Optimizing Java Bytecodes

Michał Cierniak
Wei Li
Department of Computer Science
University of Rochester
Rochester, NY 14627
{cierniak,wei}@cs.rochester.edu

February 1, 1997

SUMMARY

We have developed a research compiler for Java `class` files. The compiler, which we call *Briki*, is designed to test new compilation techniques. We focus on optimizations which are only possible or much easier to perform on a high-level intermediate representation. We have designed such a representation, *JavaIR*, and have written a front-end which recovers high-level structure from the information from the `class` file.

Some of the high-level optimizations can be performed by the Java compiler which produces the `class` file. There is however a set of machine-dependent optimizations which have to be customized for the specific architecture and so can only be performed when the machine code is generated from the bytecodes, e.g, in a Just-In-Time (JIT) compiler.

We choose memory hierarchy optimizations as an example of machine-dependent techniques. We show that there is an intersection of the set of machine-dependent optimizations and the set of high-level optimizations. One such example is array remapping which requires multi-dimensional array references which are not present in the bytecodes and at the same time requires information about memory organization and the mapping of bytecodes to machine instructions.

We develop a set of optimizations for accessing array elements and object fields and show their impact on set of benchmarks which we run on two machines with a JIT compiler. The execution times are reduced by as much as 50% and we argue that the improvement could be even higher with a more mature JIT technology.

1 Introduction

The definition of the Java programming language includes both the definition of the programming language itself [1] and the definition of the virtual machine [2] which can run compiled Java applications represented in the form of *bytecodes* [2]. An application in the bytecode form can run on any computer with an implementation of the Java virtual machine.

A bytecode form of an application is distributed in `class` files which in addition to the bytecodes contain information required for linking and verification.

On most computers, efficient execution of Java bytecodes requires compilation to native machine instructions. The compilation can be performed off-line or on-demand, while the application is being loaded. The latter option — Just-In-Time (JIT) — is used in the majority of current implementations.

As the technology matures, advanced compiler optimizations are being proposed to speed up execution of Java applications (unless explicitly stated otherwise, by a "Java application" we mean either a stand-alone application or an applet). Many techniques used to optimize programs written in other programming languages can be successfully applied to Java programs, other techniques are unique to Java.

This paper concentrates on optimizations which rely on the knowledge of the target architecture, so they cannot be performed by the compiler which generates the `class` file, since the target machine is not known at that time. At the same time, the optimization techniques we consider cannot be easily performed on bytecodes directly and require the recovery of high-level representation of the code which is being optimized.

Briki is a compiler developed to research the issues of potential benefits of high-level optimizations for Java programs. Briki reads in a Java program distributed in the bytecode form, converts it into JavaIR (an intermediate representation used to represent Java programs in our compiler), performs the optimizations, and writes out the optimized code. We are primarily interested in the configuration of Briki which performs JIT compilation, i.e., a compiler which is integrated with the virtual machine and generates machine code for immediate execution.

The current implementation of Briki, which was used in the experiments presented in this paper, reads in a `class` file and writes the optimized code to another file in the form of Java source. We chose off-line compilation and Java source as the output form for the ease of debugging and better understanding of the quality of the recovered code. A JIT implementation of Briki which will integrate our compiler with kaffe [3], a publicly available JIT Java system, is under way.

Section 2 presents the organization of Briki including a brief discussion of *JavaIR*, our intermediate representation for Java programs. More details on JavaIR can be found in [4]. Section 3 discusses our approach to high-level structure recovery. Our optimization techniques are described in Section 4. We include two appendices to make our presentation clearer. Appendix A shows the recovery process on a sample, short method. Appendix B presents an example optimization process for one of our optimization techniques for a much simplified benchmark. Experimental results are given in Section 5.

2 Organization of briki

2.1 Implementation

The design of Briki was influenced by the following goals:

- The intermediate representation should be high-level enough that most of the information contained in the source program is easily accessible and reasoning about the program can be done in source-level terms. Section 4.1 gives an example of an optimization which requires a high-level intermediate representation.

- The compiler should be light-weight so that the overhead of integrating it into other applications would be minimized.

The extra work required to recover the high-level structure from the bytecode sequence can be only justified if it enables optimizations which would otherwise be impossible or very difficult. Data transformations for multidimensional arrays (see Section 4.1) are a good example for that as they would be impossible to perform without the notion of a multidimensional array reference. Since the bytecodes do not encode directly multidimensional array references, all such references are translated by Java compilers to a sequence of one-dimensional references. Before data transformations are even considered, we have to recover the original array references.

2.2 Intermediate representation

The intermediate representation is a syntax tree with a node for every Java class defined in the input. Every class node contains references to nodes for all interfaces, fields and methods defined for that class as well as some other information (access flags, a reference to the super class etc.). *JavaIR* is our implementation of this intermediate representation. An overview of JavaIR is presented in this section. More details on JavaIR can be found in [4]. Section 3 discusses some issues related to the problem of the recovery of the structure needed to build JavaIR from the bytecode representation of a program.

In the recovery process we attempt to achieve a structure which is as close to the Java source as possible. For the benchmarks we have used, our IR corresponds directly to a Java source form of the same benchmark. It is however possible that a `class` representation of an application either does not correspond to a legal Java source or it would be impractical to recover the source form. Section 3.5 is devoted to those problems.

Figure 1 shows a class hierarchy for a subset of the classes used in JavaIR. The version of Briki used for the experiments presented in this paper is implemented in Java, we are currently working on integrating Briki with kaffe, a JIT compiler.

We will use the following, very simple example to illustrate how JavaIR represents Java programs.

```
class simple {
   int i;
   public simple() {
      i = 0;
   }
} //simple
```

A complete JavaIR representation of class `simple` takes care of many details that are required to maintain the meaning of the program. The image gets even more complicated by the fact that some information is replicated so that it can be accessed quickly.

The JavaIR representation is not *exactly* the same as the original source. Instead, JavaIR corresponds closely to the code generated by the compiler. In our example, the code for the constructor contains *implicitly* two additional statements:

```
public simple() {
  super();    // implicit call
  i = 0;
  return;     // implicit statement
}
```

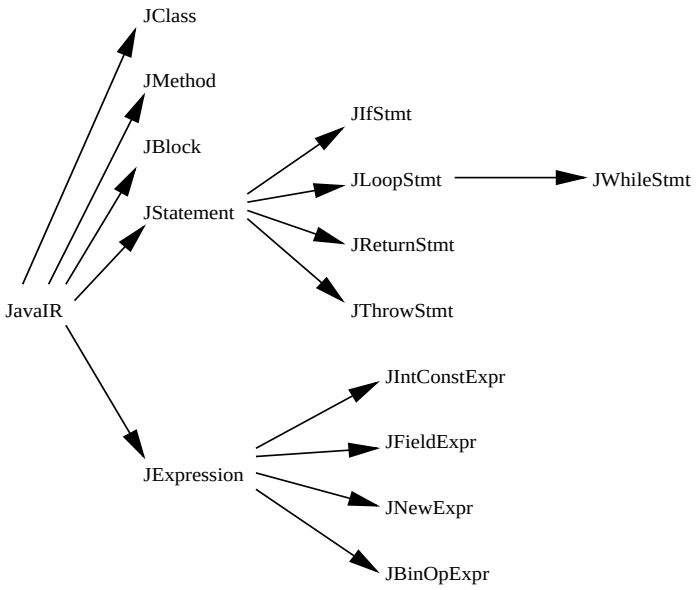

Figure 1: Class hierarchy of the nodes in the JavaIR intermediate representation (only selected expression and statement types are shown)

All implicit methods, statements and expressions are represented explicitly in JavaIR.

All nodes in the syntax tree are objects of some class derived from the class `JavaIR` (see Figure 1). The intermediate representation defines classes which are not derived from `JavaIR`, but none of them can be used as a node in the tree. A Java class is represented with an object of the class `JClass` which contains lists of objects representing fields, methods and interfaces. A method is represented with an object of the class `JMethod`. If the method is not native, this object contains a reference to an object of type `JBlock` which in turn contains a reference to the symbol table and a list of statements. A statement may contain other statements or expressions. Fields and interfaces follow the similar idea, but their representation is much simpler.

3 Recovering high-level structure

One of the important features of our compiler is the ability to recover high-level structure of a source Java program given its bytecode representation. Section 2.1 explains why it is necessary. In this section we discuss the process of that recovery as performed by our bytecode to JavaIR front-end. We start by giving a simple example of a bytecode sequence and the recovered source code. Note that our front-end actually recovers an internal representation which is not directly printable. However, since for our applications there is a straightforward mapping between the JavaIR described in Section 2.2 and a Java source, we simply show the recovered source code. The rest of this section shows a selection of more interesting problems which must be addressed for the recovery to take place. The problem of high-level structure recovery is very similar to the problem of decompilation. Previous work on decompilation addressed this issue primarily for reverse-engineering reasons. While our goal is different (we want to recover the high-level structure to enable

some compiler optimizations), we borrow from existing decompilation techniques in our design. Existence of many decompilation tools shows that high-level structure recovery from low level code is possible. Indeed there exists a decompiler for Java [5] which given a `class` file will produce its representation as Java source. Another interesting decompiler is described in [6]. In our work, we have identified the following problems.

- Converting a stack-based code without branches to a syntax tree. This is generally straightforward. The only problem that required extra attention was the use of temporary variables during the evaluation of a complex expression on the stack. Dataflow analysis is needed to ensure that flow dependencies are not violated. Section 3.2 discusses this issue.
- Identifying types of local variables. Again, dataflow analysis is used to disambiguate types of variables. The disambiguation may be needed due to one of the two reasons: the same variable may be used to store values of different types or assignments which are allowed in the bytecode must be changed to narrow down expression types (e.g., Boolean values are represented internally as integers, but in JavaIR, we want to represent them as Boolean values).
- Converting short-circuit operators into expression form. This is performed by a flow patterns recognition mechanism similar to the one used in [6]. Section 3.3 contains further discussion.
- Converting branches to high-level statements (loops, conditional statements, `break`, and `continue` statements. We have based our approach on the algorithm presented in [7]. Some modifications to the algorithm were necessary to more completely eliminate branches. Section 3.4 show our approach.

Our general approach is to first recover simple expressions and statements within basic blocks (bytecode sequences without branches). This process creates a *control flow graph* (CFG — see Section 3.3 for a simple example) with high-level constructs in its nodes. Then we convert CFG edges to structured control flow constructs using techniques described in Sections 3.3 and 3.4.

3.1 An example

Consider the following code fragment of a disassembled `class` file. It is not apparent what this sequence of bytecodes does.

```
 0 iload_1
 1 iconst_1
 2 if_icmpeq 10
 5 iload_1
 6 iconst_3
 7 if_icmpne 18
10 aload_0
11 iconst_1
12 putfield #4 <Field cond1.i I>
15 goto 23
18 aload_0
19 iconst_2
```

```
20 putfield #4 <Field cond1.i I>
23 aload_0
24 dup
25 getfield #4 <Field cond1.i I>
28 iconst_1
29 iadd
30 putfield #4 <Field cond1.i I>
```

One can of course understand this code with some effort. Most traditional compiler optimizations can operate on an intermediate representation which closely corresponds to the bytecode.

This paper deals with compiler algorithms which are best expressed in terms of high-level constructs like a for-loop or a multi-dimensional array reference. Neither of those two constructs is explicit in the bytecode which implements them with low level operations. In many cases it is however possible to recover a representation of a method with those high-level constructs.

Consider again the above sequence of Java VM instructions. Our compiler parses it and generates an equivalent JavaIR representation which can be printed as the following Java source fragment (as described earlier the internal JavaIR representation is not directly printable since it is a collection of Java objects with references to each other).

```
if(((a1 == 1) || (a1 == 3))) {
   this.i = 1;
} else {
   this.i = 2;
} //if
this.i = (this.i + 1);
```

Note that `a1` is used to represent the first argument to the method containing this sequence of bytecodes. The identifier used by the programmer in the source cannot be recovered.

3.2 Extracting expressions and simple statements

The design of Java bytecodes makes it relatively easy to recover expressions and simple statements (assignments and method invocations). Recovering statements which change the control flow requires more work since their implementation in the bytecodes uses branches which we do not want to use in JavaIR. We show in Sections 3.3 and 3.4 how to convert branches to structured control flow constructs.

To convert a sequence of bytecodes contained in a basic block to high-level expressions and statements, we symbolically execute each basic block using a temporary stack to emulate a Java virtual machine. For instance the following basic block from the example shown in Section 3.1

```
23 aload_0
24 dup
25 getfield #4 <Field cond1.i I>
28 iconst_1
29 iadd
30 putfield #4 <Field cond1.i I>
```

is converted to

```
this.i = (this.i + 1);
```

To see how the symbolic emulation works, consider the instruction `iadd` above. Processing of the previous instructions have placed two references at the top of the stack, one for the expression representing the constant 1 and another one for the expression `this.i`. Our front-end creates a new JavaIR object for integer addition and initializes its two operand fields to the two values from the top of the stack which are being removed from the stack. The new object is being pushed on the top of the stack.

This approach generally works for the application we looked at. The only interesting problem that must be solved is caused by using the stack for storage of variable and field values which have been overwritten during a calculation of a complex stack expression. Consider the following example

```
aload_0                              z
getfield #4 <Field cond1.i I>        z
aload_0                          x
iconst_1                         x
putfield #4 <Field cond1.i I>    x
aload_0                              z
getfield #4 <Field cond1.i I>        z
iadd                                 z
aload_0                              z
putfield #5 <Field cond1.j I>        z
```

The sequence marked with `x` is converted to

```
this.i = 1;
```

and the sequence marked with `z` is converted to

```
this.j = (this.i + this.i);
```

but the first occurrence of `this.i` on the right-hand side of the second assignment should have the value that the field `i` had *before* the assignment of 1, and the second occurrence should have the value that the field `i` had *after* the assignment of 1. Our solution keeps track of the flow of values and when necessary creates temporary variables to hold the values that the Java VM stores on the stack. For the example above, we recover a sequence corresponding to the following source fragment:

```
tmp = this.i;
this.i = 1;
this.j = (tmp + this.i);
```

3.3 Short-circuit operators

Consider again the bytecode sequence shown in Section 3.1. An alternative representation called a control flow graph (CFG) is shown in Figure 2. A CFG contains a node for every straight-line code sequence (a *basic block*) and an edge for every branch (a conditional branch results in two edges, an unconditional branch corresponds to a single edge). Nodes in Figure 2 are labeled with the numbers corresponding to the address of the first instruction of every basic block. Labels on the edges mark “true” and “false” branches.

Our recovery mechanism recognizes patterns in the CFG corresponding to different short-circuit operators in the manner similar to the technique described in [6]

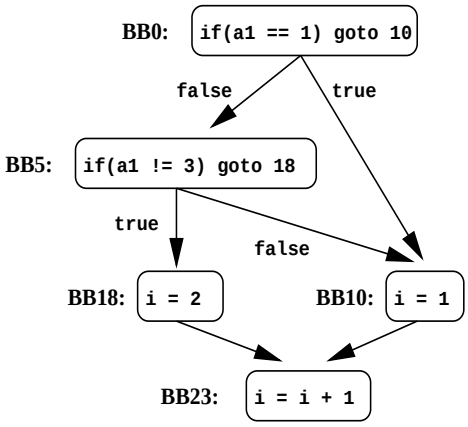

Figure 2: A control flow graph for a short-circuit operator

3.4 Goto elimination

This step is based on the algorithm described in [7]. We have extended this algorithm to eliminate branches in some cases in which the original algorithm failed. Nevertheless, for some CFG's we are not able to completely eliminate branches. For reducible flow graphs, it is always possible to completely eliminate branches, but sometimes doing requires inserting new loop statements or duplicating code. While those approaches would be acceptable if our goal was to recover legal Java source for the widest possible range of `class` files, adding new loops or duplicating code would not help us in any high-level compiler optimizations. Therefore, we decide to allow branches in JavaIR at the cost of not being able to optimize those programs with the current, preliminary version of Java (since our compilation process involves compiling Java source generated from JavaIR). Branches will not however be a problem in the JIT version of Briki which will be the next step of our research.

1. Collapse short-circuit operators.
2. Find natural loops.
3. Insert repeat nodes.
4. Find dominators, heads and follow sets.
5. Expand loops.
6. Call `getform`.
7. Call `addbranch`.
8. Clean the graph (push loop termination conditions into loop headers, unify types, etc.)

3.5 Difficulties in structure recovery

The current implementation of Briki parses a `class` file, represents it in JavaIR, performs the desired optimizations and prints the optimized source to a file. We later use a standard

compiler to convert the resulting source file to the bytecode form. The off-line process was chosen by us for the ease of debugging and more flexibility experimenting with new optimization techniques. Ultimately, we want to build a JIT version of Briki which would perform the same optimizations on JavaIR, but it would directly generate machine code from the optimized JavaIR rather than generating the source first.

For the `class` files we looked at so far, it is possible to efficiently recover a JavaIR form of those applications which can be printed out as legal Java source. It is however possible to create `class` files for which this process would be expensive or would not yield any benefit for our optimizations techniques. It is important to stress here that for our optimizations we are interested in recovering a *high-level structure*. Some bytecode sequences do not have such structure and although it would be possible to for instance eliminate all branches and generate legal Java source for such programs, the *structure* created by the branch-elimination process would be artificial and would not enable any additional high-level optimizations.

We also note that it is possible that a given application distributed in the `class` file form would not be representable as legal Java source (one such example is given in Section 13.4.6 of [1]). While this would cause problems for the current implementation of Briki, it would not affect the JIT version of our compiler.

3.6 Recovery algorithm

3.6.1 High-level overview

We sketch here the process of high-level structure recovery. We assume that all data structures for classes and class members have been initialized.

The algorithm performs the following steps for every non-native method.

1. Scan all opcodes and mark basic blocks.

2. Parse the exception table (see Section 4.7.4 of [2]) to mark portions of the bytecode stream handled by specific handlers and to mark the handlers themselves.

3. Create a control flow graph (CFG) for the method. The CFG has at least one *entry node* — for the main entry to the method — but may also contain additional entry nodes for every exception handler of that method.

4. Structure the CFG by identifying short-circuit operators and creating high-level control flow nodes as described in Sections 3.3 and 3.4 accordingly.

5. Starting from the entry nodes, visit all nodes of the CFG and interpret the code of each of them. This step creates the JavaIR nodes for every basic block. Every node of the structured CFG has two lists of JavaIR nodes: statements and expressions. The expressions are needed for values passed on the Java VM stack across basic block boundaries. The values left on the stack at the end of a basic block are being used in the successor nodes as initial stack content.

6. The information extracted in the previous two steps is used to create the complete JavaIR form for this method.

4 Optimizations

This paper describes only one group of optimizations: memory locality optimizations by data remapping. We expect that the benefits of those optimizations will be observed in addition to other optimizations performed by good JIT compilers.

Locality of memory references is extremely important on modern machines which have processors much faster than main memories. On those machines a high cache hit ratio may significantly increase execution speed. Compiler optimizations for improving locality have been studied extensively and many commercial and research compilers use those techniques. Strict definition of Java, especially its notion of *abrupt completion* [1] makes some of the existing techniques less effective for Java programs than for programs written in languages like C or Fortran.

Loop transformations belong to the category of optimizations which while extremely effective for a wide range of computationally intensive programs, are more difficult to use in optimizing compilers for Java. Java requires that the execution of a loop nest is performed in a specific order. Fortran and C have of course the same requirements, but due to the rich exception mechanism in Java, it is much easier to write a program in Java which will depend on the order of loop execution. In other languages, it usually sufficient to check if the transformed loop nest does not violate dependence relations defined by the original loop nest. In Java this may not be sufficient, because in the case of an abrupt completion of the loop nest, the exception handler may depend on partial results being generated in a specific order. Detecting that this is not the case puts an extra burden on the compiler.

Data transformations are an alternative to loop transformations. Data transformations have the advantage that they do not change the order of execution (except for subscript evaluation order which can be dealt with as described in Section 4.3.2). Which makes checking if a data transformation would change the meaning of the analyzed piece of code easier than a similar legality check for loop transformations. Applying data transformation requires other checks—ensuring that transformed arrays are rectangular (Section 4.3.1) is the most important one. In our view those checks can be performed without much effort for traditional, scientific codes. For common cases of array-based scientific applications both loop and data transformations are equally powerful. We show in [8] that a unified data and loop transformation framework is needed for best performance.

We focus here on the effect of Java semantics on data layout optimizations which improve memory hierarchies utilization. We have previously described similar optimizations for programs written in C and Fortran [8]. Changing the data layout for those languages is very difficult since both C and Fortran define the layout of their data structures and any changes may be attempted only after compiler analysis which ensures the legality of a data transformation. Such analysis is often very complex and sometimes fails inhibiting any potential optimizations.

Changing the layout of data structures is much easier in Java since the bytecodes do not use the concept of an address and cannot access, say, a field in an object by specifying an integer offset from some base pointer. Similarly, pointer arithmetic cannot be used to access elements of an array.

4.1 Array layout optimizations

Cache locality can be often significantly improved by changing the layout of array elements in memory so that for a given access pattern, array elements are more likely to lie in the cache which can be accessed much faster than main memory.

Our array layout optimizations optimizations rearrange multi-dimensional arrays in the context of nested loops. To perform any of those, we first have to recover multidimensional array references and for-loops from the bytecode representation. See [8] for a more comprehensive discussion of array and loop transformations.

4.2 Field layout optimizations

Many Java programs create data structures built with objects. Accesses to such objects can be made much faster if fields which are likely to be accessed together are placed in consecutive memory locations so that they are likely to reside in the same cache line. This optimization increases the likelihood that even if the access to the first field missed in the cache, the access to the second field will be a cache hit.

4.3 Legality of code and data transformations

Before we apply our transformations, we have to ensure that they do not change the meaning of the program. There are three issues we have to deal with for our transformations.

- Arrays we remap must be rectangular.
- The order of evaluation of a remapped array reference must be the same as for the original expression (see Section 15.12.1 of [1]).

The rest of this section gives more details about each of those three potential problems.

4.3.1 Array shapes

Array remappings used in our data and code transformations framework [8] can be applied only to multidimensional array which are *rectangular*. In a rectangular array all elements in a given dimension have the same size. Compiler-supported arrays in languages like C, C++ or Fortran are rectangular (of course a programmer may implement non-rectangular arrays explicitly by linearizing the logical structure in a one-dimensional array). Multidimensional arrays in Java are more general since they are treated as arrays of arrays.

For our optimizations, we check the array was allocated using the `multinewarray` opcode and that all accesses to that array use all dimensions. This approach is more conservative than necessary, but it is very fast and completely adequate for scientific codes.

4.3.2 Array expression evaluation order

The definition of Java specifies that a multidimensional array reference must be evaluated from left to right. Our simplest remapping switches two dimensions thus changing the evaluation order.

We solve this problem by checking if subscript expressions have potential side effects (note that a possibility of an abrupt completion is a potential side effect). Those subscript expressions that may have potential side effects, are being evaluated in the original order with the results being stored in local variables rather than being stored on the stack directly.

5 Experiments

We perform our optimizations using the kaffe JIT compiler [3]. To test the impact of the optimizations on machines with different memory and processor characteristics, we run our experiments on two computers running Linux: one is equipped with a 66MHz Intel486 processor, the other one has a 200MHz Pentium Pro processor.

5.1 Array layout optimizations

We test our optimizations on two programs ported by us directly from their Fortran versions included in the SPEC CFP92 [9] benchmark suite: mxm and vpenta.

5.2 Field layout optimizations

We show field layout optimizations on two benchmarks.

- A bibliography program keeps a collection of entries representing bibliographical references. We measure the times to perform two counting operations: by author and by year. We decided to time two operations accessing different fields to reduce the possibility that optimizing for one operation will decrease performance of another operation.
- An implementation of the Bellman-Ford algorithm for calculating shortest paths in a graph.

5.3 Results

We present normalized running times for our set of five benchmarks in Figures 3 and 4.

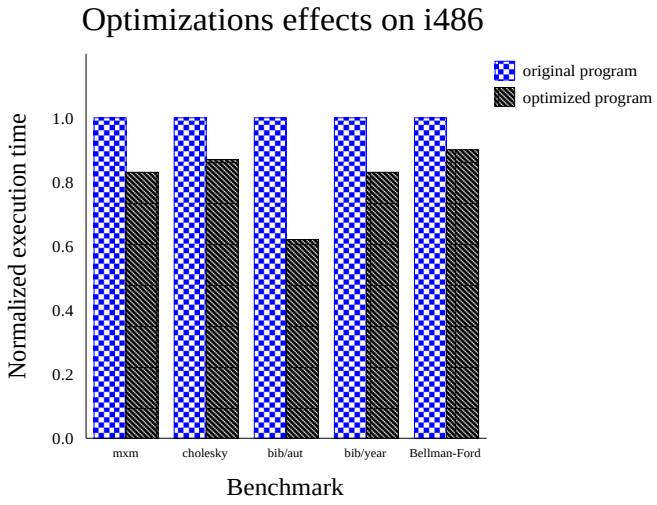

Figure 3: Speedups achieved on an i486/66 PC

The improvements for array-based optimizations are smaller than the speedups for field-layout optimizations and significantly smaller than improvements for the same programs

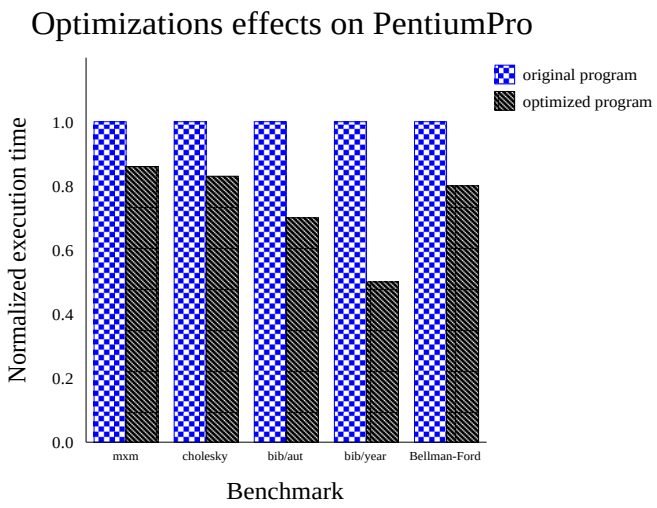

Figure 4: Speedups achieved on a PentiumPro/200 PC

implemented in programming languages like Fortran and C. We attribute that to relatively immature compiler technology for Java. Even a simple access (indexed by loop variables, like `a[i][j]`) to an element of a two-dimensional array is translated by the JIT compiler used by us to a sequence of many instructions including two conditional branches and 11 memory operations.

6 Conclusion

We have shown the improvements obtained by applying data transformations to improve the locality of memory accesses. Those optimizations cannot be performed by the Java compiler which generates the `class` file, since at that time the target architecture is not known. On the other hand some of those optimizations require high-level structure of the code being optimized which is not present in the `class` file.

Briki first recovers a high-level structure from the `class` file and then applies those optimizations to our intermediate representation before writing out the optimized code. Our results on four computationally intensive benchmarks result in reducing the computation time by 10–50% and we expect even better improvements in the future compilers.

Acknowledgements

This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

References

[1] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.

[2] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Inc., 1996.

[3] T. Wilkinson. KAFFE v0.5.5 — A JIT and interpreting virtual machine to run Java code. October 1996. [Online] Available http://www.tjwassoc.demon.co.uk/kaffe/kaffe.htm.

[4] M. Cierniak and W. Li. Briki: A Flexible Java Compiler. TR 621, Computer Science Department, University of Rochester, May 1996.

[5] H. v. Vliet. Mocha the Java decompiler. November 1996. [Online] Available http://web.inter.nl.net/users/H.P.van.Vliet/mocha.htm.

[6] C. Cifuentes. Structuring Decompiled Graphs. In *Proceedings of the International Conference on Compiler Construction*, volume 1060 of Lecture Notes in Computer Science, pages 91–105. April 1996.

[7] B. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[8] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[9] K. M. Dixit. The SPEC benchmarks. In *Parallel Computing*, pages 1195–1209, 1991.

A Example of code structure recovery

To illustrate how Briki recovers high-level structure from a bytecode representation of a method, we will show the process of structure recovery for a simple example. Relatively large sizes of our benchmark applications used to produce the results shown in Section 5 make it impractical to present them here. We chose a very short method to show here. Our example is a function which computes the nth Fibonacci number:

```
public static int fibonacci(int n) {
   int p1 = 0;
   int p2 = 1;
   for(int i = 0; i < n; i++) {
      int nextVal = p1 + p2;
      p1 = p2;
      p2 = nextVal;
   }
   return p2;
}
```

The bytecode representation of that method (after compiling with `javac`) is:

```
Method int fibonacci(int)
   0 iconst_0
   1 istore_1
   2 iconst_1
   3 istore_2
   4 iconst_0
   5 istore_3
   6 goto 22
   9 iload_1
  10 iload_2
  11 iadd
  12 istore 4
  14 iload_2
  15 istore_1
  16 iload 4
  18 istore_2
  19 iinc 3 1
  22 iload_3
  23 iload_0
  24 if_icmplt 9
  27 iload_2
  28 ireturn
```

Basic blocks start at 0, 9, 22 and 27. We label them as **BB0**, **BB9**, **BB22** and **BB27** accordingly. Figure 5 shows the basic blocks arranged in a control flow graph

The next step replaces branches with structured control flow constructs as described in Section 3.4. There are no short-circuit operators in this code. One loop is identified with a *back edge* from node **BB9** to **BB22** and so a **repeat** node is inserted before **BB22**. Next, as described in 3.6.1, we interpret each basic block converting the bytecode sequences to JavaIR. The CFG after this step is shown in Figure 6 (JavaIR is shown as Java source).

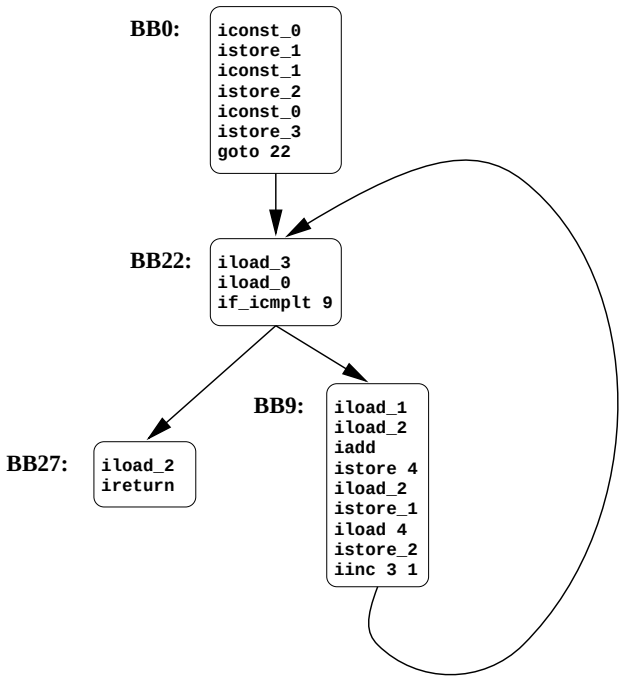

Figure 5: A control flow graph for `fibonacci`

After the structuring algorithm, the code for `fibonacci` may be presented (using the notation from [7]) as:

```
BB0
repeat R4
  if(BB22)
    BB9
  else
    BB27
  continue(1)
end of repeat R4
```

JavaIR form of `fibonacci` can be presented as the following source fragment.

```
public static int fibonacci ( int a0 )
{
   int v0, v1, v2, v4;
   v0 = 0;
   v1 = 1;
   for(v2 = 0; v2 < a0; v2 = (v2 + 1)) {
      v4 = (v0 + v1);
      v0 = v1;
      v1 = v4;
   } //for
   return v1;
} // fibonacci
```

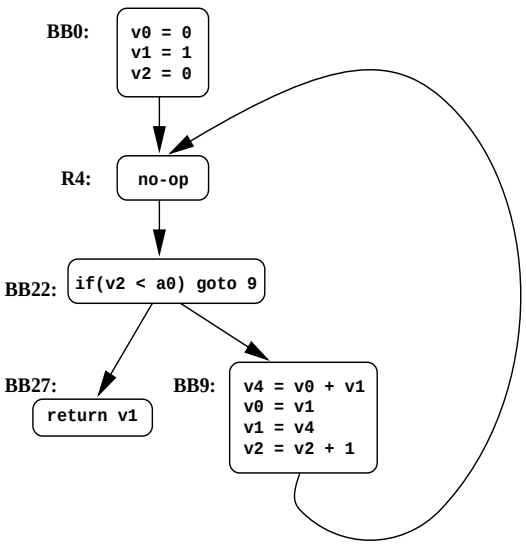

Figure 6: The CFG for `fibonacci` after translating basic blocks to JavaIR

B Example of an array transformation

We will show here a much simplified version of the matrix multiply used in our experiments. The version shown here is big enough to show how the transformation is performed. The full benchmark is too big to be presented here.

```
static public void mult(int A[][], int B[][], int C[][]) {
   int n = A.length;   // Assume A, B, C are square.
   int i, j, k;
   for(j = 0; j < n; j++) {
      for(k = 0; k < n; k++) {
         for(i = 0; i < n; i++) {
            C[i][k] += A[i][j]*B[j][k];
         }
      }
   }
} //mult
```

The bytecode representation of that method (after compiling with `javac`) is shown in Figure 7: The JavaIR form of the body of `mult` is presented in Figure 8. The goto elimiantion phase of Briki constructs the JavaIR form, so that it is equivalent to the original source shown earlier.

Our data remapping pass analyzes the JavaIR form of the method and first determines which arrays may be safely remapped. At the same time desired mappings for all multi-dimensional array references are found. This information is used to decide which (if any) arrays will be remapped. The next stage remaps all those arrays.

In our model we assume that array references with stride 1 in the innermost loop will have better locality properties than references with larger strides. A *stride* for a given reference is

```
void mult(int [][],
          int [][],
          int [][])
   0 aload_0
   1 arraylength
   2 istore_3
   3 iconst_0
   4 istore 5
   6 goto 67
   9 iconst_0
  10 istore 6
  12 goto 58
  15 iconst_0
  16 istore 4
  18 goto 49
  21 aload_2
  22 iload 4
  24 aaload
  25 iload 6
  27 dup2
  28 iaload
  29 aload_0
  30 iload 4
  32 aaload
  33 iload 5
  35 iaload
  36 aload_1
  37 iload 5
  39 aaload
  40 iload 6
  42 iaload
  43 imul
  44 iadd
  45 iastore
  46 iinc 4 1
  49 iload 4
  51 iload_3
  52 if_icmplt 21
  55 iinc 6 1
  58 iload 6
  60 iload_3
  61 if_icmplt 15
  64 iinc 5 1
  67 iload 5
  69 iload_3
  70 if_icmplt 9
  73 return
```

Figure 7: The bytecode representation of the method `mult`

the distance between addresses of the two array elements accessed in consecutive iterations of the loop being considered.

For our example, the innermost-loop strides for references `C[i][k]` and `A[i][j]` are greater than 1 and the stride for `B[j][k]` is 0. A data transformation cannot change the innermost stride the reference to array `B` since the loop index variable is not used in any of its subscripts, we can however improve the strides for the other two references.

The transformations used by us will correspond to swapping the two dimensions of arrays `A` and `C`. This will change the innermost loop to:

```
for(i = 0; i < n; i++) {
   C[k][i] += A[j][i]*B[j][k];
}
```

Note that for the simplified example used in this section, this transformation alone would not be legal and our compiler would not perform it. Remapping of arrays `A` and `C` would be valid only if either:

- We would at run-time verify at the beginning of `mult` that the arrays are rectangular and insert code fragments at the beginning and at the end of `mult` to convert the arrays between the original and optimized mappings. The runtime overhead incurred by this approach may eliminate any benefits we got from the improved locality in the body of `mult`.

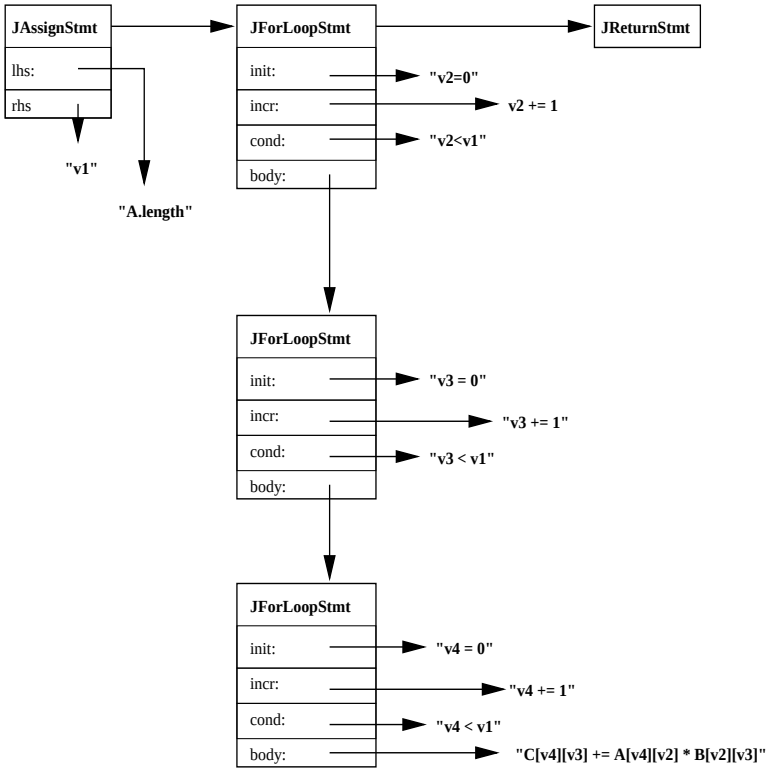

Figure 8: The JavaIR form of the body of `mult`

- We would at compile-time analyze a broader context in which the arrays of interest are used. Then, we can ensure that all other references to the transformed arrays (including their allocation) will be changed accordingly.