The Role of Method Call Optimizations in the Efficiency of Java Generics

Jeffrey D. Lindblom, Seth Sorensen, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
lindb310@umn.edu, soren585@umn.edu, elenam@umn.edu

Abstract

The Java Virtual Machine (JVM) is the run-time environment of Java programs. It is an interpreter of bytecode, a language into which Java programs are initially compiled. Modern JVMs are packaged with the Just In Time (JIT) compiler, a compiler which performs a series of run-time optimizations as the program is being executed.

Some of these optimizations attempt to reduce the overhead of Java method calls. One of these optimizations is known as devirtualization. Devirtualization works to eliminate the overhead of virtual calls caused by the run-time look-up of target method calls in a class hierarchy. It does so by pointing the method call directly to its target method. Another of these optimizations is method inlining. Inlining goes one step further than devirtualization, and eliminates the method call entirely, replacing it with the methods code itself.

Our research focuses upon the efficiency of Java generic types, which are constructs that allow a programmer to define collections parametrized over a type T. When such a collection is instantiated, this type parameter T is replaced by a concrete type such as String or Integer. Making use of generic types may improve the run time of a Java program, but in other similarly structured cases a programs run time may suffer.

With relation to Java generics, we present observations on the optimizations of devirtualization and inlining and their corresponding effects on the run time. These observations are obtained using a variety of methods: profilers, run-time records of JVM operations, and direct timings of the Java programs. Given that the JVM is an extremely complex and fairly under-documented software system and most profilers provide a very limited amount of data it can be difficult to describe optimizations of the JVM with any amount of detail using these tools. We have however managed to detect specific instances of devirtualization and inlining taking place during the execution of Java programs.

1 Introduction

In its early days the Java programming language got a reputation of being slow. This is because Java code is compiled to bytecode which is then interpreted by the *Java Virtual Machine (JVM)*, and interpretation is a slow process. However, soon after Java release a significant research effort was put into incorporating run-time program optimizations into JVMs.

A modern JVM is a multi-threaded environment that continuously performs a large number of tasks. It creates and maintains a sophisticated internal code representation. At the same time it profiles a Java program being executed to determine areas that would benefit the most from optimizations and performs the optimizations. It selectively compiles bytecode to machine code and reloads methods on the fly, replacing them by their faster versions. All of these operations are performed in the span of milliseconds. In short, JVMs are some of the most complex software systems currently in wide use.

JVMs are constantly modified to produce the fastest code for the newest architectures. It is not surprising that predicting, or even understanding, JVM optimizations is a very challenging task. While there are tools that allow one to monitor and control the JVM, these tools are not necessarily kept up to date with the increasingly vast array of JVM features. Their accuracy may also be affected by the fact that they run concurrently with the JVM itself, possibly creating an "observer effect" by modifying the running times and the patterns of the JVM [9].

Our work stems from studying the performance of Java generic types, with a particular focus on an inheritance pattern which we refer to as bound narrowing [15]. This pattern may lead to certain method call optimizations. Our goal is to determine whether these optimizations actually take place. In this effort, we evaluate relative usefulness of tools that provide run-time information about a program. We have successfully detected certain types of method call optimizations by obtaining consistent results from several observation approaches.

2 Background

2.1 Java Execution Model

Unlike statically compiled languages, in which a program is compiled into machine code and optimized in the same step, the Java programming language has a two-phase compilation model. First, a Java program is compiled into an *internal representation* known as *bytecode*. This bytecode is then interpreted, optimized, and executed by the JVM. In our research, the JVM we work with specifically is the Java *HotSpot*™ Virtual Machine[1] [8].

[1]There are two modes of the HotSpot JVM: client and server. Our research focuses on the server mode since it typically produces faster code.

HotSpot, like most modern JVMs, comes equipped with the *Just-in-Time compiler* (JIT). As the name suggests, the JIT works in parallel with the JVM to manipulate and compile bytecode into other internal representations 'on-the-fly'. While bytecode is interpretable by the JVM on its own, efficiencies in speed can be obtained by using a different internal representation of the same instruction set. These Java program code transformations belong to a set of code modifications we call *JIT optimizations*. The JVM, through its complex analysis of the Java program code, schedules JIT optimizations throughout the execution of the Java program [14] [1].

2.2 JIT Optimizations

JIT optimizations are queued and subsequently executed as tasks by *threads*. Threads are spawned processes that can run in parallel to other processes. This queue processing structure allows for the JVM to perform many complex operations concurrently. From what we have observed, how the JIT optimization tasks are delegated to threads can have varying impacts on a Java program's overall run-time. The JIT optimizations we focus on primarily are those concerned with the *devirtualization* of method calls and *method inlining*, explained below [1].

In the Java language there exist class hierarchies of varying levels of abstraction. For example one might define a variable of the class `Animal` that references an object of its subclass `Fish`. Now assume we would like to call a method `swim` on this variable. Initially it remains unclear whether we should call the method under `Animal` or the method under its subclass `Fish`. The JVM solves this problem using a *virtual* method call [6].

A virtual method call is a process that looks up and executes the target method for a referenced object. In terms of the example above, the virtual method call of `swim` on our variable of class `Animal` would traverse the `Animal` class hierarchy until the target `swim` method of subclass `Fish` was located. Correspondingly, this is the method that is called.

While virtual method call look-up is a necessary process, it can be rather inefficient. In some cases, JIT can optimize for this inefficiency. When a virtual method call performs look-up to the same subclass consistently and repeatedly, a relationship between the virtual method call and that subclass is established. When the JVM recognizes such a relationship, the JIT can optimize the virtual method call by compiling it to jump directly to the subclass method. This process is called method devirtualization, and it eliminates the method look-up procedure entirely.

Devirtualization presents a vulnerability however, as there is no guarantee that the object we assign our variable to will remain consistent. Take, for example, our `Animal` variable. Let us say that its swim method call was devirtualized to jump to the swim method under the subclass `Fish`. Now, if we reassign our `Animal` variable to the subclass `Monkey`, the swim method will incorrectly jump to the swim method for `Fish`, presenting a class mismatch of methods. To prevent this from happening, a safeguard trigger is placed before the devirtualized method call. This trigger forces a recompilation of the devirtualized method call back to a virtual method call if we encounter a class mismatch [7].

Method inlining is the elimination of a method call altogether. For this process the JIT takes a given method call and replaces that call with the code it represents. Inlining removes the overhead of a method call and creates opportunities for other optimizations since a consecutive code sequence with no jumps to other parts of the program is easier to optimize. Constraints can be defined within the JVM on how much code can be inlined into any given method, and these restrictions can significantly impact the run-times of the Java programs. As with devirtualization, inlining can be reversed if the method being called has changed.

The JIT optimizations of devirtualization and inlining generally require a method "warm up" before their utilization. This warm up is measured in the number of calls to a given method. Once a certain method call threshold is reached, the method is considered *hot* and is scheduled for an optimization. In the HotSpot Server JIT, the method call threshold for becoming hot is 10000 calls, though both devirtualization and inlining have been known to occur prior to reaching this threshold in some cases [13] [8].

2.3 Internal Java Code Representation

The HotSpot Server JIT starts with the internal code representation of bytecode. Bytecode, as previously mentioned, is compiled Java code that allows for interpretation by the JVM. As the bytecode is interpreted and analyzed, it may be compiled into different internal code representations by the JIT for optimal run-time. One of these representations is the native code, which is the machine code compiled specific to the operating system on which the Java program is being run. Another representation is the *Intermediate Representation* (IR), which is used to represent the program as it is being executed in the JVM. The IR is based upon the graph of an expressive form called *Static Single Assignment (SSA)*.

SSA is a standard that constrains variables to exactly one assignment. That is, if you were to assign the integer `5` or the result of an operation `3+2` to an `int` variable x, x can not be used to store any further values. The advantage this form provides for IR is that it allows for a construction of a slightly modified version of a *Value Dependence Graph* called the *Sea of Nodes*. The Sea of Nodes structure is based upon the idea that nodes constitute values, such as the constant `5`. These nodes are graphed in terms of their dependency to each other, which is why SSA is important. Without SSA, nodes could exchange their values in-place, and that would make the graph algorithmically complex and inefficient to maintain.

The Sea of Nodes construct provides several benefits: global constants are recognizable as single nodes with a constant value, dead (unusable) code is easily isolated as nodes that no reachable part of the graph depends on, and future nodes can be optimally inserted into the structure [3, 10]. Sea of Nodes structure can consist of a large number of nodes for even a small program because separate nodes are generated for all intermediate results in a program.

3 Java Generics and Bound Narrowing

When programming in Java it is often useful to have a generic data structure which can utilize any one type of object. In such a case you can then specify the class of object the data structure uses in the declaration of each individual instance. *Generic types* are a feature in Java that allow just that. For example, when you want to create two `ArrayLists`, one of `Integers`, and one of `Strings`, Java allows you to make two instances of the same class, `ArrayList<T>`, that can use all of the methods defined in `ArrayList<T>`, but are restricted to storing objects of type `Integer` and `String` respectively. In this case

```
public Class ArrayList<T> { // class declaration ...}
```

is the generic type and `T` is the *formal type parameter*. In the actual declaration of your instance of `ArrayList<T>` you would replace `T` with the *actual type parameter* which in this example is either `Integer` or `String`:

```
ArrayList<Integer> intArrayList = new ArrayList<Integer>();
ArrayList<String> strArrayList = new ArrayList<String>();
```

Each of these `ArrayLists` is restricted to storing objects of the actual type parameter, `Integer` and `String` respectively. If an attempt is made to add an object of a type other than that of the actual type parameter then there is a compilation error.

Java's implementation of generic types uses *type erasure* to ensure backwards compatibility of generic types with previous versions of Java and increase the type safety of the program. When a program using generic types is compiled, the JVM using type erasure removes all type parameters from generic type declarations and inserts type casts in front of all objects that are returned from the data structure. This ensures that each element of the generic type is of the type given by the actual type parameter.

In the above example there is no restriction on the class you may use for the actual type parameter, however it is possible to impose such a restriction in your own generic types. Using *type bounds* you can, for example, create your own implementation of `ArrayList` that restricts the actual type parameter to objects that implement `Comparable` interface:

```
public Class ComparableArrayList<T extends Comparable>
```

Now suppose you would like to create a class `ArrayListInteger` that extends `ArrayList` but the parameter type is restricted to Integer. You can do so in a two different ways:

```
public class ArrayListInteger extends ArrayList<Integer>
public Class ArrayListInteger<T extends Integer>
                              extends ArrayList<T>
```

We refer to classes with more specific bounds than their superclass such as these as being *bound narrowed*. Bound narrowing allows for the creation of `ArrayListInteger` methods that are specific to `ArrayLists` of `Integers`, and due to type erasure these methods can assume that all elements in an instance of `ArrayListInteger` are of type `Integer`. In previous research we had observed that implementing a class in such a way

can lead to an increase or decrease in run times, or have no effect at all when compared to run times of a program that does not use generic types [2][12].

4 Code Examples and Testing Procedures

4.1 Test Code and Goals

Our examples use a copy of the class `HashMap<K,V>` in the Java Collections library. The class provides the functionality of a hash table and is parameterized over two types: `K` for keys and `V` for values. It implements an interface `Map<K,V>`, also generic in `K` and `V`. Both `K` and `V` have the default `Object` bound. In our study we extended the `HashMap` class to our own class `Narrowed` that narrows the type bounds for the key and the value to `Integer` and `String`, respectively:

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

Note that the narrowed class no longer has type parameters since the bounds are narrowed to concrete types. The class overwrites several methods of `HashMap`. We added methods to both classes to study effects of bound narrowing. We compared the runtimes to those of the same class extension, except with the default `Object` bound for `K` and `V`. We called this class `Generic`:

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

The generic class has the type parameters `K` and `V`, just like the `HashMap` class from which it inherits.

We create an instance of a `Narrowed` class that we refer to via its interface `Map`:

```
Map<Integer, String> map = new Narrowed();
```

Likewise we create an instance of a `Generic` class, also referred to via the interface:

```
Map<Integer, String> map = new Generic<Integer, String>();
```

In order to minimize memory management effects and prevent the program from running garbage collection (which would interfere with our program timing geared towards measuring inlining and devirtualization effects) we initialize our test hash tables with just four elements and iterate over these elements in a large number of loops.

In both classes we test multiple calls to a method `containsValue`. The method is as follows in `Narrowed` class. The method performs a linear search for a value in a table of hash table entries which contains key/value pairs.

```
public boolean containsValue(String value) {
    // some unimportant code removed

    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;
    }
```

The method with the same name in `Generic` class is analogous, but with the signature

```
public boolean containsValue(V value)
```

where `V` is the type parameter which at run-time is erased to `Object`. In our tests we compare the running times of a large number of calls to `containsValue` on the `map` variable for the bound-narrowed class `Narrowed` and the fully generic class `Generic`.

Multiple factors may create differences in running times between the `Narrowed` and `Generic` classes. `Narrowed` has more specific type information (the value parameter is known to be `String`) which makes it easier for the JIT to inline or devirtualize the `equals` method called on the parameter. This would make `Narrowed` faster. On the other hand, bound narrowing in `Narrowed` leads to an extra type check: when the method `containsValue` is called on `map` (the generic interface type), a run-time check is needed to make sure that the value is indeed a `String` as required for the narrowed form of the method which would lead to `Generic` class being faster. The JIT may compensate for either delay by an optimization, and success or failure of certain optimizations may in turn allow or prevent subsequent optimizations. Our project focuses on determining whether either inlining or devirtualization takes place.

Our tests run in the HotSpot 1.6.16 JVM. This JVM was used because it is the only JVM that we know of that manifests instability: the same program executed multiple times under the same conditions has different run times. Section 6 shows that we were able to attribute these time differences to success or failure of specific method call optimizations.

4.2 Inner Loop Method and On Stack Replacement (OSR)

A common way of measuring running time of a certain program feature (e.g. a method call) is to run a very large number of loop iterations that perform that operation over and over to get aggregate times that are precisely measurable. It is also important that the number of method calls is significantly higher than the JVM warm-up threshold (see Section 2.2) so the method gets optimized almost right away. Our tests follow this approach: we call the method in question 100,000,000 times. This results in program running times on the order of 10 sec. and a clear separation between times for different versions of the code. However, instead of just putting the testing loop directly into `main` or a method it calls, we separate out 10,000 iterations of the loop into a method that we call `innerLoop` that we then call 10,000 times (resulting in the same total number of method calls). The reason for this is to prevent an *on stack replacement* explained below.

JIT optimizations are done on a method-by-method basis. When a method is compiled or optimized in some other form, subsequent calls to the method will load its new code. However, this approach would not allow optimization of a method that is called in the beginning of the program and does not finish until the program's end (e.g. `main`). Since this method is constantly located on the program stack, there is no opportunity to replace its next call with an optimized version. The JIT handles this situation via *on stack replacement*, abbreviated as OSR: the method is replaced by its optimized version directly on the program stack while the program is being executed [5]. However, the OSR optimization works on fragments of the method code (such as `main`), and not on the entire method. This approach can lead to optimizations that are less efficient than a whole-method optimization would produce [4]. Our use of the `innerLoop` method guarantees that we observe full optimized methods, and not their OSR version.

5 Methodology

Xprof. The HotSpot JIT comes equipped with a profiler, called *XProf*. A useful feature of XProf is that it is embedded in the JVM. This allows XProf to have a fairly low profiler overhead [11]. When we ran our code with XProf, we found that run times of all program runs increased by a constant factor. This indicated that our observed program run-time patterns were identical to those observed without XProf enabled. This type of *observer effect* is highly preferable to that which results from running other profilers, such as HProf, with your Java program. The difference with HProf running is that not only do the program run times change, but the resulting patterns observed in run times are significantly altered [9].

Xprof allows us to detect inlining by measuring the amount of time a method is positioned on a program stack: if a method does not appear on the stack then it is likely to have been inlined. One has to be careful, however, because profilers may only monitor certain types of code representation. Among other useful information produced by Xprof is the time that the JIT spends in various modes, such as compiling. It is important to note that XProf operates by sampling run-time data rather than an aggregate analysis. This is a reason why XProf is able to maintain a low overhead, but it is also its limitation [11].

JVM Compilation Logs. The HotSpot JIT allows the user to specify an optional start-up flag `-XX:+LogCompilation`, which, when present, instructs the JVM to output a structured XML log file detailing compilation-related activity during the execution of a Java program in the HotSpot JVM[2]. These compilation logs provide us with a great deal of information relating to when and how the JIT enacts optimizations to Java code. Node counts in the Sea of Nodes internal representation, method call optimizations, and other useful information are all provided within the logs in a chronological[3] and mostly timestamped fashion. Even so, while LogCompilation is indeed a great resource it is not all encompassing. Relying upon it as a reference primarily can be dangerous, as we have en-

[2]Additionally the flag `-XX:+UnlockDiagnosticVMOptions` needs to be set to enable logging

[3]Chronological as per thread.

countered, because the true causes of observed run times and patterns can be absent from these logs.

While compilation logs have useful information, they often also contain a great deal of data irrelevant to the optimizations we would like to track. To help filter the useful data out into a human readable format, as seen in figure 4, we use XSLT files. XSLT files allow us to convert XML logs to HTML, isolating the items of interest in a preferable fashion. For example, if we were aiming to pull out all the optimization tasks that inlined methods, we could select only tasks that have non-zero inlined bytes. We also have control over what attributes of the items we can see and can highlight important information by color-coding.

Disabling Inlining A HotSpot flag we often work with in our research is `-XX:-Inline`, which sets the JIT to not inline any methods within the Java code. The advantage of using this flag is the ability to compare run-time and JIT optimization differences between inlined and non-inlined runs.

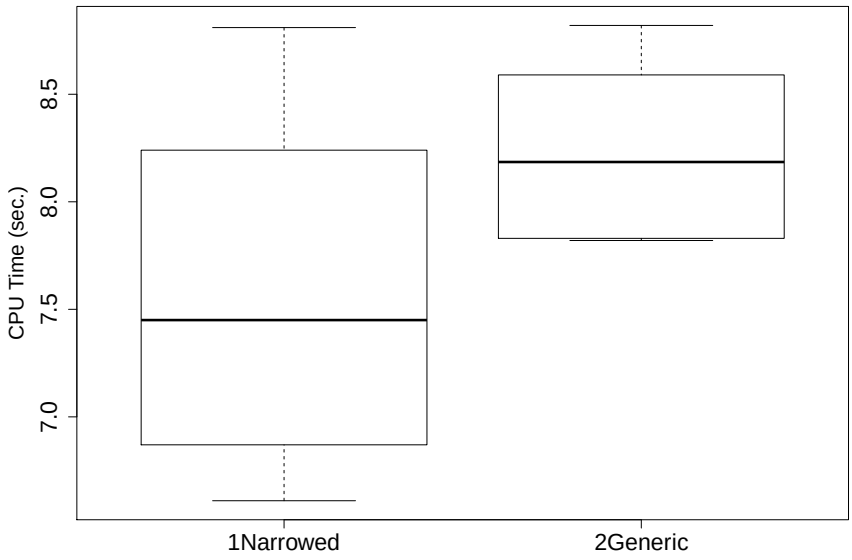

Figure 1: Test Results (Regular Run)

6 Observations and Results

6.1 Instability

The cornerstone of our research is the observed instability of run times for identical Java program test runs, see the graph in figure 1. To isolate the causes of this instability, we attempt to keep our testing environment as consistent as possible. In this effort we isolate our JVM process to only one CPU, and restart the JVM for each and every run.

In our observations we have been able to distinguish two types of instability:

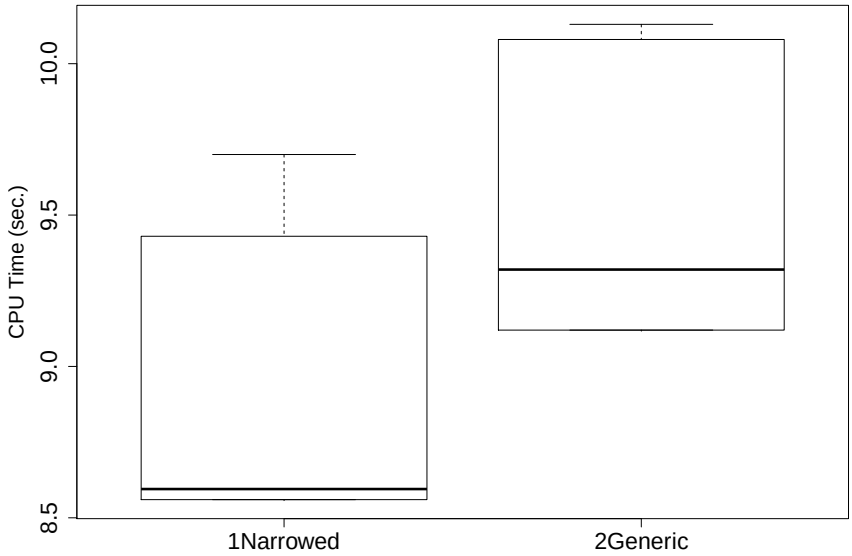

Figure 2: Test Results (No Inline Run)

1. Instability that correlates with differing optimization and compilation patterns.
2. Instability that does not appear to correlate with differing optimization and compilation patterns.

The run times that make up these instability graphs are almost always bi-modal, i.e. there exists a clear gap between the slower run times and the faster run times. We will refer to these two types of test runs as *slow runs* and *fast runs* respectively.

In figure 1 we have split the graph into two test run-time sets: the left set is the class `Narrowed` with a narrowed bound and the right set is the class `Generic` with a generic (`Object`) bound (see Section 4). The `Narrowed` results are bi-modal: five of the tests run in 6.8 sec. or close (fast runs), and the other five are at 8.1 sec. or slightly higher (slow runs). The `Generic` runs have less variation as the fastest are on the order of 7.8 sec. and the slowest are 8.8 sec. As we show later, the left set exhibits the first type of instability, and the right set exhibits the second kind. We also observe that the fast runs of `Narrowed` are significantly faster than `Generic`.

The set of results in figure 2 demonstrates test runs in which we have disabled inlining through the use of the `-XX:-Inline` flag. Note that the running times increased to between 8.5 sec. and 10 sec. in the absence of inlining. These results are useful because we can observe the effects of devirtualization in the absence of inlining, and observe that devirtualization alone is sufficient to make `Narrowed` run faster. Just like in the runs with inlining enabled, the `Narrowed` tests are faster than `Generic`. Both the left and the right set demonstrate a bi-modal distribution, but just like `Generic` runs in figure 1, differences in times do not seem to correspond to factors detectable by the tools that we use.

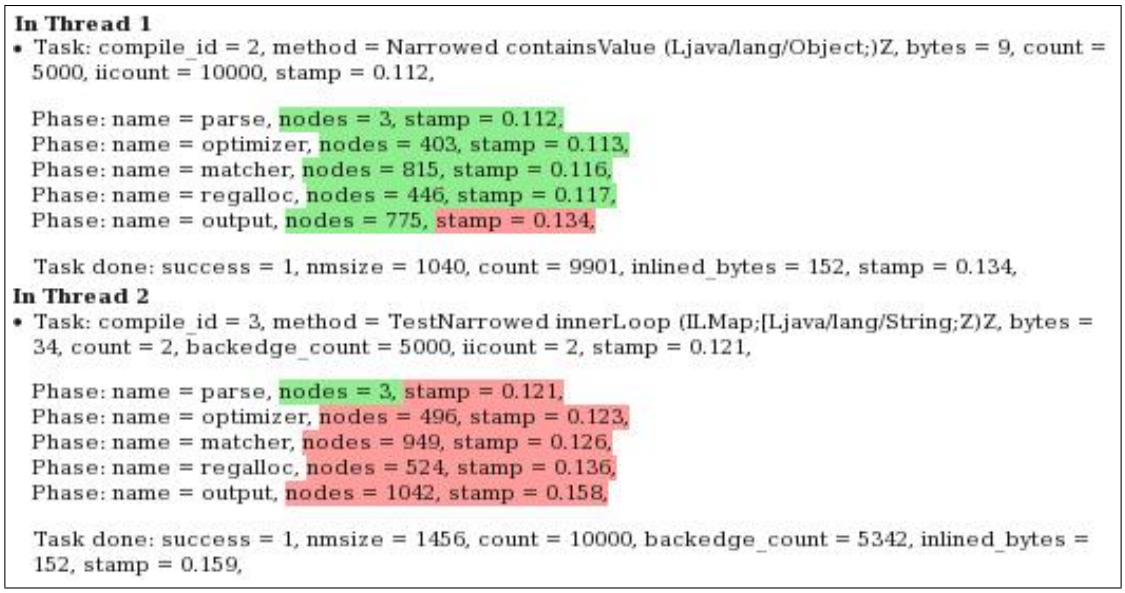

Figure 3: XSLT Filtered Compilation Log for Fast `Narrowed` Test Run

6.2 Compilation Logs and Effects of Threading

In this section we discuss the observed differences in logs between the fast and slow `Narrowed` runs. These are runs in which differences in run times correlate to differences in logs, i.e. they have the first kind of instability. One of the more useful ways of comparing the differences between slow and fast runs is by comparing their compilation logs side-by-side.

Observe figures 3 and 4. Here we have pulled out two JIT optimization tasks from compilation logs of two `Narrowed` test runs and filtered them using an XSLT file. Logs show several phases of optimization, including their corresponding counts of participating nodes in the Sea of Nodes representation of the code. As can be seen, the node counts differ for the `method = Narrowed containsValue` task (we show the common part of the two logs in green and the different parts in red). In our research, we have consistently observed these two different node count patterns to differentiate slow test runs from fast test runs. Interestingly, it has also been consistently observed that the presence of the first thread overlapping with the second thread correlates with faster test runs.

Yet another feature that distinguishes between fast and slow runs is present in the task that optimizes `innerLoop` method. Figures 6 and 7 show a fragment of the logs (formatted with a different XSLT file) for this task. In the fast run inlining of `containsValue` is successful, but in the slow run it fails with a message "already inlined into a big method". Note that the JIT makes several attempts to inline the method throughout its running time so there are other failed attempt at inlining in both the fast and the slow runs, making detection of inlining non-trivial.

There is also a correlation between fast runs and more time being spent in the *register allocation* (`regalloc`) phase. Figure 5 shows the complete (not filtered by XSLT) re-

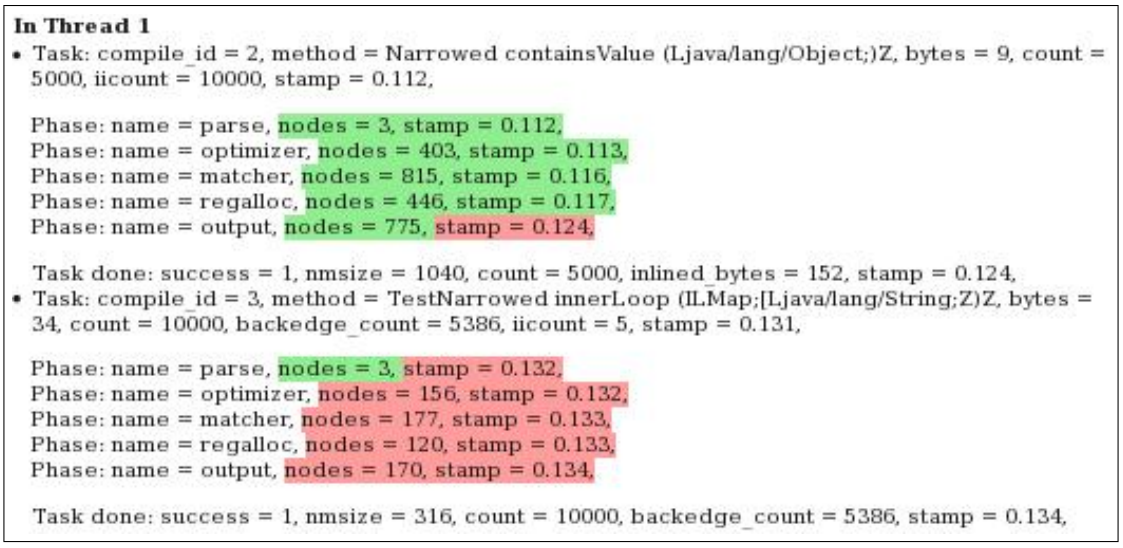

Figure 4: XSLT Filtered Compilation Log for Slow `Narrowed` Test Run

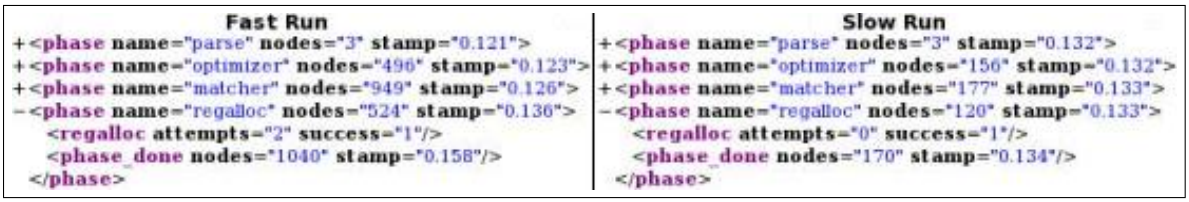

Figure 5: Raw Compilation Log: `regalloc` phase comparison of `Narrowed` Test Runs

galloc phase. Notice that the fast run corresponds to more attempts at register allocation and significantly more nodes involved. Fast runs also have a longer time spent in compilation phase in the profiling (xprof) logs. Our hypothesis is that fast runs spend more time on register allocation, leading to more efficient use of registers, and thus to faster runs. It also seems plausible that successful inlining provides a larger code block for the register allocation algorithm to work with, ultimately enabling a faster program. It is also worth noting that unstable `Generic` runs as well as those in which we disabled inlining have the same number of nodes in the internal representation at the same phases of the optimization, regardless of whether the runs are fast or slow. This indicates that the instability of these runs is of the second kind, as mentioned in Section 6.1, and the tools that we are currently using do not provide any insight into the causes of these time fluctuations.

6.3 XProf and Inlining

One of the most interesting test run observations we have been able to extract using XProf is the percentages of compiled code and interpreted code for class methods positioned on the JVM *execution stack*, i.e. being executed by the JVM. We show examples of this data for a fast and a slow run in figures 8 and 9 respectively. The slow run spends 77% time running the `containsValue` method (compiled to native code), whereas the fast run

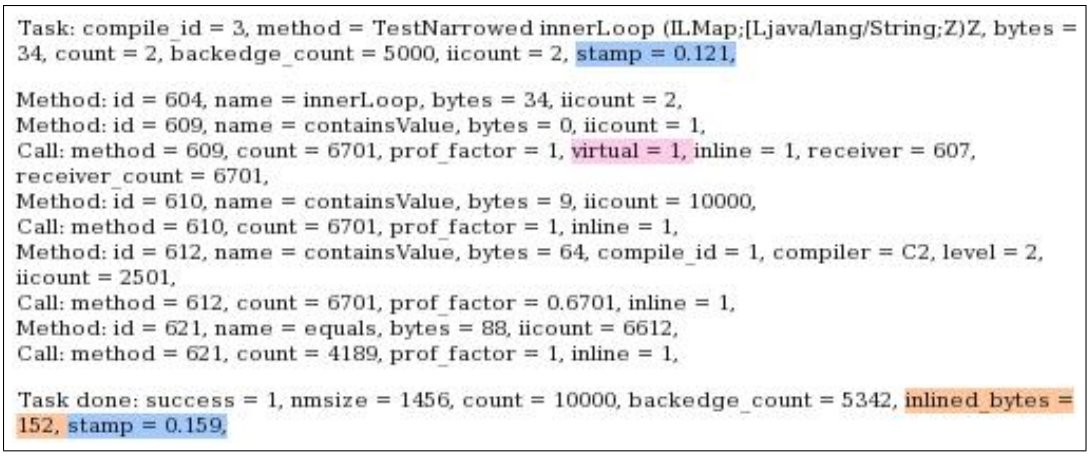

Figure 6: XSLT Filtered Compilation Log for Fast `Narrowed` Test Run

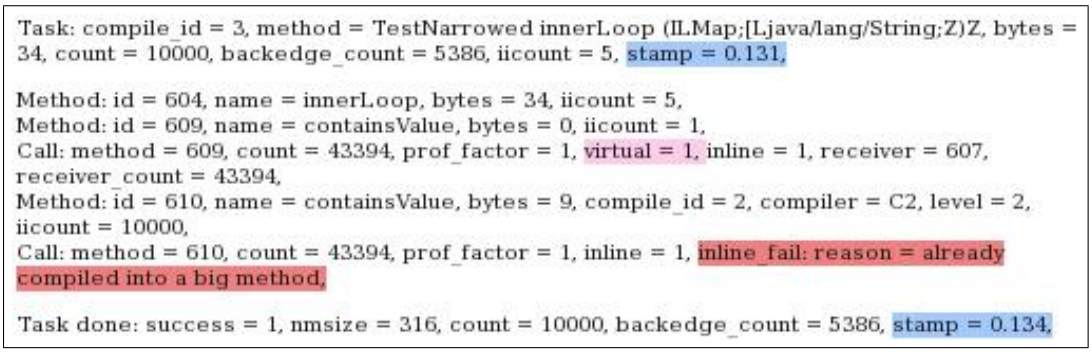

Figure 7: XSLT Filtered Compilation Log for Slow `Narrowed` Test Run

only spends 0.1% of its time in this method (presumably before the method gets inlined).

We also observe that `equals` does not appear in any of the Xprof logs (including the `Generic` runs) which points to a successful inlining of `equals` in all cases. Strangely, `equals` is also absent in the runs in which inlining is disabled. The best explanation we can come up with is that the `equals` method is stored in a different representation and may not be detectable by XProf or may be unaffected by `-XX:-Inline` flag. We do not, however, have a way of checking these hypotheses at the present moment.

6.4 Interpretation of Observations

Based on our observations we conclude that fast runs of `Narrowed` successfully inline the `containsValue` method, while slow runs of `Narrowed`and all runs of `Generic` fail to inline it. We also observe that fast `Narrowed` runs have a more efficient register allocation phase, even though it takes longer than that of slow runs. The efficiency of this phase is possibly enabled through successful inlining, as observed correlations would suggest, though we have yet to confirm this hypothesis. Based on the runs with inlining disabled we conclude that devirtualization of `containsValue` is by itself sufficient to

```
    Compiled + native   Method
99.1%   672  +     0    TestNarrowed.innerLoop
 0.1%     1  +     0    Narrowed.containsValue
99.3%   673  +     0    Total compiled
```

Figure 8: XProf Output for Fast `Narrowed` Test Run

```
    Compiled + native   Method
77.0%   676  +     0    Narrowed.containsValue
21.8%   191  +     0    TestNarrowed.innerLoop
98.7%   867  +     0    Total compiled
```

Figure 9: XProf Output for Slow `Narrowed` Test Run

make `Narrowed` runs faster than `Generic` runs. We also conjecture that inlining of `equals` takes place in all runs and seems to be unaffected by the `-XX:-Inline` flag.

7 Conclusions and Future Work

Our research goal was to determine whether certain methods were inlined and/or devirtualized in our examples, focusing particularly with those involving bound narrowing of Java generics. Throughout this research, we have explored a variety of tools for observing Java program behavior. Despite their unreliability and a lack of precise documentation for each of these tools individually, we were able to combine their functionality to successfully detect inlining and, with a somewhat lesser degree of confidence, devirtualization.

In our framework we were able to take advantage of fast and slow runs of identical Java programs. The methodology that we developed and confirmed using these unstable programs can also be extended to a wider range of programs, leading to a better understanding of the effects of bound narrowing and generic hierarchy in general.

Our future work includes applying the methodology to more generic examples as to explain other run-time behavior patterns associated with bound narrowing. This in turn would help software developers to choose efficient generics inheritance patterns for their applications. Another promising research direction is to extend our methodology to more modern JVMs, such as the most recent update of Java SE 6 and Java SE 7.

References

[1] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2000), OOPSLA '00, ACM, pp. 47–65.

[2] BLOCH, J. *Effective Java (Second Edition)*. Addison-Wesley, 2008.

[3] CLICK, C., AND COOPER, K. D. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst. 17*, 2 (Mar. 1995), 181–196.

[4] CLIFF CLICK. What the heck is OSR and why is it bad (or good)? *Blog entry, November 11 2011*.

[5] FINK, S. J., AND QIAN, F. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2003), CGO '03, IEEE Computer Society, pp. 241–252.

[6] GOSLING, J., JOY, B., AND STEELE, G. L. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[7] ISHIZAKI, K., KAWAHITO, M., YASUE, T., KOMATSU, H., AND NAKATANI, T. A study of devirtualization techniques for a Java Just-In-Time compiler. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2000), ACM Press, pp. 294–310.

[8] LINDHOLM, T., AND YELLIN, F. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

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

[10] MASON CHANG. Combining analyses, combining optimizations - summary. http://www.masonchang.com/blog/2010/8/9/sea-of-nodes-compilation-approach.html.

[11] ORACLE LABS. Profiling. http://wikis.oracle.com/display/MaxineVM/Profiling.

[12] SJOBLOM, I., SNYDER, T. S., AND MACHKASOVA, E. Can you trust your JVM diagnostic tools? In *MICS 2011* (2011), pp. 1–15.

[13] SUN DEVELOPER NETWORK. The Java HotSpot performance engine architecture. *Sun Microsystem* (2007).

[14] SUN DEVELOPER NETWORK. The Java HotSpot™server VM. *Sun Microsystem* (2008).

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