Can You Trust Your JVM Diagnostic Tools?

Isaac Sjoblom, Tim S. Snyder, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
sjobl014@umn.edu, snyde479@umn.edu, elenam@umn.edu

Abstract

Programmers may use Java diagnostic tools to determine the efficiency of their programs, to find bottlenecks, to study program behavior, or for many other reasons. Some of the common diagnostic tools that we examined are profilers and the Java Virtual Machine (JVM) options that make some internal JVM information available to users. Information produced by these tools varies in degrees of clarity, accuracy, and usefulness. We also found that running some of these tools in conjunction with a program may affect the program's behavior, creating what we refer to as an "observer effect". We examine several tools and discuss their level of usefulness and the extent to which they impact program behavior. Additionally, we discovered program instability, i.e. a tendency of a program to change its behavior when executed with different monitoring tools or multiple times with the same tool. We discuss potential causes for instability based on information obtained via running the HotSpot JVM with an option for logging its internal compilation and optimization process.

1 Introduction

Java programming language is executed by an interpreter known as the Java Virtual Machine (JVM). Modern JVMs are equipped with sophisticated tools for dynamic program optimization that profile and optimize the program as it is being executed. In this complex setup, monitoring a program by trying to examine its run-time behavior becomes quite challenging. The reason for this is that the monitoring tool and the JVM that is running and optimizing the program all share the same resources (CPU, cache, etc). This may create an "observer effect": the change in the program behavior when it is being executed together with a monitoring tool, whether embedded into the JVM itself or an external one.
In this project we are interested in studying Java program behavior caused by a certain inheritance pattern related to generic types that we refer to as *bound narrowing*. Programs with bound narrowing that are very similar to each other exhibit different behavior in terms of their run time. While trying to explain the differences, we tried various Java diagnostics tools, both stand-alone (such as HPROF profiler) and those embedded into the HotSpot™VM. In particular, we explored a *log compilation* option available in the HotSpot JVM. This option keeps a detailed log of optimizations performed at the run time.
We present the observations about the accuracy and usefulness of the monitoring tools, using bound narrowing examples as a case study. We show that most of the programs are *stable* with respect to the monitoring tools considered in this paper, i.e. relative times in a group of programs remain the same regardless of the tools (although constant differences in times for the entire groups have been observed). However, some programs are *unstable*, i.e. their behavior patterns change depending on whether they run in the JVM under default conditions or with a monitoring tool. In fact, some programs behave differently when executed repeatedly in the JVM with the *log compilation* option. Since this option produces a detailed log of both successful optimizations and failed attempts, we were able to determine what causes the differences in behavior based on these logs.

2 Java Virtual Machine and Diagnostic Tools

2.1 Java Compilation Model

Unlike statically compiled languages, in which a program is optimized and compiled into machine code in the same step, Java has a two-phase compilation model. First, the program is compiled by a *static compiler*, such as Javac, into bytecode for portability. Second, the bytecodes are executed on any system that has a Java execution environment, referred to as the *Java Virtual Machine* (JVM). The JVM that comes standard with the Java Development Kit (JDK) is called HotSpot™VM.
Most modern JVMs, including HotSpot, are equipped with a *Just-in-time compiler* (JIT) that performs optimizations as the program is being run and may convert a portion or all of a program's bytecode to native code "on the fly". This process is called *dynamic compilation*. Two common examples of JIT optimizations are *constant propagation* and *inlining*. Constant propagation is when references to a constant in a program are replaced with the constant's value. Inlining is when the JIT replaces method calls with all of the method in-

structions. This removes the method call, and therefore method look up, and allows for further optimizations to take place. Inlining would be similar to copying a whole method and pasting over the call to that method.

2.2 HotSpot Server Just-In-Time (JIT) Compiler

The HotSpot JVM features two modes, *client* and *server* that differ in their JIT compilers. The server mode provides more optimizations than the client mode and as a result may have faster run times than client for longer programs. However, the optimizations may be more time-consuming than those applied by the client mode [6]. The server mode is intended for server-side applications. Client mode is the default for HotSpot on 32-bit architecture. It provides faster application startup and requires less memory than server. It is intended for client-side Java programs, such as applets, and is currently not supported for 64-bit architecture [3]. In this paper we are only considering the server mode.
HotSpot JVM provides multithreading support. Multithreading is the parallel execution of a program through various processes, effectively executed by the hardware but may be handled by the virtual machine. When handled by the hardware, the processes are divided between the systems processors. However, our system has one processor to limit variations during dynamic compilation as much as possible. A feature of this JVM is preemptive multithreading based on native threads; that is, that threads defined in a program are handled by the JVM and passed to the operating system for support. We observe that this extends to multithreading JIT compilation, that is, optimizations in JIT compilation may take place in different threads.

2.3 Profilers, HPROF

A profiler is a piece of software that observes and interacts with a program to gather information about its execution, such as data on how often methods get called. HPROF is a specific profiler that can run in two different modes, *sampling mode* and *method counting mode* [2]. Sampling mode will run by finding out what the stack is like at certain time intervals to get an estimate for how often methods are being called and where they are on the stack most. Method counting mode, on the other hand, will count how often methods get called by putting in at least one line of code that creates a counter in the method that increments on each call. This approach is called *bytecode injection*. Method counting mode causes a drastic increase in run time due to an overhead of counting. In addition inlining is disabled because of bytecode injection. This is a type of "observer effect": observing a program by means of a profiler changes its behavior [1].

2.4 Internal JVM Diagnostics

Internal JVM diagnostic tools were created by the JVM developers to provide themselves access to various internal JVM information. These tools were not developed for Java programmers, which may be the reason that the internal JVM diagnostic tools are somewhat under-documented and not guaranteed to work for every JVM distribution [5].

```
...
14 HashMap::putAll (133 bytes)
15 NarrowedIS::put (13 bytes)
16 NarrowedIS::put (38 bytes)
1% HashMap::putAll @ 90 (133 bytes)
17 HashMap::entrySet0 (28 bytes)
18 HashMap$EntrySet::iterator (8 bytes)
19 HashMap::newEntryIterator (10 bytes)
...
```

```
...
14 HashMap::putAll (133 bytes)
15 NarrowedIS::put (13 bytes)
16 NarrowedIS::put (38 bytes)
17 HashMap::entrySet0 (28 bytes)
18 HashMap$EntrySet::iterator (8 bytes)
19 HashMap::newEntryIterator (10 bytes)
...
```

Figure 1: Showing different results for print compilation for two runs of the same program (only a fragment shown).

Two of the relevant diagnostic tools we were using are *Print Compilation* and *Log Compilation*. They are both flags provided by the JVM. On the command line the flags look like this [5]:

```
-XX:+PrintCompilation
-XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation
```

When the print compilation flag is turned on, a message is printed whenever JIT translates a method into machine code. This flag provides some information about the JIT behavior, but its output format does not have a clear publicly available documentation and the results can be confusing. An example of this is the field *size of method in bytes* where the number associated with this field is, as evidence suggests, a somewhat arbitrary number to approximate the size of the method but does not have a clear definition. A method is often compiled to native code more than once, with different byte sizes. An additional complication arises from the fact that output of print compilation differs for multiple runs of the same program, even if the runs take the same time, as demonstrated in Figure 1. In Figure 1 we show a partial print compilation output for two runs of the same program that resulted in the same running times. We see that between lines 16 and 17 of the output of the first run there is a reference to `HashMap::putAll` that does not appear in the second run's output.

The log compilation flag turns on logging of the compilation process: when and in what order program code is translated into bytecode, optimized and translated into native code where applicable. It also records the information about threads of the JIT compiler and which tasks they perform. When the log compilation flag is turned on, a second flag, unlock

diagnostics, must also be turned on. The log is written as a large XML file recording every action of the JIT, with timestamps. This option produces accurate detailed information for analyzing program behavior. However, we have observed that using these flags leads to changes in the behavior of some programs, most likely due to recording information forced by unlocking the JVM diagnostics See section 5 for more details.

3 Effects of Bound Narrowing

3.1 Generic Types in Java

Generic types are a feature of Java that allows writing program code that is parameterized over types. For example, generic types allow creation of a `Stack` class that that allows instances of `Stack` to be constructed with any object such as strings or integers, but will not allow mixing strings and integers in the same instance of `Stack`. In this case `String` or `Integer` class are referred to as type parameters for the generic `Stack` class. An example of generic types is the following class:

```
public class HashMap<K, V>
```

`HashMap` has two type parameters, a key type `K` and a value type `V`. An example of the instantiation of `HashMap` would be:

```
HashMap<K,V> myHashMap = new HashMap<Integer,String>();
```

Here, we instantiate `HashMap` with an `Integer` for a key and a `String` for a value. This limits the instance of this class `myHashMap` to only be used with an `Integer` for a key and a `String` for a value.
There can also be *type bounds* placed on generic types. A type bound is a class or interface that limits the generic parameters to that class or subtypes of that class or interface.

```
public class ComparableHashMap<K extends Comparable,
                               V extends Comparable>
```

This ensures that the key and the value are of types that implement `Comparable` interface. Implementations of `HashMap` allow for construction of an instance involving any class for the key or the value. `ComparableHashMap` has a more strict implementation where the construction of an instance demands a `Comparable` or a subtype of `Comparable` for the key and value.
Generic types may be subtypes of other generic types. Commonly the type bound of a subtype is the same as the type bound of its supertype. However, Java allows for the subtype to have a more restrictive bound than its supertype; also, a non-generic class may inherit from a generic class or interface by "hard-wiring" a specific type for a type parameter. We refer to these inheritance patterns as *bound narrowing*.
An example of bound narrowing where a non-generic class is inheriting from a generic class is the following:

```
public class NarrowedHashMap
     extends HashMap <Integer, String>
```

Here, `NarrowedHashMap` inherits from the class `HashMap` where the key and value are specifically `Integer` and `String`, respectively.

3.2 Delays Associated with Bound Narrowing

In our previous research, we found that bound narrowing causes substantial runtime inefficiencies when the JIT is disabled. When bound narrowing is present, the system has to verify that objects passed to a method of a more specific (narrowed) container by a call to a method of its more general supertype are of the correct type. When the JIT is not allowed to perform the necessary optimizations such as inlining, described above, these type checks have to be performed every time an object is placed into a container. We have found that some of these delays will go away in runs under default HotSpot JVM conditions (with the JIT optimizations on) while others remain. In the current research, we are continuing to study bound narrowing, specifically when the delays remain and why [7].

3.3 Test Cases

The code that we have been testing is based on the `HashMap` class in the Java collections library. We chose this code base because of a specific class called `PrinterStateReasons` that is a part of the javax package. The reason that `PrinterStateReasons` is important to us is because it is a narrowed version of the generic `HashMap` and is a real-life example of bound narrowing. Below, we are showing only relevant fragments of code.

```
public final class PrinterStateReasons extends
    HashMap<PrinterStateReason, Severity>
```

The bound here is on the way down from `HashMap` because the `PrinterStateReasons` class has a more specific bound than the generic `HashMap`. `PrinterStateReason` is a class that provides information about a printer's current state while `Severity` provides information on how severe that state is. Our version of this code uses `Integer` and `String` instead since `PrinterStateReason` and `Severity` have a limited number of instances.
Our test code accesses the class being tested via an interface, `Map`, with a key and a value both bounded to `Object`. We have a class, `HashMap`, that implements this interface that is still completely generic. These classes are copies of the standard Java collection packages and `PrinterStateReasons` with our own methods added as needed.

```
public class HashMap<K,V>  extends Map <K, V>
```

Our *test classes*, which are the four subclasses in the table below, all extend `HashMap` and are based off of the `PrinterStateReasons` class . Each of these has a different bound combination as seen in the table below.

Name	Notation	Key	Value
NarrowedIS	NIS	Integer	String
GenericKeyIS	GKIS	Object	String
GenericValueIS	GVIS	Integer	Object
GenericKeyValueIS	GKVIS	Object	Object

We define four versions of the testing code, one per each of the test classes above, that test our code base by repeatedly calling a method in order to determine its running time.
Our tests start with the generic `Map` interface instantiated as one of our four classes. We then call a method on the interface and the call gets passed down to the specific class. The narrowing happens on the arguments passed from the method in `Map` to the method in our specific class . The check happens in those classes because in `Map` the arguments are bound to `Object` while the narrowed classes are bound to either `Integer` or `String`. Note that the type check may be optimized away by the JIT (see Section 3.2).
Our *test cases*, the methods that we are calling in our test classes, for the most part, use a pair of methods that perform an item-by-item search through the `HashMap` to try to find either a key or value equal to what we pass in. An example of this is the `containsValue` method shown below. This is just one of many different methods that we have tested.

```
public boolean containsValue(V value) {
    Entry[] tab = table;
    for (int i = 0; i < tab.length; i++)
        for (Entry e = tab[i]; e != null; e = e.next)
            if (value.equals(e.value))
                return true;
    return false;
}
```

This method is a copy of the method in `HashMap` which we duplicated for studying purposes. The table variable is the array of type `Entry` that `HashMap` uses to store its data. Between the four classes of ours, two (`NIS` and `GKIS`) have the `String` type bound for `V`, and the other two have the `Object` type bound. Consequently the type of the parameter `value` in the bytecode is `String` in `NIS` and `GKIS` and `Object` in `GVIS` and `GKVIS`.

```
Map<Integer, String> map = new GKVIS<Integer, String>();
initMap(size, keys, values, map);
boolean result = true;
double timeStart = System.currentTimeMillis() / 1000.0;
for (int i = 0; i < outerLoops; i++) {
    for (int j = 0; j < 10000; j++) {
        result &= map.containsValue(values[j % size]);
    }
}
double timeEnd = System.currentTimeMillis() / 1000.0;
if (result) {
    System.out.println(timeEnd - timeStart);
}
```

`initMap()` will add items to our `Map` from our set of keys and values until we reach the size that we pass in. The given arrays have only four distinct elements (repeatedly added to the `HashMap` as needed) to eliminate the need for garbage collection and thus to guarantee that the running times are not affected by it. We record the start time before we run our

tests, and then record it again right before the program finishes. The `result` variable (a boolean) alternates between `true` and `false` in the loop and is used after the loop (with the value `true` since the loop runs an even number of times). We use this "dummy" variable `result` so that the JIT compiler does not optimize the method calls away entirely since the JVM compiler will eliminate method calls as useless code if there is no side-effect from the call.
In order to obtain running times on the order of seconds, we loop over method calls a large number of times (200,000,000 in examples in this paper). This allows us to accurately measure the differences in time between the test classes. We separate the iteration into doubly nested for loops for easier control of the number loops (the outer loop counter is set as a flag for the test runs).

3.4 Other Test Cases

The other methods in `HashMap` hierarchy that we tested are listed below:

- `IS-put-inl` is the same method as `put` in `HashMap`, with the test method for it manually inlined into the classes that we are testing, such as `NIS`, `GKIS`, etc. The reason for manually inlining the testing method was to explore a particular way of parameter passing in the program.

- `IS-contains-v-inl` is the same method as `IS-contains-v`, with the test method inlined in the same way as `IS-put-inl`.

- `II-contains-v-inl` is the same method `IS-contains-v` except the type of `value` is `Integer` instead of `String` and its testing method is inlined in the same way as `IS-put-inl`.

4 Effects and Accuracy of Java Monitoring Tools

4.1 Pattern Classification for Test Cases and Program Stability

In our tests we have come across different patterns of time differences between the four test classes, three of which come up in the tests that we describe in this paper. First, there is a *no significant difference* pattern which is a run where the four classes all run in the same times (we consider differences under 0.4 sec to be non-significant, although most cases in this category are much tighter, often within 0.1 sec). Second, there is the *bound narrowing* pattern that manifests the bound narrowing delay described in Section 3.2, where the runs with the narrowed value (`NIS` and `GKIS`) are slower than the generic runs (`GVIS` and `GKVIS`). The final pattern is what we call the *value pattern*. The value pattern is where the narrowed value classes are faster than the generic classes. In these runs the JIT successfully optimizes away the bound narrowing delay and then takes advantage of knowing the exact class that the `equals` method in `containsValue` (and in similar methods) is called on.

The exact target class information allows JIT to inline the method, leading to faster runs for the narrowed tests[1].
Our goal is to determine why some runs display a specific pattern. We also discovered that some test cases act differently under different testing conditions, for instance displaying the bound narrowing pattern under the default conditions and the value pattern when tested with HPROF. We refer to runs that exhibit *the same pattern* for all testing methods (i.e. the default mode of testing, HPROF, Print Compilation and Log Compilation) as *stable*, and to those that have different patterns as *unstable*. An unstable test case may also have two different behaviors in the same mode of testing in different tries. In our experience this may happen in Log Compilation tests.
In the remainder of the section we show results for some stable and some unstable runs for different ways of testing and discuss the result.
All tests are run on the same machine in /tmp to ensure that network problems do not interfere with the tests. Below are the machine and the JVM specifications:

```
AMD AthlonTM 64 Processor 3200+
512MB DDR RAM
Fedora Core 7
Kernel: 2.6.23.17-88.fc7 SMP i686
Java Version: Sun JDK 1.6.16
Time Binary: GNU time 1.7
```

We use the `taskset` command to pin the JVM process (and all of its subprocesses) to one CPU.

4.2 Results for Default Conditions.

In this section we show the results of running the four test cases described in Sections 3.3 and 3.4 under the default conditions in the server mode of the HotSpot JVM. Figure 2 shows the graph for `containsValue` corresponding to the data in the table. The graph exhibits the bound narrowing pattern. Due to lack of space we are not showing the graphs for the other three test cases and just summarize all four results in the table:

Name	**NIS**	**GKIS**	**GVIS**	**GKVIS**	**Pattern**
`contains-v`	16.18	16.19	15.40	15.55	BN
`put-inl`	8.48	8.50	8.05	8.07	BN
`contains-v-inl`	13.94	13.91	15.40	15.72	Value
	NII	**GKII**	**GVII**	**GKVII**	
`contains-v-inl`	8.09	8.11	8.09	8.09	No-diff

The first column is the name of the test case: `containsValue` is the test class explained in Section 3.3, and the other four runs are described in Section 3.4. We make 200,000,000 method calls for each method being tested in each of the four test classes. The numbers

[1]In tests that call `equals` method on a key and not on a value we also found a *key pattern* which is the same as the value pattern, but it is sped up on runs with a narrowed key (`NIS` and `GVIS`) instead of a narrowed value. Due to space limitations we are not describing these tests in the paper.

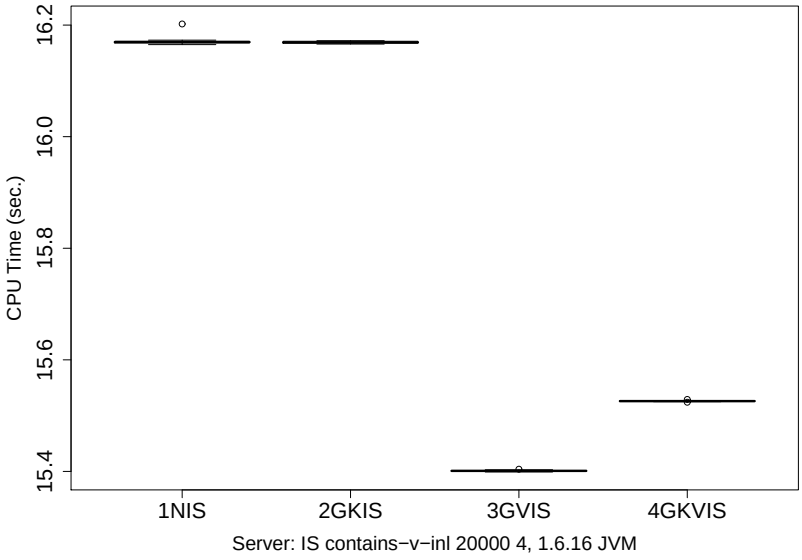

Figure 2: `containsValue`, default HotSpot conditions: bound narrowing pattern.

are the averages of program running times, in seconds, for 20 repetitions of each test class. The last column shows which pattern the test case follows: "BN" stands for the bound narrowing pattern, "Value" stands for the value pattern and "No-diff" means that there were no significant difference between the four runs.

4.3 Results for HPROF in Method Sampling Mode

Figure 3 shows the results of running the `IS-contains-v` test case with the HPROF profiler in the sampling mode (described in Section 2.3). Not surprisingly, the running time is more that twice that of the original's. This is due to the overhead of collecting the stack sampling information as the program is running. Quite surprisingly, however, the test exhibits the value pattern: the code versions specialized in the value run faster than those with a generic value. This contrasts the bound narrowing pattern of this test under the default conditions.

The difference in pattern can clearly be characterized as an observer effect. However, this observer effect is surprising for two reasons. Firstly, unlike the counter mode of HPROF, the sampling mode does not perform bytecode injection. The sampling of the program stack is performed independently from the program itself. Although it requires stopping the program every once in a while to record the contents of the stack, this only adds a constant factor to the program running time and should not affect the relative running times of the four different versions. The second reason this result is unexpected is that most of our test cases (including all three of the comparison runs) are stable with respect to HPROF, i.e. they preserve their default pattern[2] The table below summarizes the results for all four

[2]We observed a similar instability in a test similar to `IS-contains-v`, not shown due to lack of space.

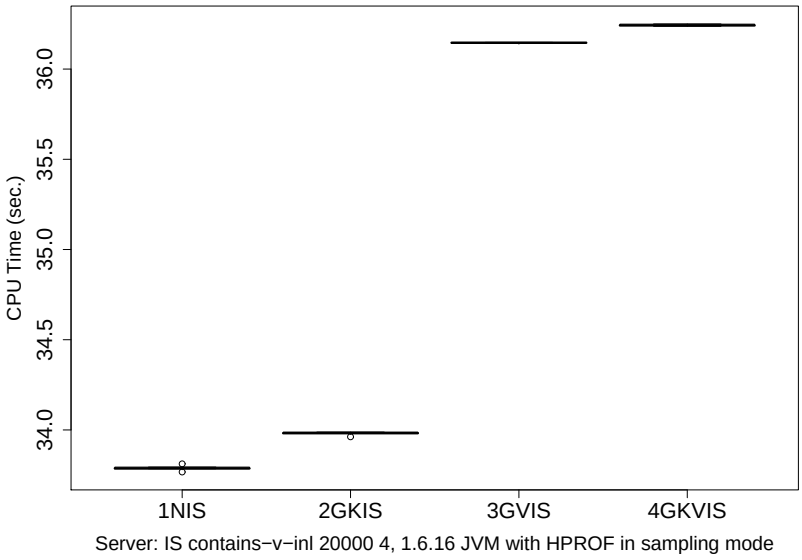

Figure 3: `containsValue`, HPROF in Method Sampling: value pattern

test cases. The measurements are determined exactly the same as for the default conditions, except each run is repeated 5 times, not 20.

Name	**NIS**	**GKIS**	**GVIS**	**GKVIS**	**Pattern**
`contains-v`	33.79	33.98	36.14	36.24	Value
`put-inl`	16.89	16.89	16.05	16.06	BN
`contains-v-inl`	34.58	33.22	35.84	35.84	Value
	NII	**GKII**	**GVII**	**GKVII**	
`contains-v-inl`	17.28	17.13	17.28	17.13	No-diff

4.4 Results for Print Compilation and Log Compilation Flags

Print compilation results do not change the running time for any of the test cases we have tried, and therefore we are not including its results. Log compilation, however, leads to interesting results for `IS-contains-v`, see Figure4. Some of the results exhibit bound narrowing pattern, and some have value pattern. These results confirm the fact that `IS-contains-v` is unstable. They also provide us with compilation logs for both of the patterns that allow us to study the differences in JIT behavior between the bound narrowing and the value pattern of the same program. Section 5 shows the important fragments of the logs and provides discussion. Note that the other three runs are stable, i.e. preserve the default pattern, as shown in the table below. The averages are for five runs per test class.

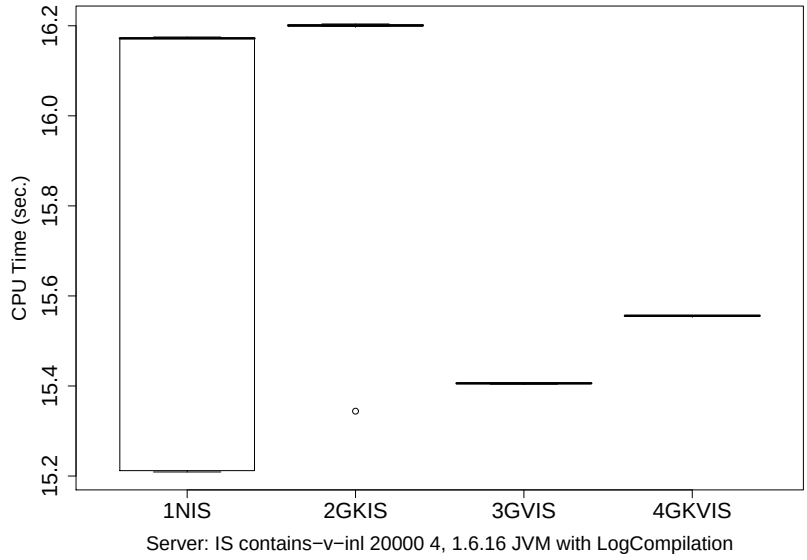

Figure 4: `containsValue`, Log Compilation: split between bound narrowing and value patterns

Name	**NIS**	**GKIS**	**GVIS**	**GKVIS**	**Pattern**
`contains-v`	2: 15.21 3: 16.17	1: 15.34 4: 16.20	15.40	15.56	Unstable
`put-inl`	8.48	8.51	8.05	8.07	BN
`contains-v-inl`	14.05	13.95	15.58	15.35	Value
	NII	**GKII**	**GVII**	**GKVII**	
`contains-v-inl`	8.09	8.09	8.12	8.09	No-diff

5 Analysis of Compilation Logs and Implications

5.1 Compilation Logs

We have provided a sample of two of the logs obtained by the Log Compilation tests of `IS-contains-v` for the `NIS` run: one faster run at 15.21 seconds and one slower run at 16.17 seconds. We are only showing fragments of the logs for space and clarity reasons since the logs themselves are large XML files. The first of these is a run that exhibited the value pattern and the second exhibited the bound narrowing pattern. For more in depth explanations of the log compilation terms, see [4].

```
FASTER RUN:
First Thread:
<klass id="511" name="java/lang/Object" flags="1"/>
<method id="609" holder="533" name="equals" return="486"
```

```
      arguments="511" flags="1" bytes="88" iicount="6611"/>
<call method="609" count="4188" prof_factor="1" inline="1"/>
<method id="610" holder="533" name="equals" return="486"
      arguments="511" flags="1" bytes="88" iicount="6612"/>
<call method="610" count="4189" prof_factor="1" inline="1"/>

Second Thread:
<task compile_id="1" compile_kind="osr"
method="TestNarrowedIS main ([Ljava/lang/String;)V"
bytes="2330" count="1" backedge_count="14564" iicount="1"
osr_bci="942" stamp="0.125">
<method id="715" holder="533" name="equals" return="486"
arguments="511" flags="1" bytes="88" iicount="6612"/>
<call method="715" count="4189" prof_factor="1" inline="1"/>

SLOWER RUN:
<task compile_id="1" compile_kind="osr"
method="TestNarrowedIS main ([Ljava/lang/String;)V"
bytes="2330" count="10000" backedge_count="5803" iicount="1"
osr_bci="942" stamp="0.131">
<klass id="511" name="java/lang/Object" flags="1"/>
<method id="707" holder="603" name="containsValue"
return="486" arguments="511" flags="4161" bytes="9" compile_id="2"
compiler="C2" level="2" iicount="10000"/>
<dependency type="unique_concrete_method" ctxk="603" x="707"/>
<call method="707" count="35620" prof_factor="1" inline="1"/>
<inline_fail reason="already compiled into a big method"/>
```

The abbreviation *osr*, as seen in the samples, is short *for on-stack replacement*. On-stack replacement is the process of replacing a method with its more optimized version directly on the program stack as the program is being executed. Log Compilation shows that the JIT runs two separate threads that start at the same time. Our observations show that there is no significance to a particular operation taking place in the first or the second thread. However, we have found that second thread will typically have the optimizations for faster runs. The iicount and backedge_count are for how many times the method in question gets called, but they do not seem to be very important. The key is whether or not the test case gets inlined into the calling context successfully.

The first run, the faster of the two and showing the value pattern, inlines `containsValue` into `main` first, and it later inlines `equals` into `containsValue`. The second run, the slower of the two and showing the bound narrowing pattern, does things the opposite way, inlining `equals` into `containsValue` first. However, when it tries to inline `containsValue` into `main`, it fails with the reason of ”already compiled into a big method”. This fail reason means that `containsValue` was too large after it was inlined with `equals` to be inlined into `main`. In this specific test, `main` was the calling context (see the second code fragment in Section 3.3) of `containsValue`.

The logs for the tests we ran fall under one of three different categories. If, as is the case for `IS-put-inl` the narrowed test classes fail to inline into the calling context, we get the bound narrowing pattern. `IS-put-inl` is a more complicated method, and we do not have complete analysis of its compilation log at this point. We observed that in the case of `IS-contains-v-inl` the narrowed test classes succeed in inlining the testing method into the calling context, but the generic ones fail. This corresponds to the value pattern (the narrowed classes are faster than the generic ones). Finally, when all four of the classes succeed in inlining into the calling context, we get the no significant difference pattern which is the case for `II-contains-v-inl`.

5.2 Analysis of Compilation Logs and Hypotheses

In our analysis of the compilation logs, we have come up with a hypothesis to explain the three different patterns that we have seen. No significant difference shows up if the compilation log has both the narrowed and generic classes succeed in inlining the test case into its calling context. The value pattern shows up if the inlining of the generic classes fails and the inlining of the narrowed succeeds. Bound narrowing shows up if narrowed classes fail to inline the tested method into the calling context, regardless of whether the generic classes succeeds or fails to inline the test case. We have found that if generic fails to inline and narrowed fails, then it shows bound narrowing. We have reason to believe that if generic succeeds inlining then the bound narrowing will be more extreme.
In addition, we have observed that test runs with an `Integer` bound for a value (rather than the `String`, as in most of our examples) are stable and fall into value pattern or no significant difference, except on `put` runs. We believe that the reason for this is because the method size of `equals` in `Integer` is smaller which make the inlining of the method work irregardless of order. The exception being `put` because the methods for those runs are larger and fail to inline.

6 Conclusions and Future Work

The table below shows the summary of comparing different diagnostic methods:

Diagnostic Method	**Time Change**	**Pattern Change**	**Information Amount**
Print Compilation	no time change	no pattern change	some
hprof(sampling)	approximately doubles times	can change patterns	moderate amount
hprof(counting)	drastic increase	significantly	inaccurate
Log Compilation	no significant change	can change the pattern	detailed, accurate

Print compilation is the method with the least impact, but it does not produce enough information to be able to detect important optimizations, in particular inlining. As shown in [1], HPROF in the method counter mode creates a drastic time increase and may disable inlining, therefore it is not useful for our purposes. The sampling mode of HPROF increases the running times by a moderate amount. However, it may change the behavior of unstable

programs. Finally, log compilation produces helpful information, but may also change the behavior of unstable programs. We observed that unstable programs may produce different running times in repeated runs with log compilation. We were able to analyze the logs for two runs with the log compilation option that resulted in two different running times. The logs gave us an insight into the behavior related to the bound narrowing inheritance. We were able to identify a sequence of optimizations that leads to failure to inline the method under testing. We developed a hypothesis for explaining the distinction in patterns for our current set of tests.

Future work includes analysis of more compilation logs with the goal of generalizing the patterns to other tests, such as those that use `put` methods which we currently cannot explain. We also plan to check the dependency of this approach on the maximum method size for inlining by setting the JVM flag that controls this parameter. Another promising direction to explore is multithreading in JIT: we can set the number of threads and thus get an indirect control over thread scheduling. Finally, we plan to identify code patterns that lead to program instability. It may be wise to advise programmers to avoid these patterns.

References

[1] MACHKASOVA, E., ARHELGER, K., AND TRINCIANTE, F. The observer effect of profiling on dynamic java optimizations. In *OOPSLA Companion* (2009), pp. 757–758.

[2] O'HAIR, K. *HPROF: A Heap/CPU Profiling Tool in J2SE 5.0*, 2004.

[3] ORACLE TECHNOLOGY NETWORK. *Frequently Asked Questions About the Java HotSpot VM*. http://www.oracle.com/technetwork/java/hotspotfaq-138619.html.

[4] ORACLE TECHNOLOGY NETWORK. *HotSpot Glossary of Terms*. http://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html.

[5] ORACLE TECHNOLOGY NETWORK. Java hotspot vm options. *ORACLE*.

[6] SUN DEVELOPER NETWORK. The java hotspot performance engine architecture. *Sun Microsystem* (2007).

[7] TRINCIANTE, F., SJOBLOM, I., AND MACHKASOVA, E. Choosing efficient inheritance patterns for java generics. In *MICS 2010* (2010), pp. 1–19.