McGill University
School of Computer Science
Sable Research Group

Profiling The Kaffe JIT Compiler

Sable Technical Report No. 2

Raja Vallée-Rai

February 11, 1998

Abstract

This report documents our attempts at instrumenting the Kaffe JIT compiler in order to estimate the execution cost of each bytecode instruction in the Java virtual machine.

1 Introduction

Sun Microsystems Inc.'s introduction of Java has generated much excitement in industry and academia. Not only does it tout several programmer friendly features such as dynamic linking and garbage collection, it is also a cross platform language. These features are not for free, however, for Java is several times slower than other languages such as C and C++. Because of its cross platform nature, Java source code is only compiled to bytecode for the Java Virtual Machine. To execute this bytecode, a Java Virtual Machine is then used to either interpret the code or compile it to native code.

Optimizing the Java language is quite different from traditional languages. First, there are two places where optimizations can occur: the bytecode itself or the execution of the bytecode (the virtual machine implementation.) Second, when optimizing the bytecode, the cost of each bytecode instruction is not fixed. It is dependent on the virtual machine you choose to execute the program on. We will see that the costs of the bytecode instructions can differ greatly. This means that although one can perform traditional optimizations like copy propagation and common subexpression eliminitation, these might not have any noticeable effect, unless of the costs of doing copies and recomputing expressions are noticeable.

The motivation for this project comes from our earlier attempts at optimizing Java bytecode. We failed at producing noticeable speed-ups. To help us understand why, our goal was to produce a profiling tool which estimates the cost of each bytecode instruction and the total amount of time spent executing each one.

This report presents how we modified Kaffe, the results that we obtained, and then the observations we drew to help our Java optimization strategy.

2 Profiling Kaffe

This project would not have been possible without the freely available Java virtual machine Kaffe[1], written by Tim Wilkinson. Kaffe is an Java implementation which runs under several platforms. In order to reduce the complexity of the project, we modified the program for only one of the platforms supported: Linux/x86. Kaffe can also be used as both an interpreter and a JIT (just-in-time) compiler. Because the JIT compiler is more efficient and probably represents a more typical virtual machine implementation, we chose to modify the JIT compiler version.

The execution times involved for bytecodes are on the order of nanoseconds, and so it is impossible to time them directly. Instead, we opted to approximate the execution cost by counting the number of native instructions each bytecode corresponds to. This section explains this strategy in detail.

[1]For your own copy, visit http://www.kaffe.org

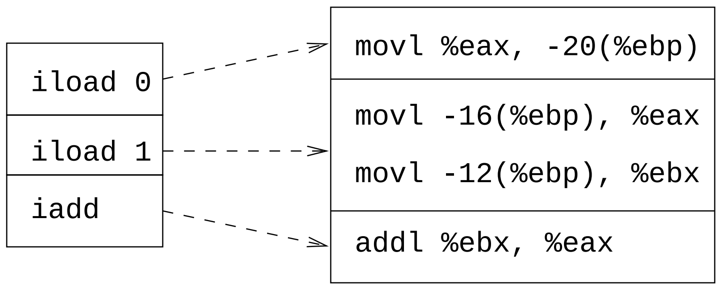

Figure 1: Illustration of the bytecode to native instruction conversion.

2.1 How Kaffe Works

The operation of the Kaffe JIT compiler is straightforward. When it encounters a method which has not been translated yet (referenced in a `invokevirtual` bytecode instruction for example), the runtime system translates it. Each bytecode gets translated as a fixed series of native instructions plus a few prologue and epilogue instructions for register loading or spilling. See Figure 1 for an illustration.

2.2 A Simple First Attempt

The method used to profile Kaffe is to inject native code into the translated method which records the number of times each bytecode instruction is executed and the cost incurred. This additional code is inserted before the native code for the bytecode. For example, if the bytecode instruction `iload 0` is normally converted by Kaffe into

```
movl %eax, -20(%ebp)
movl -16(%ebp), %eax
```

then the produced code would be

```
incl iloadcount
addl $2, iloadcost
movl %eax, -20(%ebp)
movl -16(%ebp), %eax
```

which increments the iloadcount and increases iloadcost by 2. Thus the average cost per use of the bytecode instruction `iload` is calculated as:

$$averageiloadcost = iloadcost/iloadcount$$

It is important to calculate these costs dynamically, because bytecode instructions are not always converted to the same sequence of native instructions. For example, 90% of the occurences of `iload 0` might be converted to one native instruction and the rest to two native instructions, so the average cost should be 1.1.

2.3 Problem 1: Accidentally Changing the Status Register

The first problem encoutered with this strategy is that these profiling instructions may interfere with normal program execution. For example, a previous bytecode may set the x86 status register, the next instruction may use that value, as in:

```
cmpl $0, %eax           % first bytecode instruction
incl if_cmpeqcount      % profiling instructions for next instruction
addl $1, if_cmpeqcost
jeq 12345               % second bytecode instruction
```

in this case the profiled version will execute differently because the `incl` and `addl` instructions set the status register as well. The solution to this problem is to wrap each profiling instruction with a pair of `pushf` and `popf` native instructions which save the status register and then restore it. The profiling instructions are then:

```
cmpl $0, %eax
pushf
incl if_cmpeqcount
addl $1, if_cmpeqcost
popf
jeq 12345
```

2.4 Problem 2: Bytecode Cost Differ Each Execution

The second problem faced is that the costs for an instance of a bytecode instruction may not always be the same. Although a particular bytecode may get translated into 10 native instructions, only half of those may be executed if they contain jumps or conditionals. Thus the cost should be 5 and not 10. For example, if the bytecode instruction

```
tableswitch(x)
{
    0 -> 100
    1 -> 200
    2 -> 300
}
```

gets converted into

```
pushf
incl tableswitchcount
addl $6, tableswitchcost
popf
cmpl $0, %eax
jeq 100
cmpl $1, %eax
jeq 200
cmpl $2, %eax
jeq 300
```

then it is incorrect to charge 6 native instructions for every execution of the `tableswitch` instruction, because it is not always guaranteed that the `tableswitch` statement will choose the last alternative and actually execute 6 native instructions.

The solution for this problem is to simply add the cost of each native instruction just before executing it, guaranteeing that the bytecode cost is increased by exactly the number of native instruction executed. The correctly produced profiled code is then:

```
pushf
incl tableswitchcount
popf
pushf
incl tableswitchcost
popf
cmpl $0, %eax
pushf
incl tableswitchcost
popf
jeq 100
pushf
incl tableswitchcost
popf
cmpl $1, %eax
pushf
incl tableswitchcost
popf
jeq 200
pushf
incl tableswitchcost
popf
cmpl $2, %eax
pushf
incl tableswitchcost
popf
jeq 300
```

Note how we must wrap each `incl` with `pushf` and `popf`.

2.5 Problem 3: Soft Calls

After having solved these two problems, the project would be complete if bytecode instructions were always converted into an explicit series of native instructions. Unfortunately, they are not, for sometimes a converted bytecode instruction generates a native `call` instruction which calls a C routine.

Since we are interested in evaluating the cost of every bytecode, and this "soft call" may potentially be executing hundreds of native instructions, we cannot ignore the costs of the routine. We need a method to evaluate the number of native instructions being executed in each routine. We considered two solutions:

Dynamic Disasssembler

Before any code for the compiler is executed, we make a call to a disassembler routine. In this routine, we disassemble every method which can be the target of a soft call, or the target of a call from within a soft call routine (and so on recursively). Once disassembled, we know the beginning and end of each instruction in memory, and so can inject profiling code which increases a counter before each native instruction is executed. This solution sounds nice in theory, but is wraught with difficulties:

1. Once the code is modified, there is no guarantee that the code size can be increased without bumping into some other piece of memory (and triggering a memory protection fault.) So a new block of memory needs to be allocated for the code.
2. Putting the routine at a new location in memory requires that every call to this routine be changed so that it calls the new address. This requires the dissassembling of every routine in memory and examining the instructions for a `call` instruction.
3. When disassembling a routine, the end of the routine is not automatically known. So, how does one know when to stop disassembling? Flow analysis needs to be performed in order to determine where the last instruction resides.

Needless to say, this method would be extremely painful to implement. Luckily for us, we implemented a much simpler solution:

Compile Time Modifications

Since we have the source code for the soft calls, we can simply make the compiler produce assembly for these routines and modify them directly. We wrote a simple tool (in Java!) which parses assembly files and then performs the annotation. Here is an example of the transformation:

```
main:
    pushl %ebp
    movl %esp,%ebp
    addl $16909060,100000
    addl %eax, 100000
```

becomes

```
main:
        pushf
        incl softCallCost
        popf
        pushl %ebp
        pushf
        incl softCallCost
        popf
        movl %esp,%ebp
        pushf
```

```
incl softCallCost
popf
addl $16909060,100000
pushf
incl softCallCost
popf
addl %eax, 100000
```

The tool avoids modifying labels, and uses the contents of the assembly directives to determine which instruction lies in which function.

Additional Issues

1. Every call to a soft call must be wrapped up like:

   ```
   movl $0, softCallCost
   call newObject
   addl $softCallCost, newSoftCallCost
   ```

   where you reset the counter, and then when the call returns, add the counter value to the total cost for this bytecode instruction.

2. Note that every soft call uses the same global variable to record its cost. This means that our profiler will work incorrectly if you have multiple threads running truly concurrently. For example, if two bytecodes start being executed at the same time, and they both make calls to soft calls, then they will be simulateneously increasing the global counter, and both record higher costs than expected.

3. Modifying the assembly of the soft call routines may generate costs which are too low. If the function called is `newObject` and the code for `newObject` is completely self contained, then the cost for `newObject` will be accurate. However, if `newObject` calls a function such as `sprintf` for which we do not have the assembly code, then the costs for that call will not be recorded. This does arise in a few cases, but is not too serious since the results obtained give us some lower bounds, and they are already quite high.

2.6 Limitations of our Tool

1. Our tool will generate incorrect results if multithreading occurs truly concurrently. This is not a problem for Kaffe, because under our platform, it simulates multithreading with a single thread.

2. The costs for the soft calls are just lower bounds because we can not modify all of the code executed when a soft call is made.

3. A serious limitation is that we do not take into account method invocation prologues and epilogues. For example, when a method is being invoked, after the `invokevirtual` bytecode instruction is executed, the method saves some registers. These saves are currently not being

charged to any bytecode instruction, but they should[2]. So all of the invocation bytecode instructions costs are probably 5-15 native instructions too low.

3 Results and Observations

This section presents some sample results produced by the profiler on synthetic examples, as well on real world programs. Here is an explanation of the terminology used in the figures below. At the end of this section, these results are put to use and some conclusions are drawn about optimizing Java.

Explanation of Terminology

- count: number of times this bytecode instruction was executed.
- direct cost: total # of native instructions executed for the given bytecode, excluding soft calls instructions
- soft cost: total # of native instructions executed for the given bytecode in soft calls
- total cost: total # of native instructions executed for the given bytecode
- direct cost/use: average # of native instructions executed per instance of this bytecode, excluding soft calls
- soft cost/use: average # of native instructions executed per instance of this bytecode in soft calls
- total cost/use: average # of native instructions executed per instance of this bytecode
- % total execution time: % of the total number of native instructions spent executing this bytecode

3.1 Sample Reults: Simple Invocation

Here is what our tool produces on a simple example which repeatedly invokes a method.
Java source code:

```
class Invocation2
{
    public static void main(String[] argv)
    {
        String s = "Hey!";
        int l;

        for(int i = 0; i < 10; i++)
```

[2]They should probably be charged to the `invokevirtual` instruction, although this is not entirely clear since the pushes of arguments before the `invokevirtual` are separate bytecode instructions, but could be considered part of the `invokevirtual` instruction as well.

```
            l = s.length();
    }
}
```

Java bytecode:

```
   0 ldc #1 <String "Hey!">
   2 astore_1
   3 iconst_0
   4 istore_3
   5 iload_3
   6 bipush 10
   8 if_icmpge 22
  11 aload_1
  12 invokevirtual #2 <Method int length()>
  15 istore_2
  16 iinc 3 1
  19 goto 5
  22 return
```

Results of profiler:

```
Instruction count for class: Invocation2

                     count             cost                      cost/use          %  total
                              direct     soft      total     direct  soft  total   execution
        iconst_0:        1 (       1        0         1)  [   1.00   0.00   1.00]  { 0.472%}
          bipush:       11 (      11        0        11)  [   1.00   0.00   1.00]  { 5.189%}
             ldc:        1 (       1        0         1)  [   1.00   0.00   1.00]  { 0.472%}
         iload_3:       11 (      22        0        22)  [   2.00   0.00   2.00]  {10.377%}
*        aload_1:       10 (      20        0        20)  [   2.00   0.00   2.00]  { 9.434%}
        istore_2:       10 (      10        0        10)  [   1.00   0.00   1.00]  { 4.717%}
        istore_3:        1 (       3        0         3)  [   3.00   0.00   3.00]  { 1.415%}
        astore_1:        1 (       1        0         1)  [   1.00   0.00   1.00]  { 0.472%}
            iinc:       10 (      20        0        20)  [   2.00   0.00   2.00]  { 9.434%}
       if_icmpge:       11 (      22        0        22)  [   2.00   0.00   2.00]  {10.377%}
            goto:       10 (      30        0        30)  [   3.00   0.00   3.00]  {14.151%}
*         return:        1 (       1        0         1)  [   1.00   0.00   1.00]  { 0.472%}
   invokevirtual:       10 (      70        0        70)  [   7.00   0.00   7.00]  {33.019%}

direct instructions: 100.00%   soft call instructions:  0.00%
```

Discussion

1. In this simple example, we see that no time is spent in soft calls. It is mostly spent doing the `invokevirtual` method. The cost of 7.00 for the `invokevirtual` method is a bit misleading, because as mentioned in the section on "Limitations of our Tool", it does not take into account the pushing and popping of registers on the method entry and exit. The cost is probably truly 15-20, and so most of the execution time, in this case, is spent invoking and returning.

2. We can also see that most of the instructions get translated into a few native instructions. This is comforting because it indicates that perhaps Java can be sped up considerably through proper JIT compilation.

3. Somewhat of an anomaly, the `goto` bytecode instruction requires 3 native instructions, whereas intuitively it would seem it that only one would be sufficient (a `jmp` instruction.) This result is probably due to the fact that register allocation in Kaffe is basic-block based, and at the end of a basic block (such as at the goto) the registers pushed onto the stack.

3.2 Sample Results: Array indexing

This result examines the cost of accessing arrays. Remember that in Java accesses to arrays are always bounds checked.

Java source code:

```
class Array
{
    public static void main(String[] argv)
    {
        int[] integers = new int[10000];

        for(int i = 0; i < 10000; i++)
            integers[i] = i;
    }
}
```

Java bytecode:

```
   0 sipush 10000
   3 newarray int
   5 astore_1
   6 iconst_0
   7 istore_2
   8 iload_2
   9 sipush 10000
  12 if_icmpge 25
  15 aload_1
  16 iload_2
  17 iload_2
  18 iastore
  19 iinc 2 1
  22 goto 8
  25 return
```

Results of profiler:

```
Instruction count for class: Array

                     count              cost                         cost/use           %  total
                              direct    soft     total     direct  soft   total    execution
        iconst_0:        1 (       1       0         1)  [  1.00   0.00   1.00]  { 0.000%}
          sipush:    10002 (   10002       0     10002)  [  1.00   0.00   1.00]  { 4.757%}
         iload_2:    30001 (   50002       0     50002)  [  1.67   0.00   1.67]  {23.781%}
*        aload_1:    10000 (   20000       0     20000)  [  2.00   0.00   2.00]  { 9.512%}
        istore_2:        1 (       3       0         3)  [  3.00   0.00   3.00]  { 0.001%}
        astore_1:        1 (       1       0         1)  [  1.00   0.00   1.00]  { 0.000%}
*        iastore:    10000 (   80000       0     80000)  [  8.00   0.00   8.00]  {38.048%}
            iinc:    10000 (   10000       0     10000)  [  1.00   0.00   1.00]  { 4.756%}
       if_icmpge:    10001 (   20002       0     20002)  [  2.00   0.00   2.00]  { 9.513%}
            goto:    10000 (   20000       0     20000)  [  2.00   0.00   2.00]  { 9.512%}
*         return:        1 (       1       0         1)  [  1.00   0.00   1.00]  { 0.000%}
*       newarray:        1 (       5     246       251)  [  5.00 246.00 251.00]  { 0.119%}

direct instructions: 99.88%   soft call instructions:  0.12%
```

Discussion

1. One of the features that Java supports is array bounds checking, and this example exposes the cost of this feature. In regular C, array indexing can be done typically with one native instruction (base + offset), so we see here that it is about 8 times more expensive in Java.

If your program consists mostly of array indexing, then it is clear that optimizing the use of arrays is a requirement for fast virtual machines.

2. The `newarray` bytecode instruction is quite costly. See the "Sample Result on Instantiation" for a discussion of instantiation.

3.3 Sample Results: Monitors

Java's main concurrency primitive is the *monitor*. Here we look at the costs associated with this synchronization primitive.

Java source code:

```
class Monitor
{
    public static void main(String[] argv)
    {
        Object obj = new Object();

        for(int i = 0; i < 100; i++)
            synchronized (obj)
            {
                int x = 5;
            }
    }
}
```

Java bytecode:

```
   0 new #1 <Class java.lang.Object>
   3 dup
   4 invokespecial #2 <Method java.lang.Object()>
   7 astore_1
   8 iconst_0
   9 istore_2
  10 iload_2
  11 bipush 100
  13 if_icmpge 49
  16 aload_1
  17 astore_3
  18 aload_3
  19 monitorenter
  20 iconst_5
  21 istore 4
  23 jsr 37
  26 goto 43
  29 astore 5
  31 jsr 37
  34 aload 5
  36 athrow
  37 astore 6
  39 aload_3
  40 monitorexit
  41 ret 6
  43 iinc 2 1
  46 goto 10
  49 return
```

Results of profiler:

```
Instruction count for class: Monitor

                     count              cost                     cost/use          %  total
```

```
                          direct      soft       total     direct   soft   total     execution
      iconst_0:       1 (        1          0          1)  [  1.00   0.00    1.00]  { 0.004%}
      iconst_5:     100 (      100          0        100)  [  1.00   0.00    1.00]  { 0.412%}
        bipush:     101 (      101          0        101)  [  1.00   0.00    1.00]  { 0.416%}
       iload_2:     101 (      202          0        202)  [  2.00   0.00    2.00]  { 0.831%}
*      aload_1:     100 (      200          0        200)  [  2.00   0.00    2.00]  { 0.823%}
       aload_3:     200 (      300          0        300)  [  1.50   0.00    1.50]  { 1.235%}
        istore:     100 (      100          0        100)  [  1.00   0.00    1.00]  { 0.412%}
        astore:     100 (      200          0        200)  [  2.00   0.00    2.00]  { 0.823%}
      istore_2:       1 (        3          0          3)  [  3.00   0.00    3.00]  { 0.012%}
      astore_1:       1 (        2          0          2)  [  2.00   0.00    2.00]  { 0.008%}
      astore_3:     100 (      100          0        100)  [  1.00   0.00    1.00]  { 0.412%}
           dup:       1 (        1          0          1)  [  1.00   0.00    1.00]  { 0.004%}
          iinc:     100 (      200          0        200)  [  2.00   0.00    2.00]  { 0.823%}
     if_icmpge:     101 (      202          0        202)  [  2.00   0.00    2.00]  { 0.831%}
          goto:     200 (      300          0        300)  [  1.50   0.00    1.50]  { 1.235%}
           jsr:     100 (      400          0        400)  [  4.00   0.00    4.00]  { 1.646%}
           ret:     100 (      200          0        200)  [  2.00   0.00    2.00]  { 0.823%}
*       return:       1 (        1          0          1)  [  1.00   0.00    1.00]  { 0.004%}
 invokespecial:       1 (        5          0          5)  [  5.00   0.00    5.00]  { 0.021%}
*          new:       1 (        4        347        351)  [  4.00 347.00  351.00]  { 1.445%}
*  monitorenter:    100 (      500       9625      10125)  [  5.00  96.25  101.25]  {41.677%}
*   monitorexit:    100 (      500      10700      11200)  [  5.00 107.00  112.00]  {46.102%}

direct instructions: 14.91%   soft call instructions: 85.09%
```

Discussion

1. These results indicate that the synchronization primitives `monitorenter` and `monitorexit` are roughly a hundred times more expensive than typical bytecode instructions. This is particularly significant, because some basic Java classes such as `java.util.Vector` make extensive use of monitors, and so the `java.util.Vector` class is much slower than it should be in a singly threaded environment.

3.4 Sample Results: Instantiation

Here we examine the costs associated with creating simple objects.

Java source code:

```
class Creation
{
    public static void main(String[] argv)
    {
        for(int i = 0; i < 100; i++)
            Integer x = new Integer(5);
    }
}
```

Java bytecode:

```
   0 iconst_0
   1 istore_1
   2 iload_1
   3 bipush 100
   5 if_icmpge 23
   8 new #1 <Class java.lang.Integer>
  11 dup
  12 iconst_5
  13 invokespecial #2 <Method java.lang.Integer(int)>
  16 astore_2
  17 iinc 1 1
  20 goto 2
  23 return
```

Results of profiler:

```
Instruction count for class: Creation

                      count                 cost                       cost/use           %  total
                               direct      soft      total      direct   soft   total    execution
        iconst_0:        1 (        1         0          1)  [   1.00   0.00   1.00]  { 0.003%}
        iconst_5:      100 (      100         0        100)  [   1.00   0.00   1.00]  { 0.267%}
          bipush:      101 (      101         0        101)  [   1.00   0.00   1.00]  { 0.270%}
         iload_1:      101 (      202         0        202)  [   2.00   0.00   2.00]  { 0.540%}
        istore_1:        1 (        2         0          2)  [   2.00   0.00   2.00]  { 0.005%}
        astore_2:      100 (      200         0        200)  [   2.00   0.00   2.00]  { 0.535%}
             dup:      100 (      100         0        100)  [   1.00   0.00   1.00]  { 0.267%}
            iinc:      100 (      200         0        200)  [   2.00   0.00   2.00]  { 0.535%}
       if_icmpge:      101 (      202         0        202)  [   2.00   0.00   2.00]  { 0.540%}
            goto:      100 (      300         0        300)  [   3.00   0.00   3.00]  { 0.802%}
*         return:        1 (        1         0          1)  [   1.00   0.00   1.00]  { 0.003%}
   invokespecial:      100 (      700         0        700)  [   7.00   0.00   7.00]  { 1.872%}
*            new:      100 (      400     34882      35282)  [   4.00 348.82 352.82]  {94.360%}

direct instructions:  6.71%   soft call instructions: 93.29%
```

Discussion

1. This simple example illustrates the enormous costs associated with creating a very simple object. Because this is a synthetic example, it may not be that expensive in a real application. Even if the cost is a thousand times that of a native instruction, as long as the instruction is not used too often, then it may not account to a significant amount of time. This is why it is important to start looking at some real applications, to see how frequently each bytecode instruction is used.

3.5 Sample Results on an Application: Jimplification

Jimplification is a code transformation on Java bytecode that I wrote as another project. It is heavily object oriented and takes a significant amount of time to execute.

Results of profiler:

```
Instruction count/cost information for entire execution:

                      count                 cost                       cost/use           %  total
                               direct      soft      total      direct   soft   total    execution
     aconst_null:    21421 (    21421         0      21421)  [   1.00   0.00   1.00]  { 0.018%}
       iconst_m1:     5013 (     5013         0       5013)  [   1.00   0.00   1.00]  { 0.004%}
        iconst_0:   227683 (   237179         0     237179)  [   1.04   0.00   1.04]  { 0.196%}
        iconst_1:   233416 (   233991         0     233991)  [   1.00   0.00   1.00]  { 0.193%}
        iconst_2:     8338 (     8338         0       8338)  [   1.00   0.00   1.00]  { 0.007%}
        iconst_3:      985 (      985         0        985)  [   1.00   0.00   1.00]  { 0.001%}
        iconst_4:     2300 (     2300         0       2300)  [   1.00   0.00   1.00]  { 0.002%}
        iconst_5:       65 (       65         0         65)  [   1.00   0.00   1.00]  { 0.000%}
        lconst_0:        3 (        6         0          6)  [   2.00   0.00   2.00]  { 0.000%}
        lconst_1:       83 (      332         0        332)  [   4.00   0.00   4.00]  { 0.000%}
        dconst_0:      660 (     1320         0       1320)  [   2.00   0.00   2.00]  { 0.001%}
          bipush:    57102 (    59566         0      59566)  [   1.04   0.00   1.04]  { 0.049%}
          sipush:    50255 (    50899         0      50899)  [   1.01   0.00   1.01]  { 0.042%}
             ldc:    98837 (   136358         0     136358)  [   1.38   0.00   1.38]  { 0.113%}
           ldc_w:     1010 (     1010         0       1010)  [   1.00   0.00   1.00]  { 0.001%}
           iload:   949569 (  1991753         0    1991753)  [   2.10   0.00   2.10]  { 1.646%}
           aload:   545254 (  1098815         0    1098815)  [   2.02   0.00   2.02]  { 0.908%}
         iload_0:    24275 (    48634         0      48634)  [   2.00   0.00   2.00]  { 0.040%}
*        iload_1:   241313 (   520980         0     520980)  [   2.16   0.00   2.16]  { 0.430%}
*        iload_2:   207185 (   436080         0     436080)  [   2.10   0.00   2.10]  { 0.360%}
```

*	iload_3:	405465	(800771	0	800771)	[1.97	0.00	1.97]	{ 0.662%}
	fload_2:	1980	(2640	0	2640)	[1.33	0.00	1.33]	{ 0.002%}
*	aload_0:	1344384	(2498871	0	2498871)	[1.86	0.00	1.86]	{ 2.065%}
*	aload_1:	369135	(729493	0	729493)	[1.98	0.00	1.98]	{ 0.603%}
*	aload_2:	260225	(493078	0	493078)	[1.89	0.00	1.89]	{ 0.407%}
	aload_3:	98268	(188839	0	188839)	[1.92	0.00	1.92]	{ 0.156%}
*	iaload:	84	(840	0	840)	[10.00	0.00	10.00]	{ 0.001%}
*	aaload:	162921	(1355719	0	1355719)	[8.32	0.00	8.32]	{ 1.120%}
*	baload:	40450	(440592	0	440592)	[10.89	0.00	10.89]	{ 0.364%}
*	caload:	224704	(2249375	0	2249375)	[10.01	0.00	10.01]	{ 1.859%}
*	saload:	10	(100	0	100)	[10.00	0.00	10.00]	{ 0.000%}
	istore:	261440	(356465	0	356465)	[1.36	0.00	1.36]	{ 0.295%}
	astore:	176354	(204619	0	204619)	[1.16	0.00	1.16]	{ 0.169%}
	istore_0:	2659	(3237	0	3237)	[1.22	0.00	1.22]	{ 0.003%}
	istore_1:	26964	(31452	0	31452)	[1.17	0.00	1.17]	{ 0.026%}
	istore_2:	36733	(76712	0	76712)	[2.09	0.00	2.09]	{ 0.063%}
	istore_3:	37157	(53974	0	53974)	[1.45	0.00	1.45]	{ 0.045%}
	astore_0:	52	(101	0	101)	[1.94	0.00	1.94]	{ 0.000%}
	astore_1:	39814	(62092	0	62092)	[1.56	0.00	1.56]	{ 0.051%}
	astore_2:	59554	(75723	0	75723)	[1.27	0.00	1.27]	{ 0.063%}
	astore_3:	41924	(63685	0	63685)	[1.52	0.00	1.52]	{ 0.053%}
*	iastore:	124	(992	0	992)	[8.00	0.00	8.00]	{ 0.001%}
*	bastore:	48772	(933635	5594946	6528581)	[19.14	114.72	133.86]	{ 5.394%}
*	castore:	85016	(991359	0	991359)	[11.66	0.00	11.66]	{ 0.819%}
*	fastore:	16013	(202885	0	202885)	[12.67	0.00	12.67]	{ 0.168%}
*	sastore:	10	(120	0	120)	[12.00	0.00	12.00]	{ 0.000%}
	pop:	10139	(0	0	0)	[0.00	0.00	0.00]	{ 0.000%}
	dup:	206906	(207322	0	207322)	[1.00	0.00	1.00]	{ 0.171%}
	dup_x1:	128855	(386565	0	386565)	[3.00	0.00	3.00]	{ 0.319%}
	dup2:	83	(249	0	249)	[3.00	0.00	3.00]	{ 0.000%}
	iadd:	277153	(287160	0	287160)	[1.04	0.00	1.04]	{ 0.237%}
	ladd:	83	(166	0	166)	[2.00	0.00	2.00]	{ 0.000%}
	isub:	87828	(88372	0	88372)	[1.01	0.00	1.01]	{ 0.073%}
	imul:	16951	(17008	0	17008)	[1.00	0.00	1.00]	{ 0.014%}
	fmul:	670	(2010	0	2010)	[3.00	0.00	3.00]	{ 0.002%}
	idiv:	1952	(5990	0	5990)	[3.07	0.00	3.07]	{ 0.005%}
	irem:	12403	(40389	0	40389)	[3.26	0.00	3.26]	{ 0.033%}
	ineg:	7399	(29596	0	29596)	[4.00	0.00	4.00]	{ 0.024%}
	ishl:	1795	(3060	0	3060)	[1.70	0.00	1.70]	{ 0.003%}
	ishr:	4487	(6826	0	6826)	[1.52	0.00	1.52]	{ 0.006%}
	iand:	17701	(17701	0	17701)	[1.00	0.00	1.00]	{ 0.015%}
	ior:	1803	(1803	0	1803)	[1.00	0.00	1.00]	{ 0.001%}
	iinc:	389325	(632276	0	632276)	[1.62	0.00	1.62]	{ 0.522%}
	i2f:	670	(1340	0	1340)	[2.00	0.00	2.00]	{ 0.001%}
*	f2i:	670	(4030	20770	24800)	[6.01	31.00	37.01]	{ 0.020%}
	f2d:	660	(1320	0	1320)	[2.00	0.00	2.00]	{ 0.001%}
	i2b:	39580	(119515	0	119515)	[3.02	0.00	3.02]	{ 0.099%}
	i2c:	4852	(9704	0	9704)	[2.00	0.00	2.00]	{ 0.008%}
	i2s:	1684	(7582	0	7582)	[4.50	0.00	4.50]	{ 0.006%}
*	lcmp:	3	(30	42	72)	[10.00	14.00	24.00]	{ 0.000%}
*	dcmpg:	660	(4620	27060	31680)	[7.00	41.00	48.00]	{ 0.026%}
	ifeq:	117531	(239483	0	239483)	[2.04	0.00	2.04]	{ 0.198%}
	ifne:	76019	(150762	0	150762)	[1.98	0.00	1.98]	{ 0.125%}
	iflt:	10209	(34772	0	34772)	[3.41	0.00	3.41]	{ 0.029%}
	ifge:	39102	(95990	0	95990)	[2.45	0.00	2.45]	{ 0.079%}
	ifgt:	102004	(258896	0	258896)	[2.54	0.00	2.54]	{ 0.214%}
	ifle:	8939	(18533	0	18533)	[2.07	0.00	2.07]	{ 0.015%}
	if_icmpeq:	63442	(185498	0	185498)	[2.92	0.00	2.92]	{ 0.153%}
	if_icmpne:	68271	(116883	0	116883)	[1.71	0.00	1.71]	{ 0.097%}
	if_icmplt:	319581	(761034	0	761034)	[2.38	0.00	2.38]	{ 0.629%}
	if_icmpge:	76659	(161565	0	161565)	[2.11	0.00	2.11]	{ 0.133%}
	if_icmpgt:	40549	(88474	0	88474)	[2.18	0.00	2.18]	{ 0.073%}
	if_icmple:	94464	(196771	0	196771)	[2.08	0.00	2.08]	{ 0.163%}
	if_acmpeq:	22536	(45072	0	45072)	[2.00	0.00	2.00]	{ 0.037%}
	if_acmpne:	2782	(4214	0	4214)	[1.51	0.00	1.51]	{ 0.003%}
	goto:	248542	(454098	0	454098)	[1.83	0.00	1.83]	{ 0.375%}
	jsr:	21140	(84560	0	84560)	[4.00	0.00	4.00]	{ 0.070%}
	ret:	21140	(42280	0	42280)	[2.00	0.00	2.00]	{ 0.035%}
	tableswitch:	3995	(71011	0	71011)	[17.77	0.00	17.77]	{ 0.059%}
*	ireturn:	140937	(323121	1821490	2144611)	[2.29	12.92	15.22]	{ 1.772%}
*	areturn:	122554	(373144	4399952	4773096)	[3.04	35.90	38.95]	{ 3.944%}
*	return:	193995	(267001	3461303	3728304)	[1.38	17.84	19.22]	{ 3.081%}
*	getstatic:	4298	(10375	74407	84782)	[2.41	17.31	19.73]	{ 0.070%}
*	putstatic:	316	(1346	10880	12226)	[4.26	34.43	38.69]	{ 0.010%}
	getfield:	1201619	(1232227	0	1232227)	[1.03	0.00	1.03]	{ 1.018%}
	putfield:	318921	(365113	0	365113)	[1.14	0.00	1.14]	{ 0.302%}
	invokevirtual:	249757	(2462293	0	2462293)	[9.86	0.00	9.86]	{ 2.035%}

```
   invokespecial:   117886 (  639353         0   639353)  [  5.42   0.00   5.42]  { 0.528%}
    invokestatic:    46585 (  378848         0   378848)  [  8.13   0.00   8.13]  { 0.313%}
*invokeinterface:    73959 (  879876 38007520 38887396)  [ 11.90 513.90 525.80]  {32.131%}
*            new:    41042 (  188113 15659689 15847802)  [  4.58 381.55 386.14]  {13.095%}
*       newarray:    14053 (   74528  2670241  2744769)  [  5.30 190.01 195.32]  { 2.268%}
*      anewarray:    10987 (   72567  5463590  5536157)  [  6.60 497.28 503.88]  { 4.574%}
     arraylength:    66937 (   66937         0    66937)  [  1.00   0.00   1.00]  { 0.055%}
*         athrow:        1 (       3         0        3)  [  3.00   0.00   3.00]  { 0.000%}
*      checkcast:    36787 (  235447  3220972  3456419)  [  6.40  87.56  93.96]  { 2.856%}
*     instanceof:    19427 (  118127  2023264  2141391)  [  6.08 104.15 110.23]  { 1.769%}
*   monitorenter:    44509 (  222545  4089461  4312006)  [  5.00  91.88  96.88]  { 3.563%}
*    monitorexit:    44509 (  199176  4553182  4752358)  [  4.47 102.30 106.77]  { 3.927%}
          ifnull:    41563 (   83128         0    83128)  [  2.00   0.00   2.00]  { 0.069%}
       ifnonnull:   116398 (  177192         0   177192)  [  1.52   0.00   1.52]  { 0.146%}

direct instructions: 24.73%   soft call instructions: 75.27%
```

Discussion

1. The creation of objects through the instructions `new`, `newarray`, and `anewarray` constitute 20.4% of the total program execution time. They do not dominate as in the previous synthetic example, but still constitute an important source of delay.

2. As we see, 75% of the total program execution time is spent in the soft calls. Soft calls are basically used by the complex instructions which require runtime support, such as `new` and `invokevirtual`. Thus we see that in this program execution, these special instructions clearly dominate. Traditional optimization techniques only work on the remaining 24.73%, and so it is not clear that optimizing this section would produce noticeable speed-ups.

3.6 Sample Results on an Application: Image Recognition

In order to get an estimate of where traditional C-like programs spend their time, I tried profiling an image recognition application written by Madeleine Mony for her Image Processing course.

Results of profiler:

```
Instruction count/cost information for entire execution:
     aconst_null:      433 (     433         0      433)  [  1.00   0.00   1.00]  { 0.001%}
       iconst_m1:      132 (     132         0      132)  [  1.00   0.00   1.00]  { 0.000%}
        iconst_0:    11586 (   11798         0    11798)  [  1.02   0.00   1.02]  { 0.029%}
        iconst_1:   140626 (  140642         0   140642)  [  1.00   0.00   1.00]  { 0.348%}
        iconst_2:      455 (     455         0      455)  [  1.00   0.00   1.00]  { 0.001%}
        iconst_3:      215 (     215         0      215)  [  1.00   0.00   1.00]  { 0.001%}
        iconst_4:     4982 (    4982         0     4982)  [  1.00   0.00   1.00]  { 0.012%}
        iconst_5:       65 (      65         0       65)  [  1.00   0.00   1.00]  { 0.000%}
        lconst_0:        3 (       6         0        6)  [  2.00   0.00   2.00]  { 0.000%}
        lconst_1:        1 (       2         0        2)  [  2.00   0.00   2.00]  { 0.000%}
        fconst_0:     2045 (    4074         0     4074)  [  1.99   0.00   1.99]  { 0.010%}
        fconst_1:     2013 (    4026         0     4026)  [  2.00   0.00   2.00]  { 0.010%}
        dconst_0:        2 (       4         0        4)  [  2.00   0.00   2.00]  { 0.000%}
          bipush:    20945 (   21640         0    21640)  [  1.03   0.00   1.03]  { 0.054%}
          sipush:    13154 (   13154         0    13154)  [  1.00   0.00   1.00]  { 0.033%}
             ldc:     2925 (    3589         0     3589)  [  1.23   0.00   1.23]  { 0.009%}
           ldc_w:      204 (     204         0      204)  [  1.00   0.00   1.00]  { 0.001%}
          ldc2_w:        1 (       1         0        1)  [  1.00   0.00   1.00]  { 0.000%}
           iload:  2480483 ( 5333768         0  5333768)  [  2.15   0.00   2.15]  {13.212%}
           fload:    12798 (   22434         0    22434)  [  1.75   0.00   1.75]  { 0.056%}
           aload:   126829 (  254890         0   254890)  [  2.01   0.00   2.01]  { 0.631%}
         iload_0:     1862 (    3832         0     3832)  [  2.06   0.00   2.06]  { 0.009%}
         iload_1:   241804 (  484127         0   484127)  [  2.00   0.00   2.00]  { 1.199%}
         iload_2:    18944 (   37536         0    37536)  [  1.98   0.00   1.98]  { 0.093%}
*        iload_3:   339612 (  902546         0   902546)  [  2.66   0.00   2.66]  { 2.236%}
```

```
         fload_0:     4026 (     4026         0      4026)  [  1.00    0.00    1.00]  { 0.010%}
         fload_2:        6 (        8         0         8)  [  1.33    0.00    1.33]  { 0.000%}
*        aload_0:  1166487 ( 2328603         0   2328603)  [  2.00    0.00    2.00]  { 5.768%}
*        aload_1:    66292 (  132574         0    132574)  [  2.00    0.00    2.00]  { 0.328%}
*        aload_2:    17828 (   39496         0     39496)  [  2.22    0.00    2.22]  { 0.098%}
         aload_3:   286039 (  627324         0    627324)  [  2.19    0.00    2.19]  { 1.554%}
*         iaload:   303740 ( 2430220         0   2430220)  [  8.00    0.00    8.00]  { 6.020%}
*         faload:      194 (    2029         0      2029)  [ 10.46    0.00   10.46]  { 0.005%}
*         aaload:   779234 ( 6480845         0   6480845)  [  8.32    0.00    8.32]  {16.053%}
*         baload:    63702 (  631410         0    631410)  [  9.91    0.00    9.91]  { 1.564%}
*         caload:    60272 (  602828         0    602828)  [ 10.00    0.00   10.00]  { 1.493%}
          istore:    69695 (  132855         0    132855)  [  1.91    0.00    1.91]  { 0.329%}
            ifne:     9816 (   19707         0     19707)  [  2.01    0.00    2.01]  { 0.049%}
            iflt:    56138 (  168129         0    168129)  [  2.99    0.00    2.99]  { 0.416%}
            ifge:     3346 (    6768         0      6768)  [  2.02    0.00    2.02]  { 0.017%}
            ifgt:     3762 (    7975         0      7975)  [  2.12    0.00    2.12]  { 0.020%}
            ifle:     2123 (    4258         0      4258)  [  2.01    0.00    2.01]  { 0.011%}
       if_icmpeq:    22423 (   44992         0     44992)  [  2.01    0.00    2.01]  { 0.111%}
       if_icmpne:    60680 (   63175         0     63175)  [  1.04    0.00    1.04]  { 0.156%}
       if_icmplt:   118450 (  238645         0    238645)  [  2.01    0.00    2.01]  { 0.591%}
       if_icmpge:   597780 ( 1196847         0   1196847)  [  2.00    0.00    2.00]  { 2.965%}
       if_icmpgt:      639 (    1340         0      1340)  [  2.10    0.00    2.10]  { 0.003%}
       if_icmple:     2066 (    4390         0      4390)  [  2.12    0.00    2.12]  { 0.011%}
            goto:   650937 ( 1250839         0   1250839)  [  1.92    0.00    1.92]  { 3.098%}
             jsr:       25 (      99         0        99)  [  3.96    0.00    3.96]  { 0.000%}
             ret:       25 (      50         0        50)  [  2.00    0.00    2.00]  { 0.000%}
         ireturn:     1489 (    2978     12677     15655)  [  2.00    8.51   10.51]  { 0.039%}
         freturn:     2013 (    3989         0      3989)  [  1.98    0.00    1.98]  { 0.010%}
*        areturn:     1984 (    8120    128660    136780)  [  4.09   64.85   68.94]  { 0.339%}
*         return:    16011 (   17330     76961     94291)  [  1.08    4.81    5.89]  { 0.234%}
*      getstatic:      910 (    3366     74407     77773)  [  3.70   81.77   85.46]  { 0.193%}
*      putstatic:       48 (     369      9062      9431)  [  7.69  188.79  196.48]  { 0.023%}
        getfield:  1198572 ( 1199064         0   1199064)  [  1.00    0.00    1.00]  { 2.970%}
        putfield:   130708 (  132316         0    132316)  [  1.01    0.00    1.01]  { 0.328%}
   invokevirtual:     4953 (   47941         0     47941)  [  9.68    0.00    9.68]  { 0.119%}
   invokespecial:    14961 (   93895         0     93895)  [  6.28    0.00    6.28]  { 0.233%}
    invokestatic:     5890 (   35899         0     35899)  [  6.09    0.00    6.09]  { 0.089%}
*            new:     7087 (   44863   2806781   2851644)  [  6.33  396.05  402.38]  { 7.063%}
*       newarray:      456 (    2546     83352     85898)  [  5.58  182.79  188.37]  { 0.213%}
*      anewarray:       28 (     197     12631     12828)  [  7.04  451.11  458.14]  { 0.032%}
     arraylength:    12055 (   12055         0     12055)  [  1.00    0.00    1.00]  { 0.030%}
*      checkcast:     1003 (    6018     93138     99156)  [  6.00   92.86   98.86]  { 0.246%}
*     instanceof:       15 (      90       885       975)  [  6.00   59.00   65.00]  { 0.002%}
*   monitorenter:      344 (    1732     32593     34325)  [  5.03   94.75   99.78]  { 0.085%}
*    monitorexit:      344 (    1401     35354     36755)  [  4.07  102.77  106.85]  { 0.091%}
* multianewarray:       29 (     285    616775    617060)  [  9.83 21268.10 21277.93]
                                                                                  { 1.528%}
          ifnull:       58 (     136         0       136)  [  2.34    0.00    2.34]  { 0.000%}
       ifnonnull:     2272 (    3180         0      3180)  [  1.40    0.00    1.40]  { 0.008%}

direct instructions: 87.86%   soft call instructions: 12.14%
```

Discussion

1. For this application, the impact of the soft call instructions is much less. Traditional optimizations would clearly work, as they operate on the 87.7% which is almost the entirety of the execution time.

3.7 Costs of Bytecodes

Based on the above results, we have constructed a table estimating the cost of each bytecode instruction. See Figure 2.

With this table we can classify the bytecode instructions into three catgories:

1. **Cheap (1-4 native instructions)** These are instructions such as `iadd`, `return`, `ifeq` which

Bytecode	Cost	Bytecode	Cost
invokeinterface	525	baload	10.9
anewarray	503	iaload	10.0
new	386	caload	10.0
newarray	195	saload	10.0
bastore	134	invokevirtual	9.96
instanceof	110	invokestatic	8.13
monitorexit	107	invokespecial	5.42
monitorenter	96.9	jsr	4.00
checkcast	94.0	ifeq	2.04
dcmpg	48.0	ifnull	2.00
areturn	39.0	ret	2.00
putstatic	38.7	ifne	1.98
f2i	37.0	goto	1.83
lcmp	24.0	ishl	1.70
getstatic	19.8	iinc	1.62
return	19.2	istore	1.36
tableswitch	17.8	astore	1.22
ireturn	15.2	if_icmpne	1.04
fastore	12.7	iadd	1.04
sastore	12.0	putfield	1.01
castore	11.7	iand	1.00

Figure 2: Costs per execution of Java bytecode instructions.

get converted into a few native instructions.

2. **Not So Cheap (5-50)** Array accesses with `iaload`, and uses of `tableswitch` are significantly more expensive than the **Cheap** instructions.

3. **Expensive (50+)** Byte code instructions such as `new` and `invokeinterface` are used to support the object oriented features of Java. They are extremely expensive.

3.8 Applying Results to Optimization

The main goal for this project was to understand the Java virtual machine's behavior in order to explain why our optimizations failed produce a speed-up. Based on our results, we can make the following observation:

Main Observation: The performance of the bytecode optimizations is extremely dependent on the virtual machine implementation and the type of program being optimized.

Consider the Kaffe virtual machine. Certain bytecode instructions are extremely expensive, and these are exactly the ones which are used in heavily object oriented programs. Thus, traditional optimization which inline methods and reduce the number of computational instructions (the **Cheap** or **Not So Cheap** ones) will have little effect. In a C-like program, such as the image processing application, which tend to not use these expensive instructions, these optimizations produce a significant speed-up, as is the case with C.

4 Conclusion

The goal of this project was to evaluate the cost of each Java bytecode instruction for the Kaffe virtual machine. We succeeded, although our tool has a few limitations regarding accuracy:

- The costs for the complex instructions which require runtime support are just lower bounds, due to the lack of source code for library functions.

- Method invocation costs are not completely recorded (the pushing and popping of registers at the beginning and the end of the method come for free.)

- We made the assumption that the # of native instructions roughly corresponds to execution time. This is true in most cases, except for certain classes of instructions such as floating point operations. These need to be handled in the future.

We were additionally able to produce the total time spent executing each bytecode. This information explains why our previous attempts at optimization failed. For heavily object oriented programs, the virtual machine spends most of its time with the expensive instructions, and so traditional optimization techniques which work on reducing the # of computational instructions (the simple instructions which require no runtime support) fail. For programs which are less object oriented, such as C-like programs, the optimizations are expected to succeed.

Possible avenues for future work are:

- Improving the accuracy of the profiler by working on the above mentioned points.
- Improving the tool's interface and transforming it into a full-blown profiler (see appendix B)

Although we have taken considerable care to make sure the results are correct, their validity will be confirmed when we perform optimizations and get the speed-ups predicted with this tool.

A Usage

We have simply added a few run-time flags to Kaffe:

```
Additional options for CS621 project:
   -profile                         Report instruction counts and costs
   -profileclass <class>            Report profile info only for the given class.
   -profilemethod <method> <signature> Report profile info for the given method.
                                    (-profileclass must also be used)
   -recordload                      Creating files KaffeMethodsTranslated.txt and
                                    KaffeClassesLoaded.txt which indicate which
                                    methods and classes were loaded by the JIT compiler
   -preload                         Pretranslate the classes and methods indicated
                                    by the -recordload option
```

Sample usage:

```
kaffe -profileclass Test3 -profilemethod main ([Ljava/lang/String;)V  Test3

    Produces profiling results for only the method main in the class Test3.

kaffe -profile Sort files1.txt files2.txt files3.txt

    Invokes the class Sort on files 'file1.txt', 'files2.txt', and 'files3.txt'.
```

Preloading and Pretranslating

In order to increase the accuracy of the tool, we have provided facilities which preload certain methods and classes before the program starts executing. This prevents the JIT compiler from recording extra soft call costs for a bytecode instruction which invoke a method compilation This problem arises because the compilation process shares code with the soft calls.

B Obtaining the Tool

The tool, in its present state, can be obtained from `~kor/public/KaffeProfilingTool.tgz` on the ACAPS filesystem. Please note that this program will only run correctly under Linux/x86.

C Also A Regular Profiler?

Although we were not planning to build a regular profiler, our tool almost does the job. All it needs is a bit more work for managing the results gathered (they are stored by method) and we will have a usable profiler based on instruction counts.