;I)LHologram;
    ILOAD 4
    LDC "age"
    4)| GETSTATIC hologram/Employee.classMirror
        : LClassMirror;
    LDC InstanceHologram.getIntField
        (LHologram;LClassMirror;LString;)I
    BIPUSH 40
    IF_ICMPLT L5
    L6
    ALOAD 1
    ALOAD 2
    5)| INVOKEINTERFACE hologram/Set.add
        (LHologram;)Z
    POP
    L5
    IINC 3 1
    L2
    ILOAD 3
    ILOAD 4
    IF_ICMPLT L3
    L7
    ALOAD 1
    ARETURN
}

Figure 2. A example of how JVM bytecode is translated into hologram bytecode, which operates on mirror classes and interfaces. Modified instructions are numbered in both the original and translated bytecode listings. Only object and array instructions are modified; control flow and primitive instructions are left untouched. Note that downcasts such as the one in 3) are often necessary to ensure type safety.
3.3 Translating Code

The most obvious approach to implementing holographic execution would be customizing a language runtime to support it, but this would be non-portable. We instead chose the implementation strategy of translating programs into lifted versions equivalent to holographic object semantics. Each instance of a core language operation is mapped to methods of the mirrors API, and the implementors of those interfaces thus determine the runtime behaviour of the language.

Our particular implementation targets the JVM, and hence the source language it translates is JVM bytecode, using the ASM bytecode processing framework [6]. We label the translated versions as hologram classes. These classes are encapsulated within the holographic VM API and never directly exposed to developers or tools using holographic objects. All instance field declarations in the original classes are replaced with a single ObjectMirror instance field, and the individual bytecode instructions are lifted to operate on those instances. The transformations are all local and context free, although a single instruction will frequently be translated into multiple instructions. Figure 2 contains a small example of how bytecode is transformed.

The semantics of holographic execution imply that holographic object references require two orthogonal dimensions of polymorphism: the original class hierarchy for virtual and overloaded method invocations, and the virtual mirror interface methods for object state access. We have chosen to map the original hierarchy into an isomorphic hierarchy of hologram types which preserve the subtype relation, allowing method invocation to operate as in the original bytecode. Our techniques are similar in several respects to those used by Factor et al. [8] to transparently rename classes in order to support instrumentation of core Java classes. Note that another valid approach would be to replace object references with direct mirror references and implement dynamic method dispatch manually instead. This could be evaluated in future work, but we suspect that the overhead of handling method dispatch is likely equal to or worse than the overhead of wrapping mirrors with hologram classes instances.

Each source type is usually mapped to exactly one internal type, but in some cases maintaining the subtyping relationship in user-level code requires splitting the type into a concrete class and an interface. The mapping function between hierarchies is therefore actually defined by two functions: HC, which is guaranteed to be a concrete, instantiable class, and HT, which may be an interface. For all types C and D in the original bytecode, we have:

\[ HC(C) <: HT(C) \]
\[ C <: D \]
\[ HT(C) <: HT(D) \]

In general, \( HC(C) \) is used wherever new instances of \( C \) are created, or when \( C \) is used as the superclass of another class, whereas \( HT(C) \) is used wherever \( C \) is used as a reference type for local variable, method parameters, and so on. The cases where the two functions differ are outlined below.

**Interfaces:** Object is both the base class of all concrete classes and the top of the subtyping lattice, and hence a supertype of interfaces as well. References of type Object are mapped to a Hologram interface, which all hologram classes and interfaces implement and extend, and which has a single getMirror() method. Where Object appears as a superclass, however, an ObjectHologram class is used instead, which actually declares the mirror field and implements getMirror().

**Arrays:** Each distinct array type, which is normally created automatically in the JVM without requiring explicit class definitions in source, is mapped to a distinct class type; although there is no virtual method dispatch on arrays, array types can still create valid method overrides when used as parameter types. These must also be split, since they must be concrete and instantiable but also support multiple inheritance because of covariance, meaning that for each interface A that B implements, the hologram type for B[] must be a subtype of the hologram type for A[].

If \( T \) is an array type with reference element type \( E \) and \( n \) dimensions (i.e. the type \( E[[...[n]...]] \)), we use \( HAC(E, n) \) and \( HAT(E, n) \) to refer to \( HC(T) \) and \( HT(T) \), which will be a class and an interface, respectively. The extends and implements clauses for these types are defined as follows:

\[ HAC(E, n) \text{ implements } HAT(E, n) \]
\[ E \text{ extends } C \]
\[ HAT(E, n) \text{ implements } HAT(C, n) \]
\[ E \text{ implements } I \]
\[ HAT(E, n) \text{ implements } HAT(I, n) \]
\[ n > 0 \]
\[ HAT(\text{Object}, n) \text{ implements } HAT(\text{Object}, n - 1) \]

The last implication above arises from the combination of array subtyping covariance and the fact that \( \text{Object[]} <: \text{Object} \). Note that \( HAT(\text{Object}, 0) \) is simply \text{hologramType}(\text{Object}), which is the Hologram interface.

Since the results of translating bytecode will be the same for successive holographic VMs on the same heap dump, our system caches translated bytecode on disk to improve performance. The cache is isomorphic to a fast associative mapping keyed by class name with separate sequential chaining to handle multiple classes with the same name. This approach is effective since the name of a class is by far its most specific characteristic, but still handles multiple classes with the same name occurring in a single VM.

Holographic JVMs also provide an optional prepare operation that iterates through all currently loaded classes and
eagerly generates the translated bytecode for each, which will pre-populate the cache. This will often be the preferred workflow: a holographic JVM could be prepared in advance and the cached bytecode distributed along with the heap dump.

4. Scope
There are several obstacles that may prevent holographic execution from emulating live execution soundly. All are direct results of missing information in the snapshot, although in most cases these can be solved by additional configuration provided by the user. This section outlines the factors that limit the completeness of this technique and the extent to which we are able to overcome them in our implementation.

4.1 Missing Bytecode
The most immediate obstacle to holographic execution on the JVM is the fact that heap dumps generally do not contain any bytecode, as most JVM implementations maintain class definitions in a separate area of memory. We must somehow recover the definitions of the classes in the heap dump in order to execute any code. Class definition on the JVM is dynamic: the core ClassLoader class and its subclasses are used to dynamically locate the bytecode for a requested class name at runtime. The implementation of these class loaders can be arbitrarily complex and is often non-trivial in popular application containers such as OSGi, and so providing the missing bytecode in a holographic VM though manual configuration is not feasible.

Our solution is to leverage the fact that nearly all class loaders eventually load bytecode from a class file on the file system, and more specifically one that matches the requested class name. We use holographic execution itself to call the appropriate method on the class’ loader to read the contents of the matching class file. This approach is valid for the vast majority of Java code, but for full generality this piece of the architecture is pluggable so that more unusual class loaders eventually load bytecode from a class file on the file system. However, the configuration of a holographic VM includes such implementations for the most commonly encountered native methods (MNMs), and includes such implementations for the most commonly encountered native methods in the JRE.

In most cases the implementations of MNMs can be quite naive and unoptimized. In the context of supporting post-hoc debugging and analysis the raw efficiency of the implementation is not the primary concern, so long as the methods semantics can be accurately reproduced. See Section 5.2 for a discussion of the efficiency of our architecture.

Native methods can be left unimplemented, or they can be expressly marked as forbidden because their semantics require accessing their external environment. In either case the unsupported native method is replaced with an MNM stub that throws an exception. This means that classes with unimplemented or forbidden native methods can still be loaded and used in holographic execution, so long as those native methods are not actually called. This is critical since the classes in the JRE include over 1000 native methods, many of which involve some form of input from or output to the external environment.

Note that application classes outside of the standard language runtime can also include native methods, and so if a developer wishes to execute holograph code that will hit
those methods they must provide the required MNMs themselves. Native methods are much less prevalent in application code than in the core JRE, however, and the burden of providing these alternate implementations is far less than the burden of re-implementing everyday code as in the reflection-based analysis approach.

### 4.3 Class Initialization

Class initialization occurs in Java when a class is first used, and involves invoking a special static method in the class’ bytecode called an initializer. This can have arbitrary effects on the object graph, and holographic execution must preserve this behaviour by invoking the initializer of any uninitialized class before accessing it.

Like a class’ bytecode, however, the initialization flag is not present in most heap dumps, so there is no direct way to tell if a class was defined but not yet initialized at the time of the dump. Since failing to initialize an uninitialized class can lead to inconsistent, unsound errors, holographic execution must raise an exception if it attempts to load a class whose initialization status is indeterminate.

We observe that in almost all cases the initialization status of a class can be automatically inferred from other data, based on the rules for when initialization must occur. Before class initialization, every non-constant static field has a default value: false for boolean fields, null for object references, and so on. Setting a value on a static field forces initialization, so a non-constant static field in the heap dump with a non-default value implies that the class must be initialized. Conversely, if the execution of a class initializer has the definite effect of setting a non-default value on a field but that field has the default value in the heap dump, the class must not be initialized.

In addition, given the definition of each class’ initializer, we can define a preordering $A \triangleright B$ to mean “the initialization of class $A$ forces the initialization of class $B$.” If we use $\text{initialized}(A)$ to symbolize that class $A$ is initialized, we have two additional judgements we can use to infer whether a class is initialized:

$$\frac{\text{initialized}(A) \land A \triangleright B}{\text{initialized}(B)}$$  
$$\frac{\neg\text{initialized}(B) \land A \triangleright B}{\neg\text{initialized}(A)}$$

To take advantage of these rules, when a class initializer method is encountered while translating bytecode, the holographic VM architecture performs a conservative analysis of the effects of the method. We use an abstract interpretation [17] similar to the type inferencing algorithm used by JVMs to verify bytecode, where the abstract values are three-valued booleans indicating whether a value is a default value, is not, or could be either. The output of this data-flow analysis is both a three-valued boolean for each static field and the set of classes the method’s execution is guaranteed to force the initialization of. When it is necessary to check if a holographic class $A$ is initialized, the class’ static field values are compared with the static analysis results as described above. All classes $B$ for which $A \triangleright B$ are also checked recursively, and if any are definitely uninitialized $A$ is determined to be uninitialized as well.

This analysis is sound but not complete: classes may still be encountered for which the rules above are not enough to infer whether it is initialized. We further observe, however, that many class initializers are idempotent, in that they may be executed more than once without any additional side-effects. This means they can safely be run on classes that may already be initialized. The architecture thus includes another pluggable mechanism for users to mark specific classes as having idempotent class initialization. In Section [23] we provide evidence that this necessity should be relatively rare. It is also possible for a class to have non-inferable initialization status and a non-idempotent initializer, but we have yet to encounter such a case.

### 4.4 Concurrency

Our holographic VM implementation is currently limited to single-threaded execution, but there are no assumptions in the architecture that would prevent concurrent holographic execution. Like the JDI model, executing code in a holographic VM happens in the context of a specific thread mirror from the heap dump, and uses a dedicated native thread in the host VM to execute the translated bytecode. The semantics are identical to invoking a method on a paused thread while debugging a live process.

In order to support multiple native threads simulating multiple holographic thread executions, the data structures used in the mutable object graph layer described in Section [32] simply need to be replaced with their appropriately synchronized equivalents: replacing HashMap instances with ConcurrentHashMap instances, for example. The synchronization overhead will have a negative impact on performance, which should be the subject of future evaluations, but this will enable more complex post-hoc application simulation.

## 5. Evaluation

This section evaluates two primary research questions:

1. To what extent does the holographic VM architecture improve on the reflection-based approach to heap dump analysis?
2. Is holographic execution responsive enough for a typical heap dump analysis scenario?

### 5.1 Case Study: Diagnosing a Memory Leak

To evaluate the feasibility and utility of object holographs, we augmented the Eclipse MAT to leverage them as much as possible, and then used the modified tool to diagnose a real world memory leak contributed by an end user.
5.1.1 Extending the Eclipse MAT

A large portion of the Eclipse MAT user interface centres around navigating and summarizing the object graph through predefined parameterized queries, some of which are directly analogous to source-level operations; “Extract List Values,” for example, iterates through a list’s entries in the same way as list iterator objects do. Our primary augmentation of the tool was to define two additional generic queries whose implementation used holographic execution.

The first is “Evaluate Expression,” which parses and evaluates a given code snippet in the context of the each object selected in the tool. This is accomplished by adapting a holographic VM to the JDI and reusing the implementation of the Eclipse debugging UI. It supports either evaluating the expression once for each selected object or collecting all selected objects into a single Collection through a boolean-valued “aggregate” parameter.

The second is “Load and Run Code,” which evaluates the contents of a specified method from a given class file on disk. This is accomplished by using holographic execution to create a new class loader instance, pass the class bytecode into the appropriate method to make the class loader define the new class, and then actually invoke the target method. This allows users to define more complicated queries via additional code compiled against the original application binaries. This query also supports the same “aggregate” parameter.

We also replaced several existing queries with equivalent versions that used holographic execution. The “Extract List Values” query, for example, was reimplemented to invoke the iterator() method on any collection and then use the result to iterate over the collection’s elements. This not only increased the generality of the resulting queries, but also enabled them to accept newly created holographic objects as well as existing heap dump objects. Replacing these reflection-based query implementations also eliminated thousands of lines of code, showing that holographic execution also simplifies tool development.

5.1.2 Debugging Experience

The Juno release of the Eclipse C and C++ Development Tools (CDT) contained a memory leak indexing a large project caused the Eclipse runtime to exhaust all available memory, where the same project was successfully indexed in previous versions of the CDT. The user reporting the bug was able to upload a 1 gigabyte heap dump from the time of failure, but because the project that caused the error contained proprietary code they were not allowed to provide the actual project source. This hindered attempts by the CDT contributors in the following months to reproduce the problem, despite multiple other users reporting the same bug.

https://bugs.eclipse.org/bugs/show_bug.cgi?id=400073

Listing 3. Analysis code used to diagnose the Eclipse CDT memory leak bug

```java
public static Map<String, Integer> findDuplicates(Collection<CPPASTName> names) {
    SortedMap<String, Integer> counts = new TreeMap<String, Integer>();
    for (CPPASTName n : names) {
        String nKey = n + " - " + Arrays.toString(n.getNodeLocations());
        Integer count = counts.get(nKey);
        if (count == null) {
            count = 0;
        }
        counts.put(nKey, count + 1);
    }
    return counts;
}
```

The CDT contributors were able to determine that approximately 80% of the heap was retained by over 1.8 million instances of the class CPPASTName and their related child objects. This class is used to represent unique occurrences of symbols in C++ source code after preprocessing, and the bug reporter’s estimate of the actual number of such symbols in their code was smaller by a factor of six. Our initial theory for the memory leak was that the indexing process was creating multiple duplicate name objects representing the same locations in the source code. A straightforward way to investigate this theory is to iterate over all of the name objects and group them by their locations, in order to detect multiple names from the same location.

Obtaining the necessary bytecode for the relevant classes in the uploaded heap dump was not difficult in this case: we only required the appropriate versions of the Java 6 JRE and the Juno Eclipse distribution. We then authored a small helper method, built against the matching version of the CDT source code, which iterates over a sequence of CPPASTName objects and populates a map keyed by a string representation of their locations. See Figure 3 for the relevant source code. This analysis would be very time-consuming to implement using reflection: although the CPPASTName class has a field for storing its location, it is lazily calculated on request using several related datatypes, and so for the majority of the objects in the heap dump this field contains null.

We executed this code using the “Load and Run Code” query described above on the first 100,000 CPPASTName objects in the heap dump, resulting in a new holographic HashMap object. To examine its contents in the Eclipse MAT UI, we executed a holographic query to extract the key and value pairs from any Map instance. The results confirmed our theory that there were many sets of duplicates,
in many cases over a dozen symbols with the same name and location.

Given that many of the most duplicated symbols were from a common library, our next theory was that the indexer was creating a separate symbol instance every time a header file was included. We selected one of the most duplicated symbols and began to test this new theory by writing code to print out the path of include declarations for each. The first step was traversing the parse tree to find the compilation unit containing each name, which we achieved using the "Evaluate Expression" query on the string "getTranslationUnit()".

We were surprised to find that each symbol came from separate compilation units. Executing "getFilePath()" on each revealed that they were all for the same source file. From this point it was relatively simple to use existing MAT queries to find the references keeping the extra parse trees from being collected by the garbage collector, in particular a thread local that was not cleared after use. This analysis was presented on the online bug report, and a fix was submitted shortly after by one of the project contributors.

5.2 Performance
To determine whether holographic execution is performant enough for its intended use, we created a test harness that iterates through every object in a VirtualMachineMirror and executes the toString method on each one, measuring the time taken to return from the invocation. This benchmark was chosen because it is easy to implement and applicable to any Java codebase, and yet exercises a surprising amount of code; even very simple implementations of toString are often only the tip of the iceberg when all of the methods that are ultimately used in their control flow are included.

We ran our benchmark against three sample applications. jre only is a stub application including only an empty main class, for the purpose of benchmarking only the contents of the JRE. tomcat is the Apache Tomcat web server, version 7.0.37, after serving the initial welcome page. eclipse is the Eclipse IDE, build 20130614-0229, with a minimum of plugins installed in order to keep the total class and object count manageable.

For each sample application, we used the JDI to connect to and pause the live process, captured a single heap dump, and then ran the benchmark against both the live process and the snapshot. We used the performance of remote execution on a live process as the baseline, measuring the performance of holographic execution as a kind of overhead compared to this baseline. These experiments were performed on a MacBook Pro laptop with a 2.4 GHz Intel Core 2 Duo CPU and 8 GB of RAM, running Mac OS X version 10.7.4. Figure 3 presents our results.

Since there is no convenient method for uniquely identifying objects in Java, and hence no convenient way to correlate the object mirrors in the two different VM mirrors, the data are analyzed as two independent sets. It would be instructive in future work to develop an algorithm for matching objects between VMs, possibly using structural comparison or raw memory addresses, in order to match times and analyze the average overhead.

We were surprised to discover that holographic execution is actually faster in all cases than remote execution using the JDI. This is due to the fact that the JDI relies on interprocess communication to pass values between the target and source VMs, whereas a holographic VM’s state resides in memory with the caller, and for relatively simple objects this invocation overhead is greater than the time needed for the execution itself. A future evaluation could connect the JDI to a remote holographic VM instead of a local one to normalize this difference, but this would require additional engineering to accomplish. In addition, since a holographic VM does not have to reside in a separate process, the lower latency of reflective calls is in fact observed in tooling, although traded off by the overhead of keeping the object graph in memory locally.

The most expensive aspects of holograph execution are fetching the bytecode for the original classes as described in Section 4.1 and translating that bytecode to produce hologram classes as described in Section 3.3. When the extra step of loading a heap dump and preparing the holographic VM are included, the total times to run the benchmark on either a live or dead process are similar. Since the results of this process for each class in the heap dump are cached on disk, however, successive analysis runs can avoid this processing time. Our experience shows that the translation time is consistently about 5 seconds per megabyte of class file content. Preprocessing the entire Java Runtime Environment (JRE), which consists of over 20,000 classes and over 60 MB of bytecode, can be done in just under five minutes.

5.3 Completeness
The major limitation on completeness in this system is the possibility that the execution of useful code could encounter unsupported or illegal native methods. Our experimentation has initially targeted the Mac OS X distribution of Java 7 release 5, and for every native method in that JVM’s runtime we have either provided a mirror-based alternate implementation or explicitly determined that it requires illegal access to the external environment and marked it as forbidden. Appendix A lists our categorization of these native methods.

Our implementation currently includes 99 alternative implementations of native methods, with a total of 1443 source lines of code. This serves as a rough evaluation of the effort involved in supporting a particular VM implementation. Only those methods in the sun.misc.Unsafe class are specific to the exact JVM implementation we used, as they involve raw memory addressing that depends on the exact memory layout of its objects. There are also a handful of platform-specific classes such as UNIXFileSystem that contain native methods, but this is only a superficial platform dependency since the actual MNMs for such methods are only trivially different. 187 legal native methods are not yet implemented.
<table>
<thead>
<tr>
<th></th>
<th>Application</th>
<th>jre_only</th>
<th>tomcat</th>
<th>eclipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>456</td>
<td>2657</td>
<td>7610</td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>2249</td>
<td>46387</td>
<td>99452</td>
<td></td>
</tr>
<tr>
<td>Live VM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg. toString time (ms)</td>
<td>15.9</td>
<td>25.9</td>
<td>33.2</td>
<td></td>
</tr>
<tr>
<td>Max. toString time (ms)</td>
<td>1748</td>
<td>22041</td>
<td>80234</td>
<td></td>
</tr>
<tr>
<td>Std. Dev. toString time (ms)</td>
<td>74.4</td>
<td>279.3</td>
<td>512.6</td>
<td></td>
</tr>
<tr>
<td>Holographic VM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prepare time (s)</td>
<td>44</td>
<td>171</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>Avg. toString time (ms)</td>
<td>5.4</td>
<td>2.5</td>
<td>7.4</td>
<td></td>
</tr>
<tr>
<td>Max. toString time (ms)</td>
<td>1279</td>
<td>8804</td>
<td>55867</td>
<td></td>
</tr>
<tr>
<td>Std. Dev. toString time (ms)</td>
<td>38.7</td>
<td>78.7</td>
<td>325.6</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.** Results of executing Object.toString on every object in a VM, comparing performance on a holographic VM versus a live VM via the Java Debugging Interface. Although the time to translate the bytecode for all classes in the VMs is significant, once loaded the local holographic VM actually executes these methods faster than the remote process. In all cases the minimum toString() time was 0ms, as several classes define the method to simply return a constant or recalculated value, and hence return essentially instantaneously.

In our prototype, as they were not encountered in our experiments, but these are all either trivial variations on other implemented methods, such as alternatives for different primitive types, or alternative ways of accessing the reflective properties already supported by the mirrors API.

The other limitation on completeness is the possibility of encountering classes in the heap dump with indeterminate initialization status. In the process of supporting our experiments we encountered nine such classes. We explicitly marked these as having class initializers that were safe to re-run, implying they could safely be executed even if they may have already run; these are listed in Appendix B.

6. **Related Work**

6.1 **Mirror-based Behavioral Interception**

Prior work has examined the idea of customizing the behaviour of a language’s objects via implicit mirror implementations. Such objects have been called *mirages* [15] or *virtual values* [4], and have largely been studied in the context of behavioural intercession, or augmenting or replacing behaviours on existing objects. Holographic objects are similar in implementation, but focus instead on reproducing the behaviour of base objects with no actual base object available in the runtime to provide the base behaviour. In our case behavioural intercession is limited to the case of replacing native methods, with the specific requirement of not deviating from base behaviour, and is not exposed to developers that use the library.

In addition, prior work has presented implementations of such objects in dynamic languages, and doing the same in a statically-typed language such as JVM bytecode without requiring a custom language runtime presents fundamental challenges that affect the design of the interfaces to those objects. The Jikes Research Virtual Machine [3] also implements the behaviour of a JVM on another JVM, and the majority of its code is ordinary Java. Its object model is not intended to be pluggable, however, and although it could be made so this would not necessarily be any easier than customizing any other JVM implementation.

Lorenz and Vlissides [13] describe how pluggable reflection enables more flexible language tools, using a documentation generator and an object-to-component generator as examples. A holographic language runtime represents another client of a language’s pluggable reflective API, and allows the language’s implementation itself to be decoupled from the representation of its runtime state. Unlike Lorenz and Vlissides’ examples, holographic execution requires a reflective API that represent computation and not just code, a distinction clarified by Bracha and Ungar [5].

6.2 **Reproducing Past State and Behaviour**

The idea of checkpointing and resuming execution recurs in several contexts, notably in operating system or hardware virtualization. Some language runtimes also support resuming from a snapshot [1], and Java itself includes a small amount of this behaviour in its shared memory class file cache feature. In all of these cases the restored process is a normal, unrestricted instance, and care must be taken to ensure that invalid references to the outside world are not created. By contrast holographic execution as described in this paper is intended to support diagnosis and analysis of the state of a system in the past. It requires no foresight, but at the cost of restricting additional execution and hence not truly restoring a dead process. It also does not currently support resuming execution from the exact time of the snapshot, although this is conceivably possible if a more precise snapshot such as a core dump were used, along with techniques for recreating the captured call stack via program slicing [18].

6.3 **Heap Dump Analysis**

Several other approaches have been used to analyse heap dumps, usually in the context of identifying the source of
memory leaks. Maxwell et al. [14] use graph mining to locate potential leak candidates. As we have illustrated in Section 5.1, holographic execution is a complementary technique which does not preclude the use of the reflective API on a heap dump. For example, the aforementioned work includes a case study of a memory leak in a scripting language parser. The graph mining technique identifies a non-standard linked-list implementation containing a long series of regular expression matches, which is helpful but does not fully diagnose the root cause. We postulate that applying holographic execution to print out the semantic contents of this list could be extremely useful in diagnosing the actual source of the leak.

6.4 Static Code Analysis

Kozen and Stillerman [12] use a static analysis of class initializers similar to ours to initialize classes eagerly, in order to improve startup performance and catch errors earlier. Their algorithm ignores initializer effects with respect to static fields, but is flow-sensitive and hence calculates a more precise definition of initialization dependencies than our current implementation. Integrating their approach in the future may improve the success rate of our algorithm and hence reduce the number of initializers that must be marked safely repeatable.

7. Future Work

Object holographs appear applicable to several other sources of object behaviour other than heap dumps.

- When debugging a live, remote JVM using the JDI, object holographs could be used to evaluate expressions in a separate client JVM instead of on the target, eliminating the possibility of accidentally perturbing the debugged process.
- More advanced forms of execution capture, such as omniscient debugging [17], present the state of a process at a single point in time using a similar reflective model equally incompatible with evaluating expressions.
- Deterministic replay (DR) is a very active area of research [10] [11] [19] that aims to reproduce a prior execution of a system by recording some subset of the system’s behaviour, and then instrumenting the system to force non-deterministic operations to behave as they previously did. Triggering additional execution in a DR process via a debugger can easily cause the replay to diverge due to the fragile nature of the techniques involved. Using holographic execution to avoid perturbing the DR process appears to be an ideal solution.

An important optimization direction is to improve the bytecode loading and translating workflow. Ideally it should be possible to produce hologram bytecode once for each version of a class as it is developed, rather than repeatedly for each heap dump it occurs in. The translation process cannot currently be applied to a single class file in isolation, however, as the standard JVM bytecode type inferencing algorithm requires access to the context of the class hierarchy. Additional engineering effort should make it possible to remove this dependency, possibly by abstracting the type inferencing to leave placeholder types and replacing them with actual types only after caching the translated bytecode.

8. Conclusion

We have presented an architecture for holographic objects, which enables restricted execution starting from the state captured in a heap dump. We have shown that this architecture supports analysis that is simple, type-safe, robust, secure, and familiar. It provides a good fit for analysis that depends on application-specific semantics, but also complements meta-level analysis by supporting a hybrid approach. Our initial experience with our implementation for Java bytecode suggests that this architecture can be effectively realized without having to customize a JVM, and is feasible for heap dump debugging and analysis, as we have presented evidence that holographic execution is competitive with live execution. We have also applied this prototype implementation to diagnose a real-world memory leak bug that went unsolved for several months. While this research is still relatively young, we are excited about continuing to build on these initial promising results.

Acknowledgments

We are extremely grateful for the feedback and encouragement received from Andrew Warfield, Ronald Garcia, and Adrian Kuhn on this work. We also thank the anonymous OOPSLA reviewers for their insightful comments.

A portion of this work was funded by an NSERC PGS D award and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

A. Illegal Native Methods in the JRE

This appendix contains our classification of all illegal native methods in the JRE. See Figure 4 for the complete list.

B. Idempotent Class Initializers

This appendix contains the list of classes we found it necessary to manually mark as having idempotent initialization, because their initialization state could not be automatically inferred. As mentioned in Section 5.3, it is likely that further refinement of the inference algorithm will eliminate some of these cases.

1. java.lang.reflect.Modifier
2. java.net.AbstractPlainSocketImpl
3. java.net.URLClassLoader
4. java.security.KeyFactory
5. java.security.SecureClassLoader
7. org.eclipse.e4.core.di.internal.extensions.EventObjectSupplier
8. org.eclipse.ui.internal.misc.Policy
9. sun.reflect.UnsafeStaticFieldAccessorImpl


<table>
<thead>
<tr>
<th>Category</th>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Class Initialization</td>
<td>java.lang.System</td>
<td>registerNatives</td>
</tr>
<tr>
<td></td>
<td>java.util.concurrent.atomic.AtomicLong</td>
<td>initProperties</td>
</tr>
<tr>
<td></td>
<td>sun.misc.VM</td>
<td>VMSupportsCS8</td>
</tr>
<tr>
<td>Drivers</td>
<td>sun.print.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.security.smartcardio.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>apple.laf.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.apple.eawt.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.apple.laf.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.sun.java.swing.plaf.gtk.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.awt.<em>, sun.awt.</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.lwawt.*</td>
<td></td>
</tr>
<tr>
<td>GUI</td>
<td>com.sun.imageio.plugins.jpeg.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.dc.pr.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.font.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.java2d.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.apple.eio.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.sun.java.util.jar.pack.NativeUnpack</td>
<td></td>
</tr>
<tr>
<td>IO</td>
<td>java.util.logging.FileHandler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.util.prefs.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.MessageUtils</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.Compiler</td>
<td></td>
</tr>
<tr>
<td>Java 7 Method Handles</td>
<td>java.lang.invoke.*</td>
<td></td>
</tr>
<tr>
<td>Management</td>
<td>com.sun.demo.jvmti.hprof.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>oracle.jrockit.jfr.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.management.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.Perf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.tracing.dtrace.JVM</td>
<td></td>
</tr>
<tr>
<td>Media</td>
<td>com.sun.media.sound.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.ClassLoader$NativeLibrary</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.System</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.net.<em>, sun.net.</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.rmi.*</td>
<td></td>
</tr>
<tr>
<td>Native Libraries</td>
<td>java.lang.ClassLoader$NativeLibrary</td>
<td>mapLibraryName</td>
</tr>
<tr>
<td></td>
<td>java.lang.System</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.Runtime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.UNIXProcess</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.util.TimeZone</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.GC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.NativeExceptionHandler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.SignalHandler</td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>apple.security.<em>, sun.security.</em></td>
<td></td>
</tr>
<tr>
<td>Shared superclass</td>
<td>java.lang.Object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.Throwable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>apple.applescript.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>apple.launcher.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.apple.concurrent.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.apple.jobc.*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>com.apple.resources.LoadNativeBundleAction</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>java.lang.ProcessEnvironment</td>
<td>environ</td>
</tr>
<tr>
<td></td>
<td>java.lang.Runtime</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.lang.UNIXProcess</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.util.TimeUnit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.GC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.NativeExceptionHandler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sun.misc.SignalHandler</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Categorization of forbidden methods in the Java Runtime Environment.