# Empirical Studies of Java Optimizations

Steve Caudill
University of Minnesota Morris
Morris, Minnesota 56267
Caud0004@morris.umn.edu

Elena Machkasova (Advisor)
Division of Science and Math
University of Minnesota Morris
Morris, Minnesota 56267
Elenam@morris.umn.edu

## Abstract

Traditionally, compilers perform a dual task: they transform a program from the source code (such as C or C++) to machine code, and also optimize the program to make it run faster. Common optimizations include constant propagation and folding, method inlining, dead code elimination, and many others. Java compliers are different from C or C++ compilers: most Java compliers transform Java source code into platform-independent byte code which is later executed by Java Virtual Machine (JVM), usually equipped with a Just-In-Time compiler (JIT) to compile byte code to native machine code on the fly. In this setup, program optimizations can be performed at two levels: by the compiler (while converting Java code into byte code) and by JVM when byte code is compiled to native code as the program is executed.

In this project, we investigate optimizations that are performed by the compiler, javac, and by JVM. We compare our test program efficiency with that of a non-optimized program in order to detect optimizations being performed on the programs, and to determine at which level they are performed. Our testing programs are specifically designed to detect individual program optimizations. Our research is a work in progress. We present the current results and discuss techniques for detecting optimizations and also for determining whether these optimizations are performed at compile time or run time.

## Introduction

Originally, Java was designed as a web language, with some of the goals being client-side portability and security concerns. Some of the security concerns involved never accessing memory that has not been allocated to the program, preventing array overflows and such. This mobility came from the program source code being compiled into byte code, or .class files, which could be interpreted by any computer. As a result, the first implementations of Java were slow. Currently, Java has become faster, partly due to optimizations performed on the actual programming code or byte code. Such optimizations can be performed by a Java compiler (such as javac) or at run time by Java Virtual Machine (JVM).

The motivation for our research is to determine which optimizations are performed by the compiler, and which are performed by JVM. The optimizations that we were looking for are fairly simple ones, such as method inlining. This is where the code inside some methods is substituted for the calls to that method, eliminating the need to use memory to call and load that method. We are looking for methods of detecting optimizations that have been done, as opposed to trying to measure the gains or efficiency of various optimizations.

The actions we took to detect these optimizations were also fairly simple, mainly involving small class files, each created to test whether or not a single optimization was performed. For example, to test method inlining we created a class where the main method called an empty method a large number of times inside of a loop. We measured the time before and after the loop, and compared the time difference to that in the case of an empty loop repeated the same number of times.

Some of the programs we tested and the results are listed in this paper, as well as background on Java, an explanation of why we used the methods we did, and our plans for future research.

## Overview of Java

Since Java was originally developed to be a web language, a lot of focus was on portability of code and making sure that the programs could be run and behave the same way on any computer. Thus, when Java programs are compiled, they are compiled into byte code, which is machine-independent. The byte code, stored as .class files, can be transferred from machine to machine, as each computer uses a program (such as Java Virtual Machine) to interpret the byte code. With the addition of this step, the run times of Java programs in general were longer than languages that compile straight into native code.

Just-In-Time (JIT) techniques developed in the 60's for languages like Lisp were adapted for Java, as a possible solution [1]. JIT compilers take the byte code and compile it into machine-specific native code. The point of this is to translate byte code to native code only once. A new technology, Java HotSpot, is similar to JIT in that it translates code, however it only translates frequently used code to save on translation time [3]. The code that is translated is selected based on profiling done by HotSpot.

## Compilation Models

The traditional compilation model is one where source code is compiled into native code, with optimizations being performed along the way. The code is then run by the machine. For Java, some optimizations must be postponed until the native code is being interpreted and run by the virtual machine, which would mean that all optimizations depending on these must be postponed as well. In particular, to take full advantage of constant propagation, method inlining is done first, and then constant propagation, but since inlining is done at run time, constant propagation is delayed until then. The reason that method inlining is done at run time is that Java allows methods to be overridden by classes that inherit them. Due to the changeable nature of many methods and classes, the compiler simply cannot optimize them since it doesn’t know if a later class might inherit and change the code. Final variables and methods are much more likely to be optimized by javac, as they cannot be overridden.

## Overview of Optimizations

Due to Java’s relatively long run times, a lot of emphasis has been placed on optimization. Optimizations start by analyzing basic blocks of code. Blocks of code are groups of code that are always run sequentially, with no jumps into or out of the middle. Local optimizations are performed within these blocks of code, whereas global optimizations first rearrange blocks, then combine them, if possible, and then run local optimizations on the blocks. As a general rule, local optimizations are such that they do not require an analysis of the whole program, but rather just the small part of code that can be optimized.

Most optimizations we are looking for are small-scale, and local, with the exception of method inlining, which is a global optimization but still does not require complete program analysis. Method inlining is when the code inside a short or frequently called method is substituted for the call to the method, so that memory is not required to load and perform the method each time it is called. Constant propagation is the process of substituting a value for a variable call, if the variable never changes. Propagation is often followed by constant folding, which replaces expressions with fixed values, such as 2+3, by their value. Dead code elimination is an optimization that allows a compiler or virtual machine to ignore code and variables that are never used.

## Research

### Goals of Research

The motivation for our research is to determine which optimizations are performed by the compiler, and which are performed by JVM. Our research treats JVM and javac as black boxes. We do not study the byte code generated by the compiler or the source code of JVM. Instead, we focus on developing techniques to detect if optimizations have been performed. Due to this focus, we are not trying to measure the efficiency or gain of optimizations, or find the best way to optimize java programs.

### Technical Details

The main software that our research deals with are Java Virtual Machine (JVM), javac, and Java HotSpot. Java HotSpot is a version of JVM that allows for optimization and performs several that we have detected. JVM can run programs in two different modes, server and client. The client mode is the default setting, however we have noticed that the server mode generally has better results, with shorter run times and more optimization being done. As we are treating JVM as a black box, we do not know the exact steps taken by the server and client modes. In the server mode, HotSpot performs more optimizations, which may result in longer time to perform them but shorter overall run time.

We are documenting the differences in effects of specific optimizations, and only looking at a small window of the program's total run time. This approach is quite challenging, in that the differences in time observed are often due not to optimizations we're trying to detect, but are possibly caused by optimizations related to loops or more efficient memory management or other unrelated factors.

Another option of HotSpot that we use is the –Xint flag. This flag disables HotSpot optimizations, and is useful in determining if the compiler has performed optimizations. Due to the disabling of optimizations, programs run with –Xint take significantly longer than those run without it.

### Systems Specifications and Platforms

On a Dell Optiflex GX270 computer, using Microsoft Windows XP Professional, we use javac version 1.5.0_01, as well as Java version 1.5.0_01, and version 1.5.0_01-b08 of HotSpot for both client and server versions. For Linux, we used javac version 1.5.0, Java version 1.5.0, Java 2 Runtime Environment build 1.5.0-b64, and HotSpot Client and Server build 1.5.0-b64.

### Techniques Used

In the process of our research, we used several techniques for detecting optimizations. The most common involves creating a non-optimized test case, and then creating a manually optimized comparison class. For this comparison class, we performed the expected optimizations on the source code. For method inlining, we eliminated calls to empty methods. For constant propagation, we replaced variables with constants. This is not exactly the same as machine-optimized code, since the optimizations would not produce source code. However, this manually optimized code allowed us a method of comparison to further confirm the hypothesis that time differences were indeed caused by the optimizations we tested for.

For measuring run times, we chose to time the programs starting after the class was created, before the loop that performed the method calls, and ended the timing right after the loop. The reason for this is that we are excluding the overhead of class creation, or delays due to printing. The reason we looped the method call so many times was to ensure that the code was used frequently enough for HotSpot to optimize it [2]. Another reason to loop is to ensure that the differences in run time would be detectable.

We run test programs using HotSpot in both the client and server versions, with and without the –Xint flag for both modes. We document results of all of the four testing cases, and list the results of each in the table (see Tables 1 and 2). The reason we give the results in a range, with the minimum being emphasized, is so that it is clear that there is a difference between two times. The minimum is the most important, because it shows the potential of the optimization under the best circumstances. Test runs that are slower could be affected by process scheduling or other external factors. Occasionally, we get results that are significantly larger, probably due to such factors, which would make an average a meaningless measurement. The range is included to demonstrate a difference in run times, which would be shown by non-overlapping ranges.

## Results of Research

### Method Inlining

Since method inlining is one of the fundamental optimizations, it was the first example we studied. The simplest case of this optimization is inlining an empty method, see Figure 1.1. To test this, we created a class that was comprised of a main method that called an empty method a large number of times. The results we received from running tests using this code are shown in Tables 1.1 and 1.2.

Our conclusions from testing this program are that both the server and client modes of HotSpot perform method inlining, however javac does not. Our reasoning for these conclusions is that the manually optimized version (Figure 1.2) was run with HotSpot optimizations disabled, and there was a significant difference in run times compared to the non-optimized testing class (also run with HotSpot optimizations disabled). This difference implies that javac does not inline the method. When optimizations were enabled, both server and client version had results equal to the manually optimized versions, which implies that HotSpot does inline the method.

We tried other variations of method inlining, such as using a non-static method (see Figure 1.3), and a method that increments a static counter (see Figure 1.4). In the case of the static counter we observed the same behavior as in the main example: the method was not optimized by javac, but seemed to be optimized by server and client modes of HotSpot. In the case of the object method neither javac nor HotSpot inlined the method.

```
public class EmptyMethod {

  public final static void main(String[] args) {
    long time1 = System.currentTimeMillis();

    int x = 0;
    while(x<999999999) {
      emptyMethod();
      x++;
    }

    long time2 = System.currentTimeMillis();
    System.out.println("The method took "
       + (time2 - time1) + " milliseconds");
    }

  public final static void emptyMethod() {
  }
}
```

Figure 1.1: Test Program for Method Inlining

```
public class EmptyMethod {

  public final static void main(String[] args) {
    long time1 = System.currentTimeMillis();

    int x = 0;
    while(x<999999999) {
//    emptyMethod();
      x++;
    }

    long time2 = System.currentTimeMillis();
    System.out.println("The method took "
       + (time2 - time1) + " milliseconds");
    }

//  public final static void emptyMethod() {
//  }

}
```

Figure 1.2: Manually Optimized Program for Method Inlining

```
public class EmptyMethodObject {

    public final static void main(String[] args) {

        int x = 0;
        EmptyMethodObject obj = new EmptyMethodObject();
        long time1 = System.currentTimeMillis();
        while(x<999999999) {
            obj.emptyMethod();
            x++;
        }

        long time2 = System.currentTimeMillis();
        System.out.println("The method took " + (time2 - time1) + "
milliseconds");
    }

    public EmptyMethodObject() {
    }

    public void emptyMethod() {
    }
}
```

Figure 1.3: Non-Static Test Variation for Method Inlining

```
public class EmptyMethodObject {

    public final static void main(String[] args) {

        int x = 0;
        EmptyMethodObject obj = new EmptyMethodObject();
        long time1 = System.currentTimeMillis();
        while(x<999999999) {
            //      obj.emptyMethod();
            x++;
        }

        long time2 = System.currentTimeMillis();
        System.out.println("The method took " + (time2 - time1) + "
milliseconds");
    }

    public EmptyMethodObject() {
    }

    //public void emptyMethod() {
    //}

}
```

Figure 1.4: Non-Static Manually Optimized Variation for Method Inlining

| Program | -server | -client | -server -Xint | -client -Xint |
|---|---|---|---|---|
| EmptyMethod.java non-optimized (see Figure 1.1) | **3**-4 | **2214**-2254 | **53212**-53251 | **50206**-50794 |
| EmptyMethod.java manually optimized (see Figure 1.2) | **3** | **2218**-2246 | **22699**-22738 | **20005**-20028 |
| EmptyMethod.java modifying a static variable, non-optimized | **267**-334 | **4791**-4878 | **83043**-83122 | **104818**-104952 |
| EmptyMethod.java modifying a static variable, manually optimized | **218**-305 | **4787**-4909 | **40001**-40011 | **37846**-37965 |
| EmptyMethodObject.java with a non-static empty method, non-optimized (see Figure 1.3) | **6** | **3326**-3329 | **56404**-56702 | **54602**-54650 |
| EmptyMethodObject.java With a non-static empty method, manually optimized (see Figure 1.4) | **3**-4 | **2239**-2275 | **22733**-22770 | **19985**-20000 |

Table 1.1 Empty method inlining. All times are measured in milliseconds (ms).

| Program | -server | -client | -server -Xint | -client -Xint |
|---|---|---|---|---|
| EmptyMethod.java non-optimized (see Figure 1.1) | **0** | **1781**-1797 | **41782**-42469 | **39328**-40531 |
| EmptyMethod.java manually optimized (see Figure 1.2) | **0** | **1797**-1812 | **16937**-18203 | **15000**-15782 |

Table 1.2: Empty method inlining measured on Windows XP. All times are in milliseconds (ms).

## Constant Propagation

Another fundamental optimization is constant propagation, where fixed-value variables in the source code are replaced by their values. The exact code for the examples is shown below, in figures 2.1, 2.2, and 2.3. In order to test for this optimization, we created a class consisting of an array of size one million, and used a loop to set every position in the array equal to the sum of two variables (x and y), which had fixed values. Afterwards, we chose a random position and output the value contained there.

The reason we used the random variable and the position check was to ensure that the loop indeed ran one million times, setting the value of each position to be equal to the sum of x and y, and so each time we ran the program to test it, we checked a different position.

Figure 2.1 shows the original test code. The results for it shown in the first row of the results table (see Table 2). Figure 2.2 shows the changes in the loop, which manually performed constant propagation on the source code, and the results are displayed in the second row of the results table. Figure 2.3 is the non-final version of the loop, which was done to test if the final status influenced whether or not javac performed optimizations.

Our results show that both the server and client versions seem to perform constant propagation on the test program, but the server version takes noticeably longer to run the program than the client version. When the variables were non-final, the –Xint flag (disabling HotSpot optimizations) run time became much longer than when the variables were final, which would imply that javac does indeed perform constant propagation as long as a variable is final, but does not optimize non-final variables.

By comparing the program to manually optimized code (see figure 2.2) we conclude that in this case javac performs constant propagation (to change the values of x and y to 2 and 3, respectively), constant folding (to replace 2+3 by 5), and dead code elimination (to eliminate the variables x and y entirely).

```
import java.util.*;

public class ConstantPropagation {

    public final static void main(String[] args) {
        int[] a = new int[1000000];

        long time1 = System.currentTimeMillis();

        for(int i=0;i < a.length;i++) {
            final int x = 2;
            final int y = 3;
            a[i] = x+y;
        }

        long time2 = System.currentTimeMillis();
        Random temp = new Random();
        int tempInt = temp.nextInt(1000000);
        System.out.println("At position " + tempInt + " the number is
" + a[tempInt]);
        System.out.println("The method took " + (time2 - time1) + "
milliseconds");
    }
}
```

Figure 2.1: Test Program for Constant Propagation

```
        for(int i=0;i < a.length;i++) {
            // final int x = 2;
            // final int y = 3;
            a[i] = 5;
        }
```

Figure 2.2: Manually optimized loop in ConstantPropagation.java.

```
        for(int i=0;i < a.length;i++) {
            int x = 2;
            int y = 3;
            a[i] = x + y;
        }
```

Figure 2.3: The loop with non-final variables in ConstantPropagation.java.

| Program | -server | -client | -server -Xint | -client -Xint |
|---|---|---|---|---|
| ConstantPropagation.java with final variables (see Figure 2.1) | **13**-14 | **7**-8 | **28**-29 | **27** |
| ConstantPropagation.java manually optimized (see Figure 2.2) | **13**-14 | **7**-8 | **28**-29 | **27**-28 |
| ConstantPropagation.java with non-final variables (see Figure 2.3) | **15**-19 | **8**-9 | **133**-146 | **102**-105 |

Table 2: Constant propagation and related optimizations for final and non-final variables. All times are measured in milliseconds (ms).

**Other Optimizations**

In addition to method inlining, constant propagation, constant folding, and dead code elimination, we experimented with the optimizations of tail recursion and string concatenation.

To detect tail recursion optimization, we wrote a program with a tail-recursive method that does not have a base case. To increase the chances of optimization, we chose a final static void method. We reasoned that if recursion is optimized, no stack frames will be generated for the recursive calls, and the program will either stop or behave as an infinite loop. If recursion is not optimized, the stack frames for the method will overflow the program stack. In the experiment, the stack overflow occurred in all four modes of running the program (server, client, and their combinations with –Xint). However, the number of stack frames was different in all of these modes, with the largest for the server without –Xint option. This is possibly due to an optimization that reduces the amount of memory allocated for a stack frame. We plan to continue exploring the optimization of tail recursion.

To check optimization of string concatenation, we created a program that was computing string concatenation in a loop (assigning it to a variable), and another one where the assigned value was the already concatenated strings. The two programs had different run times in all four cases, so we conclude that the optimization has not been performed in any case.

### Future Work

At this point we have developed techniques for detecting program optimizations and for distinguishing between those performed by the compiler and those performed by JVM (with the additional distinction between the client and the server modes). We also obtained results for some simple forms of method inlining, constant propagation and folding, and dead code elimination. Future directions of this research include:

- Experimenting with inlining longer, more complex methods, including all combinations of void and non-void, final and non-final, and static and non-static.
- Experimenting with other forms of dead code elimination, such as unreachable branches of conditionals.
- Continue experimenting with tail recursion optimization.
- Study other compilers, in particular those that convert a program directly to native code, such as GCJ.

## References

[1] Aycock, John. A Brief History of Just-In-Time. ACM Computing Surveys, Volume 25, Issue 2, 2003.

[2] Kumar, K.V. Seshu, When and What to Compile/Optimize in a Virtual Machine? March 2004. ACM SIGPLAN Notices, Volume 39, Issue 3, March 2004.

[3] Sun Microsystems, “The Java HotSpot Virtual Machine” http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html