Java Annotation-aware Just-in-Time (AJIT) Compilation System

Ana Azevedo*, Alex Nicolau
University of California, Irvine
aazevedo, nicolau@ics.uci.edu

Joe Hummel
University of Illinois, Chicago
jhummel@eecs.uic.edu

UCI-ICS Technical Report No. 99-04

January 22, 1998

Abstract

The Java Bytecodes language lacks expressiveness for traditional compiler optimizations making this portable, secure software distribution format inefficient as a program representation to produce high performance native code from. The fact that some bytecode operations intrinsically enclose implicit sub-operations (e.g., `iaload`, includes the address computation, array bound checks and the actual load of the array element) allied to the fact that Java Bytecodes language implements a stack model account for this inefficiency. The language has no mechanism to indicate which sub-operations in the bytecode stream are redundant or subsumed by previous ones. The stack model having no operand registers and restricting access to only the top of the stack is also a limitation as it prevents the reuse of values and bytecode reordering. As a consequence the language inhibits the expression of simple compiler transformations, such as common sub-expression elimination and loop invariant removal, and more elaborate ones as register allocation and instruction scheduling.

Regardless of how optimized the bytecode stream is generated by the Java front-end, it misses ability to represent some code-improving transformations. When translating the bytecodes into native code, JIT technology has the options of directly translating the bytecodes into a poor native code or enhancing the quality of this translated code by applying code optimizations. This second alternative leads to a more time consuming translation process in the already time-constrained JIT technology. In this paper we present an alternative to an optimizing JIT compiler that makes use of code annotations generated by the Java front-end. These annotations carry information concerning compiler optimizations. In the translation process, an annotation-aware JIT (AJIT) system can use the annotation information to produce high performance native code without performing the code analysis and sometimes the transformations necessary for an optimizing JIT compiler. We describe the implementation of the first prototype of our annotation-generating compiler and our annotation-aware JIT system and show performance results comparing our system with other Java Virtual Machines (JVMs) running on SPARC machines.

*This work supported in part by CAPES.

1 Introduction

Java Bytecodes are emerging as a software distribution language for both their portability and safety features. The portability property of the language is ensured by the platform independent stack machine model that all Java source codes are compiled into. On the target machine side this intermediate code representation is either interpreted [17] or compiled into native code, using traditional ahead of time compilers [14, 21],or just-in-time compilers [1, 2, 16, 18, 24], the choice depending on the underlying Java Virtual Machine (JVM) support. The safety features of the language are based on the security violation checks performed at load time and at run-time [11]. Such checks include enforcement of methods and variables access modifiers, strict type-checking and array bound checking. Some of these checks are implicit in the bytecodes, forcing the JVM to perform them unless it carries out code analysis at run-time to prove they are superfluous.

In the design of the Java Bytecodes language a great effort was done to make it well-suitable for portability and for enforcing security. However, in order to widely accept Java Bytecodes as a programming language for the Internet it is is desirable that the language be also efficient when executing in different hardware architectures. Unfortunately this is the weakest aspect of the language and is currently focus of much research work. The inneficient execution of Java Bytecode programs lies with the definition of the Java Bytecodes themselves. The language is poor for expressing the result of many common and important compiler optimizations that are traditionally expressed in the native code produced by most compilers. A direct translation of the bytecode stream generated by a Java front-end into a target machine instructions results in low quality native code.

The first reason accounting for the weakness of Java Bytecodes in expressing compiler optimizations is the stack model of the language that restricts access to only the top of the stack and provides no operand registers. The restriction of access to only top of the stack prevents reordering of bytecodes that would be necessary to express code re-organization induced by compiler transformations, such as instruction scheduling. With no register to hold values, the stack model sequentializes computation and prevents the reuse of values, since operands must always be copied onto the top of the stack, which in turn also affects code re-structuring. As there are no means for referencing registers, the language also prevents the expression of register allocation. A second feature of the language that negatively contributes to its power as a program representation is the fact that some bytecodes intrinsically encapsulate many machine sub-operations (e.g., `iaload`, includes the address computation, array bound checks and the actual load of the array element). A Java front-end can detect when sub-operations are redundant or subsumed by preceding sub-operations in the program flow, and it can implement code-improving transformations that eliminate these sub-operations or move them around. However, the compiler is still limited on how to generate Java Bytecodes expressing such code changes as these sub-operations cannot be selectively moved apart from a bytecode and there is no mechanism in the language to turn them off in the bytecode. For this reason, simpler compiler optimizations such as common sub-expression elimination and loop invariant removal have limited expressiveness in the Java Bytecodes

Java Code

```
public static void foo(int a[], int b[], int offset1, int offset2){
   for (int i=0; i<a.length; i++)
      a[i] = b[i] + offset1 + offset2;
}
```

IR

```
 1 : smovi 0, i
 2 : aadd a, "array_size_offset", _temp1
 3 : ild (_temp1), _temp2
 4 : icmpge i, _temp2, _temp3
 5 : br _temp3 (18)
 6 : ishl i, "ishift", _temp5
 7 : iadd _temp5, "array_size_offset", _temp6
 8 : aadd b, _temp6, _temp7
 9 : ild (_temp7), _temp4
10 : iadd _temp4, offset1, _temp8
11 : iadd _temp8, offset2, _temp9
12 : ishl i, "ishift", _temp10
13 : iadd _temp10, "array_size_offset", _temp11
14 : aadd a, _temp11, _temp12
15 : ist  _temp9, (_temp12)
16 : iadd  i, 1 , i
17 : jmp (2)
18 : return
```

Optimized IR

```
 1 : iadd offset1, offset2, _tmp1
 2 : smovi 0 , i
 3 : aadd a, "array_size_offset", _temp2
 4 : ild (_temp2), _temp3
 5 : icmpge i, _temp3, _temp4
 6 : br _temp4 (16)
 7 : ishl i, "ishift", _temp6
 8 : iadd _temp6, "array_size_offset", _temp7
 9 : aadd b, _temp7, _temp8
10 : ild (_temp8), _temp5
11 : iadd _temp5, _temp1, _temp9
12 : aadd a, _temp7, _temp10
13 : ist  _temp9, (_temp10)
14 : iadd i, 1 , i
15 : jmp (5)
16 : return
```

Optimized Bytecode

```
 0 iload_2
 1 iload_3
 2 iadd
 3 istore 5
 5 aload_0
 6 arraylength
 7 istore 6
 9 iconst_0
10 istore_4
12 goto 29
15 aload_0
16 iload_4
18 aload_1
19 iload_4
21 iaload
22 iload 5
24 iadd
25 iastore
26 iinc 4 1
29 iload 4
31 iload 6
33 if_icmplt 15
36 return
```

Figure 1: Java Bytecodes as a language for program representation

language. To exemplify these limitations, consider the example in Figure 1.

This example supposes a RISC-like three-address code intermediate representation (IR) is used in the Java Bytecode compiler. The leftmost column shows the IR operations corresponding to the Java code on the top of Figure 1 [1]. After optimizing this IR, the compiler was able to produce the optimized bytecode stream listed in the last column. We can notice, in the second column, some simple optimizations done by the compiler such as loop invariant removal of expression `offset1 + offset2` and the array size reference. However, although the compiler can optimize the initial IR even more by performing common sub-expression elimination of the sub-operations comprising array elements accesses (the index is the same for accessing the integer arrays `a` and `b` and therefore the array index computation in lines 6-7 and 12-13 in the first column are redundant), these transformations could not be expressed in the optimized bytecode stream. The bytecode operation for array access includes the address computation and the actual load/store of the array element and cannot be broken down in simpler bytecode operations. As a result all sub-operations are kept in the loop. A direct translation of this optimized bytecode stream yields native code in need of quality improvement.

Although a Java front-end can compile a program into a clean and highly optimized sequence of bytecode operations, the JIT compiler, in order to generate high quality-native code, will have to perform bytecode analysis to extract information about the program and use it to implement code optimizations during the

[1] array bound checks have been omitted

translation process. This introduces an overhead in the already time-constrained JIT technology. In this paper we present an alternative to an optimizing JIT compiler based on bytecode annotations. In our annotation-aware JIT (AJIT) compilation system, the translation of bytecodes into high-performance native code is accomplished with the help of extra analysis information carried along with the bytecodes in the form of annotations. Our idea of Java Bytecode annotations was first introduced in [15] and in this current work we present the details of the implementation of our annotation-generating compiler and our annotation-aware JIT system. Particularly we show how effective annotations are for carrying information concerning register allocation, common sub-expressions and value propagation. We have also collected some initial results on the performance of the code generated by our AJIT system and we have comparative figures that show how we outperform other JVM implementations on SPARC machines.

The format of this paper is as follows. In the next section we present the structure of our annotation-generating Java front-end and we discuss the types and formats of the annotations implemented in our first prototype. We also give details on our compile-time register allocation that produces annotations to support dynamic register allocation. In Section 3 we discuss our annotation-aware JIT (AJIT) system and show how it uses annotations to implement run-time register allocation and to produce native code. In Section 4 we discuss related work. This section is followed by Section 5 where we present some preliminary results on the performance of our AJIT system. We finalize the paper with our conclusions and future directions outline in Section 6.

2 Annotation-Generating Compilation System

The idea of annotating a program representation with analysis information produced by a front-end compiler comes from the need to speed up the work of a run-time code optimizing system. We have chosen Java Bytecodes as the program representation to annotate and the JVM as the system supporting annotations. This choice was based on the Java Bytecodes commercial success induced by its write-once-run-anywhere feature. However our concept of annotations can be applied to any program representation.

Our annotations types and formats vary with the kind of information that needs to be conveyed to the run-time code optimizing system. It can consist, for example, of program information available at the high level source code that is not captured by the program representation used for the software distribution and compile-time analysis information that is time consuming to produce at run-time. Figure 2 gives an overview of a general annotation-generating compilation system with the different types of annotations we have designed or we are in process of designing. During the initial Java to Bytecode translation, our annotation-generating compiler behaves as a traditional compiler. It builds a three-address code intermediate representation flexible enough to represent all the sub-operations that the Java Bytecodes are broken into. On this IR traditional code-improving compiler techniques (e.g., copy propagation, common sub-expression elimination, loop invariant code removal and register allocation) can be applied and an optimized IR is

obtained. Once this stage has been reached, each operation (or sequence of operations) is translated into optimized Java Bytecodes. An annotation generator block reads in the optimized IR and the data provided by compiler analysis and produces the different types of annotations. The compiler has a final mapping phase in which bytecode operations are paired with their corresponding IR operations and annotations data. For example, in the case of Virtual Register Allocation annotations, which will be explained in the following paragraphs, each bytecode is annotated with the source and destination registers (as well as any intermediate values) allocated for the corresponding Java IR operations operands. Finally, the bytecode stream is copied into the code attribute section of the class file together with the annotations, input into the class file as an extra code attribute. Storing annotations in this way guarantees portability and compatibility with existing JVMs that do not understand the annotations code attribute, which is by convention just ignored [11].

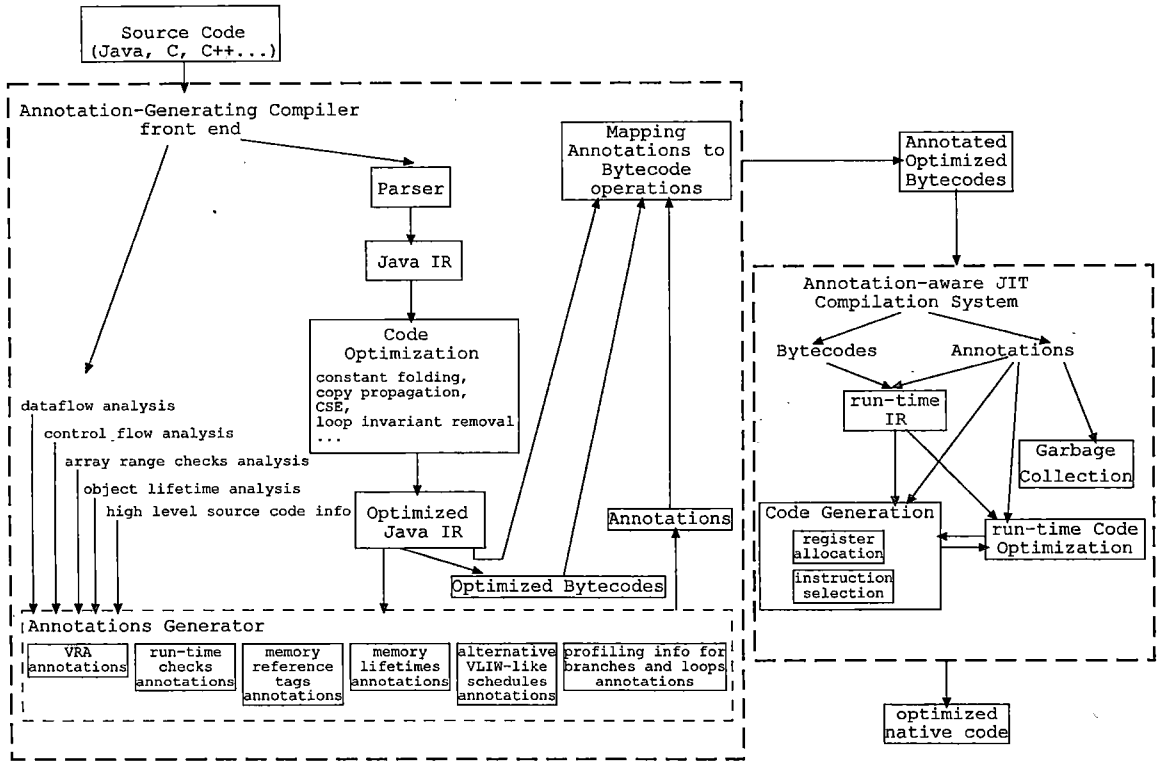

Figure 2: Annotation-generating compiler and annotation-aware JIT (AJIT) system

Our annotation-generating compiler was built on the freely available Java Bytecode compiler *guavac* version 0.3.1 [22]. From the Java source code, this compiler generates a parser tree and produces bytecodes. We augmented the compiler code by introducing functions for building and manipulating our three-address code Java IR, by implementing compiler optimizations for common sub-expression elimination, copy propagation and virtual register allocation and by designing the Virtual Register Allocation (VRA) annotations generator. VRA annotations are the type of annotations we will be discussing in this current paper. For the

other types of annotations listed in Figure 2, what they mean and the purpose they serve, the reader can resort to [15]. Some of these annotations types, such as memory lifetimes annotations, alternative VLIW-like schedules annotations and profiling information annotations are not described in our previous paper and consist part of our future work discussed in Section 6.

Virtual Register Allocation annotations represent the result of performing register allocation assuming an infinite number of registers. The information provided by the VRA annotations is used by the JIT compiler to perform a fast and efficient register allocation and also to indicate which bytecodes or bytecode sub-operations are redundant [2] or subsumed by preceding operations and therefore do not need to be translated into machine instructions. How the JIT compiler interprets the annotations, does the register allocation and produces native code eliminating redundant computation is explained in Section 3. In the remaining of this section we will dicuss the format of VRA annotations and how our front-end compiler produces them.

Each instruction defined in the Java Bytecodes language is mapped into operations in our Java IR. Annotations for virtual register allocation basically hold information on the operands of the Java IR operations. The VRA annotations represent source operands, destination operands and any intermediate value implicitly calculated by the bytecode sub-operations (e.g., array address calculation in an array load operation). For each bytecode instruction one or more VRA annotations format exist. Each format indicates how a particular bytecode sub-operation should be translated: from where to read its input operands, where to write the result into, or when it should be skipped from translation, as the result value it would produce has already been generated by some previous computation that reaches this program point and is available in a virtual register.

Figure 3 shows an example of correspondence between bytecodes, Java IR and VRA annotations formats. Each `SRC`, `EXTRA` and `DEST` fields hold virtual register numbers representing the operands for the sub-operations. In Figure 3(a) the Java IR code sequence for the computation performed by the bytecode `iaload` is illustrated. The most general format of an `iaload` operation includes 2 `SRC` fields, 2 `EXTRA` fields and one `DEST` field with header format `SRC-SRC-EXTRA-EXTRA-DEST`. The first `SRC` field represents the virtual register that holds the array object reference; the second `SRC` field represents the virtual register that holds the index; the first `EXTRA` field represents the result of the array index calculation; the last `EXTRA` field represents the result of the array address calculation; finally the `DEST` field represents the virtual register holding the array element read from memory. In case the address computation has already been computed before, as in Figure 3(b), the header `SRC-DEST` indicates that the `SRC` field holds the array element address and `DEST` field is the suggested virtual register to hold the value read from memory, meaning that the translation process can skip the sub-operations for array index and address calculation and the bytecode `iaload` can be translated into a single load operation.

In Figure 4, we show how local variables and class member variables are represented in our Java IR.

[2] redundant bytecodes appear in the optimized bytecode stream due to the stack machine model

Bytecode	Java IR
iaload	_temp0 holds array address _temp1 holds index 1 : ishl _temp1, "ishift", _temp2 2 : iadd _temp2, "array_size_offset", _temp2 3 : aadd _temp0, _temp2, _temp3 4 : ild (_temp3), _temp4
Annotated Bytecode	
opcode SRC SRC EXTRA EXTRA DEST iaload _temp0 _temp1 _temp2 _temp3 _temp4	

(a) Array element address calculation and array load

Bytecode	Java IR
iaload	_temp0 holds array element address 4 : ild (_temp0), _temp1
Annotated Bytecode	
opcode SRC DEST iaload _temp0 _temp1	

(b) Array load

Figure 3: Example of VRA annotations for `iaload` operation

Bytecode	Java IR	VRA Annotation Formats
iload	nop	CONST SRC
istore	imov _temp1, _temp2	SRC DEST
	imov const, _temp1	CONST DEST
	nop	CONST SRC
getstatic	amovi "address of class variable", _temp1 {b,c,s,i,l,d,f,a}ld (_temp1), _temp2	EXTRA DEST
	{b,c,s,i,l,d,f,a}ld (_temp1), _temp2	SRC DEST
	nop	SRC
getfield	amovi "address of object, _temp1 amovi "offset of field", _temp2 aadd _temp1, _temp2, _temp3 {b,c,s,a,d,f,l,i}ld (_temp3), _temp4	EXTRA EXTRA EXTRA DEST
	amovi "offset of field", _temp2 aadd _temp1, _temp2, _temp3 {b,c,s,a,d,f,l,i}ld (_temp3), _temp4	SRC EXTRA EXTRA DEST
	aadd _temp1, _temp2, _temp3 {b,c,s,a,d,f,l,i}ld (_temp3), _temp4	SRC SRC EXTRA DEST
	{b,c,s,a,d,f,l,i}ld (_temp1), _temp2	SRC DEST
	nop	SRC

Figure 4: Example of VRA annotations for local variables and class member variables accesses

Local variables are directly mapped to virtual registers. Local variables accesses (e.g, `iload` and `istore`) are represented in our Java IR as `nop` operations or move operations, annotated as `SRC-DEST`, `CONST-DEST`, `CONST` or `DEST`, depending on the result of optimizing the Java IR via copy propagation. When the JIT interprets these two last formats of annotations, `SRC` or `CONST`, it has the information that either the local variable is in a virtual register indicated by the byte following the header or that it is a constant. In both cases, no machine code is generated for the bytecode. Class member variables are kept as variables in memory in our front-end compiler and accesses to them are via load and store operations, as shown in Figure 4 for bytecodes `getstatic` and `getfield`. As a consequence these variables are also kept in memory in our AJIT system. In order to still allow some optimization on accesses to class member variables we devised annotations that make explicit the variable address calculation, just like we did with array references. For example, for the bytecode `getfield`, the different annotations formats `EXTRA-EXTRA-EXTRA-DEST` and `SRC-DEST` inform whether or not the variable address has been computed before. To treat these variables the same way we deal with local variables we have to extend our register allocation algorithm to assign variables to registers over the entire program and not only inside methods scope. In this case we have to take into account the side effect of all the methods that can potentially modify the member variables. The code for these methods can be known at the bytecode production time or not. We plan to modify our register allocation algorithm and use our annotation framework to allow global register allocation at run-time, much like the type of register assignment introduced by David Wall in [23].

The choice of the virtual register to hold the operations operands is crucial to the register allocation done at run-time. In order to perform a fast and efficient register allocation at run-time the VRA annotations must indicate which variables should be allocated or have more preference in being allocated to a physical register. This is accomplished by assigning at compile-time the lowest virtual register numbers to the most important variables in a method code, and then, at run-time, giving priority to assign those lowest virtual register numbers to machine registers. The details of our compile-time register allocation algorithm are presented in Section 2.1.

When designing the VRA annotations we opted for a format that was easy to be decoded by the JVM run-time system so that processing the annotations introduces low overhead. The general VRA annotations format includes a byte-long header followed by a variable number of bytes representing the virtual register numbers. The header indicates how the subsequent annotations bytes should be interpreted. In our first prototype, we did not try to optimize the space consumed by the annotations, and our measures indicate annotations can double the size of the bytecode stream [15]. Another drawback of our annotation approach that we have not yet come around in this first implementation is the need for a validation scheme to verify the extra information represented in the class file, as malicious or incorrect annotations can cause erroneous native code to be generated.

2.1 Compile-time Register Allocation

In our annotation-generating compiler we implement a modified priority-based graph-coloring algorithm. In a traditional Chaitin-style graph coloring algorithm [5, 4], an interference graph is pruned to decide the ordering in which live ranges are assigned to colors (and ultimately registers). A priority-based coloring algorithm [6] uses some heuristics and cost analysis to determine the ordering of live ranges and guarantees that the most important live ranges are assigned colors first. In our first compiler prototype, variables (including method parameters, method local variables and compiler generated temporaries) are prioritized by their static reference counts, having references inside loops, no matter how deeply nested, counting as 10. An alternative algorithm for assigning variables priorities would be to take into account profiling information instead of static reference counts. This alternative in our list of future research. As we want to keep the number of virtual registers the smallest as possible, we want to assign the same virtual register number to variables with non-conflicting live ranges. This is accomplished by building the interference graph which gives us information on conflicting live ranges. Using the information provided by the interference graph, the color assignment algorithm picks variables from the priority list and assigns virtual register numbers (colors) to them, reusing lowest virtual register numbers or creating a new virtual register number in case of conflicts with all already existing virtual register numbers.

Figure 5 shows the main function implementing the method's virtual register allocation. After the generation of the Java IR, the compiler runs data flow analyses and implements the optimizations of copy propagation and common sub-expression elimination. This optimized IR is used by the compiler to calculate the static reference counts for each variable. As seen in the code, the register allocator runs additional dataflow analyses, builds the interference graph, the variables priority list and finally assigns colors to the variables. Figure 5 also shows the variables priority algorithm. In case of matching static reference counts, the priority of a variable is dictated by the order in which it was referenced in the code. Figure 6 shows the color assignment algorithm. When assigning virtual register numbers (or colors) we associate each virtual register number with the Java type of the variable it is allocated for, and we do not allow a virtual register number holding an integer variable to later be reused to hold a floating point variable. This restriction, although it has the counter effect of increasing the number of virtual registers it guarantees that the assignment of a virtual register to a physical register is more fixed in the back-end. Otherwise, the frequent re-mapping of virtual registers to physical registers breaks the virtual registers priorities and to correct it spilling is introduced in the code, which has a worse effect.

```
void method::virtualRegAlloc(){

   liveAnalysis();
   buildInterferenceGraph();
   buildPriorityList();
   assignColors();

}

void method::buildPriorityList(){

   _priorityList.vector();

   while(hasToVisit()){
     unsigned int selected = getVariableWithMaxPriority();
     _priorityList.push_back(_nodeList[selected]);
   }
}

unsigned method::getVariableWithMaxPriority(){
  // _nodeList holds all the  variable arguments referenced in the program

  unsigned int i=0;
  while((_nodeList[i]->isVisited()) && (i<_nodeList.size())) i++;
  unsigned int max = _nodeList[i]->getPriority();
  int selected = i;

  for(i; i<_nodeList.size(); i++){
    if (_nodeList[i]->isVisited()) continue;
    else if ( _nodeList[i]->getPriority() > max){
         max = _nodeList[i]->getPriority();
         selected = i;
    }
  }
  _nodeList[selected]->visited(1);
  return selected;
}
```

Figure 5: Compile-time algorithm for virtual register allocation

```
void method::assignColors(){
   vector <varArg *>::const_iterator i;
   unsigned int nodeID, id;
   vector <varArg *>::const_iterator j;
   unsigned int c;
   unsigned int assignedColors;
   vector<unsigned int> assigned;
   vector<argType> colorTypeInfo;
   unsigned int conflict = 0;
   unsigned int k, m;

   assignedColors=1;
   assigned.vector();
   colorTypeInfo.vector();
   colorTypeInfo.push_back(t_Unknown);
   // first position not used

   for(i=_priorityList.begin(); i != _priorityList.end(); i++){
      vector <var Arg *> adjList = (*i)->getAdjList();
      for(j= adjList.begin(); j != adjList.end(); j++){
         if( (*j)->isColored()){
            c = (*j)->getColor();
            assigned.push_back(c);
         }
         else continue;
      }
      if (assigned.empty()){
         // the neighbors have not been colored
         if (assignedColors == 1){
             // first color assignment
             (*i)->setColor(assignedColors);
             colorTypeInfo.push_back((*i)->getCType());
             assignedColors++;
         }
         else{
            //choose among the existent colors.
            //if not appropriate type, create a new color
            for(c=1; c<assignedColors; c++){
               if ( ( ((*i)->getCType() == t_Reference) ||
                      ((*i)->getCType() == t_Byte) ||
                      ((*i)->getCType() == t_Character) ||
                      ((*i)->getCType() == t_Short) ||
                      ((*i)->getCType() == t_Integer) ||
                      ((*i)->getCType() == t_Long) ) &&
                    ( (colorTypeInfo[c] == t_Reference) ||
                      (colorTypeInfo[c] == t_Byte) ||
                      (colorTypeInfo[c] == t_Character) ||
                      (colorTypeInfo[c] == t_Short) ||
                      (colorTypeInfo[c] == t_Integer) ||
                      (colorTypeInfo[c] == t_Long) ) ){
                        (*i)->setColor(c);
                        break;
                      }
               else if ( ( ((*i)->getCType() == t_Float) ||
                           ((*i)->getCType() == t_Double) ) &&
                         ( (colorTypeInfo[c] == t_Float) ||
                           (colorTypeInfo[c] == t_Double) ) ){
                             (*i)->setColor(c);
                             break;
                           }
            }
            if (c == assignedColors){
               //none of the previous colors served
               (*i)->setColor(assignedColors);
               colorTypeInfo.push_back((*i)->getCType());
               assignedColors++;
            }
         } // else
      }  // if
      else{
      /* the neighbors have already been colored.
         We must check whether the existent colors
         match the neighbors'.
         If no conflict, the color is a candidate
         and it will be the chosen color
         if types are compatible  */

         k=1; conflict=0;
         while(k < assignedColors){
           for(m=0; m < assigned.size(); m++){
              if (k==assigned[m]){
                 conflict = 1;
                 break;
              }
           }
           if (conflict){
              k++;
              conflict=0;
           }
           else{
               //this color is a candidate, must check types
               if ( ( ((*i)->getCType() == t_Reference) ||
                      ((*i)->getCType() == t_Byte) ||
                      ((*i)->getCType() == t_Character) ||
                      ((*i)->getCType() == t_Short) ||
                      ((*i)->getCType() == t_Integer) ||
                      ((*i)->getCType() == t_Long) ) &&
                    ( (colorTypeInfo[k] == t_Reference) ||
                      (colorTypeInfo[k] == t_Byte) ||
                      (colorTypeInfo[k] == t_Character) ||
                      (colorTypeInfo[k] == t_Short) ||
                      (colorTypeInfo[k] == t_Integer) ||
                      (colorTypeInfo[k] == t_Long) ) )
                      break;

               else if ( ( ((*i)->getCType() == t_Float) ||
                           ((*i)->getCType() == t_Double) ) &&
                         ( (colorTypeInfo[k] == t_Float) ||
                           (colorTypeInfo[k] == t_Double) ) )
                           break;
               else{
                  // conflicting types, try another color
                        k++;continue;
               }
           } // else
         } // while
         if (k < assignedColors) (*i)->setColor(k);
         else{
            _nodeList[nodeID]->setColor(assignedColors);
            colorTypeInfo.push_back((*i)->getCType());
            assignedColors++;
         }
      } // else

   (*i)->colored(1);
   assigned.erase(assigned.begin(), assigned.end());
}
```

Figure 6: Coloring algorithm

3 Annotation-Aware JIT (AJIT) Compilation System

The lower portion of Figure 2 depicits our annotation-aware JIT (AJIT) system. We modified the public domain JIT compiler *Kaffe* system [24] (version 0.9.2) to implement our annotation scheme. The changes concentrated on a few number of files and consisted on the design of a new register allocator, modifications to the generation of *Kaffe* internal intermediate representation and to its code generator for SPARC machines. Both original *Kaffe* functions and the new included functions coexist in the system, allowing the processing of annotated methods and non-annotated methods (e.g., Java class libraries files were not annotated and when compiled they go through the original *Kaffe* translation scheme), making the system capable of compiling annotated and non-annotated Java class files.

As VRA annotations are derived from translating the bytecodes into a RISC-like three address code intermediate representation, one can wonder whether they are general, flexible and helpful enough to produce optimized code for different target architectures. We have experimented with Intel architecture in [15], and with SPARC architecture in the current paper, which represent two distinct types of machines (CISC and RISC respectively). Our annotations scheme has proved to suffice the needs for generating code for these two platforms. As we experiment with other architectures our annotations types and formats will be refined accordingly.

In the AJIT system, when a class method is first called, the bytecode stream is read into a table buffer, and if there is an annotation code attribute, the annotations are also read into an annotations table and the JIT compiler invokes the corresponding translation routine. The process of producing native code from annotated Java bytecodes is done in a single pass over the bytecode stream. A loop iterates over the bytecode stream, and as it reads each bytecode and its annotations bytes, the corresponding *Kaffe* IR operation(s) is (are) generated. The generated *Kaffe* IR operation (or sequence of operations) depends on the information provided by the annotations. This information may suggest that the bytecode translation be totally skipped or some sub-operations be eliminated or simplified. Figure 7 shows a code example of how an `iaload` bytecode operation is translated using the annotations information.

The translated *Kaffe* IR operations operands are specified by virtual register numbers, extracted from the annotations bytes. Once all bytecode stream has been processed, SPARC native code is produced from the *Kaffe* IR operations stream. At this stage, as each *Kaffe* IR operation is translated into native code, the register allocator is invoked to replace virtual register numbers by machine registers.

The run-time register allocator is a fast and effective algorithm that essentially maps each virtual register to a machine register privileging the assignment of lower virtual register numbers. This guarantees that high priority values (program variables represented by lower virtual register numbers) have preference in the register assignment. When running out of physical registers, virtual registers are mapped to temporaries on the stack. In the case of the SPARC machine, the register allocator reserves four registers of each type (four of the global integer registers `g0-g7` and four of the floating point registers `f0-f31` for evaluating expressions

that involve variables that are not mapped into machine registers). The algorithm uses a mapping table as an auxiliary data structure. The mapping table stores information on a virtual register number, a pointer to the corresponding physical register table entry and the stack offset value it should use in case of spilling.

There are some details on the initialization of the mapping table to correctly handle the SPARC register windows convention. These details are taken care of in the methods prologue and on the translation of bytecodes for accessing method local variables. Method local variables that are parameters are passed in special integer registers (`i0-i5`) forcing the mapping of virtual registers associated with these parameters. This may break virtual register priorities, and the register allocator fixes it by spilling already mapped lower priority virtual registers in case a higher priority virtual register number needs a physical register and there are none available. The algorithm uses local registers `l0-l7` , global registers `g1-g3`, any unused input register `i0-i5` and floating point registers `f0-f27` for allocation. Global registers `g4-g7` and floating point registers `f28-f31` are reserved as scratch for those virtual registers mapped as temporaries on the stack. Registers `o0-o7` are also not available for the allocator and are reserved for passing parameters to method calls.

Our current register allocation scheme does not try to minimize call costs. At method call boundaries move operations are generated to guarantee values are in the correct registers required by the calling convention and spilling of all active registers is done. Our annotation scheme could be used to carry information on which values produced in the program are later passed to methods as parameters and also which registers should be saved across procedure calls. Having the first kind of information would guide the register allocator in the virtual to physical register mapping and would avoid some copies. The second kind of information would decrease the overhead of subroutine calls by spilling only the registers that are later referenced in the program. We are currently investigating how our virtual register allocator in our annotation-generating front-end can be extended to lower the cost of method calls.

To prove that our AJIT system is an acceptable engineering solution we need to quantify the overhead of processing the annotated bytecode stream and the overhead of our mapping-based register allocation in the process of generating optimized native code. If we manage to generate better optimized native code in a shorter amount of time as compared to an optimizing JIT compiler we show that our framework is a good solution to speedup Java execution. The annotations overhead is due to the larger size of the class file that increases downloading time, the time spent on the interpretation of the information conveyed in the annotation bytes (see the extra processing required to build the *Kaffe* JIT IR in Figure 7), the time spent on the run-time register allocator and the demand for resource requirement (extra memory for storing annotations table). Network applications are sensitive to the downloading time overhead, but other types of applications that do not depend on annotated class files being downloaded are not slowed down. In our AJIT system the run-time IR (the *Kaffe* IR) is simple to build and manipulate. Other optimizing JIT systems may need a more complex IR to enable compiler transformations to be easily performed. We believe that the overhead of processing the annotations, storing them and building a simple run-time IR will ultimately be

```
define_insn(IALOAD)
{
/*
* ..., array ref, index -> ..., value
*/

  a = meth->annotations_table->entry[i];
  i++;

  if (a.header == SRC_SRC_EXTRA_EXTRA_DEST){
     index = *(a.VRAData); objref = *(a.VRAData+1);
     tmp1 = *(a.VRAData+2); tmp2 = *(a.VRAData+3); dest = *(a.VRAData+4);

     annotated_lshl_int_const(vrslots[tmp1].slots, vrslots[index].slots, SHIFT_jint);
     if (object_array_offset !=0)
        annotated_add_int_const(vrslots[tmp1].slots, vrslots[tmp1].slots, object_array_offset);
     annotated_add_ref(vrslots[tmp2].slots, vrslots[objref].slots, vrslots[tmp1].slots);
     annotated_load_int(vrslots[dest].slots, vrslots[tmp2].slots);

  }else if (a.header == CONST_SRC_EXTRA_EXTRA_DEST){
     cindex = *(a.VRAConst); objref = *(a.VRAData);
     tmp1 = *(a.VRAData+1);tmp2 = *(a.VRAData+2); dest = *(a.VRAData+3);

     annotated_move_int_const(vrslots[tmp1].slots, (cindex<<SHIFT_jint), NULL);
     if (object_array_offset !=0)
        annotated_add_int_const(vrslots[tmp1].slots, vrslots[tmp1].slots, object_array_offset);
     annotated_add_ref(vrslots[tmp2].slots, vrslots[objref].slots, vrslots[tmp1].slots);
     annotated_load_int(vrslots[dest].slots, vrslots[tmp2].slots);

  }else if (a.header == SRC_SRC_EXTRA_DEST){
     objref = *(a.VRAData); tmp1 = *(a.VRAData+1); tmp2 = *(a.VRAData+2); dest = *(a.VRAData+3);

     annotated_add_ref(vrslots[tmp2].slots, vrslots[objref].slots, vrslots[tmp1].slots);
     annotated_load_int(vrslots[dest].slots, vrslots[tmp2].slots);

  }else if (a.header == SRC_DEST){
     tmp1 = *(a.VRAData); dest = *(a.VRAData+1);
     annotated_load_int(vrslots[dest].slots, vrslots[tmp1].slots);
  }else if (a.header == SRC){
   // no action
  } else error=1;
}
```

Figure 7: AJIT translation process for an **iaload** bytecode operation

less than the overhead of building, storing and manipulating a complex IR in those systems. Finally, our run-time register allocation algorithm is an algorithm that obeys a defined mapping rule and only manipulates mapping tables and can be claimed fast. No time is spent on conflict graph construction and coloring or dataflow analysis, to cite some typical tasks of traditional register allocators. For the benchmarks tested in this paper we observed a 10% average increase in compilation time for the prototype of our AJIT system, but we expect this overhead to reduce significantly once our prototype is optimized. In the final version of this paper more thorough results on the compilation time will be included.

4 Related Work

Various approaches are being proposed to overcome the inefficiency of the direct translation of the Java Bytecodes stack language and increase the execution speed of Java Bytecodes programs. When fast compilation time is not a constraint, traditional compilation of bytecodes to some higher-level form and then to native code [7, 14] and translation of bytecodes to higher level language and then the usage of an existing compiler to produce native code [21] are examples of two alternatives. When speed of compilation is an issue, optimizing JIT compilers [1, 2, 24] try to improve the quality of the native code generated on the fly by adapting traditional compilation techniques to run-time code generation. Another time in which optimizations can be implemented is in the stage after bytecode generation and before run-time translation. Examples of this possibility are bytecode optimizers [8]. Our annotation scheme is a hybrid approach in the sense that most work is done at compile time to retain important high level program information and compiler optimizations information while at run-time lightweight code improving transformations accomplish the task of generating high quality native code.

Several research work exploit the idea of code annotations and relate to our approach. Though not designed to specifically overcome the Java Bytecodes language inneficiency, these other approaches can potentially handle it. In the context of dynamic code generation, code annotations in the form of programmer hints [12] or high-level language constructs extensions [20] serve as guide to where and on what dynamic compilation should take place. These code annotations help building optimizing just-in-time compilers by extending to run-time the applicability of traditional compiler optimizations, such as copy propagation, dead code elimination, register allocation, even allowing more advanced and costly ones as cross-module optimizations that cannot be implemented statically. In these dynamic compilation systems the bulk of the compiler work is done at run-time and oftenly there is a tradeoff between dynamic compilation speed and the quality of the generated code. This tradeoff is addressed by implementing alternative strategies for code generation in the run-time system that satisfy different goals and picking the one most appropriate for a certain problem or constraint. In our annotation scheme, this same tradeoff is addressed by conveying information that decreases the code generation effort of a JIT optimizing compiler, pushing the code generation cost to compile time as much as possible.

Research in the area of developing fast run-time algorithms for common compiler optimizations is very active. In the following paragraphs we overview how some commercial and academic systems implement run-time code optimizations, such as common sub-expression elimination, register allocation and elimination of array bounds checking, and how these implementations compare to the run-time algorithms our annotation scheme requires.

Most related to our framework of mind and our VRA annotations scheme is the work developed by Wall [23] for doing cross-module link-time register allocation. In this approach, the assignment of variables to registers is treated as a form of relocation. The compiler generates code that can be directly linked and executed, but it annotates some of the instructions with register actions that describe what needs to be done to the instruction if the variables it manipulates are assigned to a register at link time. Compared to our mapping-based register allocation this approach has the overhead of the need of building the call graph, need of doing local data flow analysis and is dependent on good usage estimates (profiling information). However, it performs global register allocation while our current implementation only works for intra procedural register allocation.

The Intel's Java JIT compiler described in [2] implements a limited form of common sub-expression elimination on the bytecode stream over extended basic blocks. Our VRA annotations scheme allows a traditional CSE algorithm to be implemented at compile time and has further advantage in revealing common sub-expressions within bytecode sub-operations. In the Intel JIT compiler, register allocation is accomplished via a priority-based algorithm. Static reference count determines variables priorities and call costs and spill costs are considered in the cost analysis for assigning registers to variables. Our mapping based register allocation is also a priority-based scheme, but faster to implement at run-time as it dispenses any form of code analysis. Besides, the VRA scheme can be expanded to allocate global variables, while the Intel JIT compiler would need interprocedural dataflow analysis to do the same, incurring in a high cost run-time algorithm. A very simple array bound checks elimination algorithm was implemented in the Intel JIT compiler, dealing only with constant indexes. As described in [15], our run-time checks annotations allow powerful range analysis to be implemented at compile time and easily conveys the resultant information to the run-time system.

Another efficient JIT compiler is CACAO [1]. The translation process builds a simple IR and a static stack structure that keeps track of instructions operands and their dependencies and is used to avoid copy operations. It also runs some stack analysis that helps reducing method calls costs. The efficiency of the register allocation algorithm relies on the coloring of local variables done by the Java front-end (that assigns the same local variable number to variables which are not active at the same time) and on the fact that stack slots variables have their lifetimes implicitly encoded. Compared to our scheme, the stack analysis information that has to be computed and used in their algorithm is given with less run-time cost by our VRA annotations. Their run-time register allocator takes into account call costs, which is lacking in our current scheme. If allocation of global variables is considered, interprocedural stack analysis would

be necessary, and the algorithm would become more expensive, just like in the case of the Intel's compiler. Other optimizations such as instruction scheduling, method inlining and array bound checks removal are planned to be incorporated into CACAO, but not yet implemented.

Kaffe [24] is a freely available JVM that runs on several platforms. The translation process builds a simple RISC-like IR as it loops through the bytecode stream. Register allocation is combined with machine code generation. The register allocator is a simple algorithm that maps stack slots and local variable slots to machine registers. When running out of registers, the least recently used register is spilled and freed for allocation. There is no special treatment to reduce call costs or take advantage of machine calling conventions, as CACAO does. At method calls, copy operations are introduced to guarantee values are in the correct register (e.g., in the SPARC architecture) and all slots that got modified are spilled. No other compiler optimization is implemented. In Section 5 we have preliminary results that compare the performance of our AJIT system with *Kaffe* and they show that we outperform in quality of generated code.

Worthy mentioning in the subject topic of optimizing run-time compilation systems is the Slim Binary project [10, 19]. The project proposes an architecture-neutral intermediate representation for software distribution, called slim binaries, that can be seen as an alternative to Java Bytecodes. Code optimizations either take place in the background while the system is running, or on specific request of the programmer. Just like in the dynamic compilation systems discussed in [12, 20], this system tries to utilize run-time information (e.g., values of variables, run-time profiling information) to perform customized optimizations. This is an example of system with a more complex tree-based intermediate representation incurring some run-time overhead. The same way our annotation scheme extends Java Bytecodes with extra information that is collected during traditional compilation, the Slim Binary representation could benefit from the annotations scheme, such as the register allocation information, to decrease run-time costs.

In all optimizing JIT compilers there is an attempt to devise compiler optimizations using linear time algorithms with respect to some parameter (e.g., for Java JIT compilers, the number of bytecode instructions, or number of local variables or stack variables could be the parameters). When designing annotations we have this concern in mind as well. Our current implementation of our annotation scheme allows run-time register allocation and guarantees the execution with linear time complexity.

5 Results

Our results revolve around four benchmarks: `Neighbor`, which performs a nearest-neighbor averaging across all elements of a two-dimensional array; `EM3D`, a code that creates a graph and then performs a 3D electromagnetic simulation [9]; `Huffman`, a character string compression and decompression application; and `Bitonic Sort`, which builds a binary tree and then performs bitonic sorting (recursively) [3]. To measure the impact of our AJIT system, we collected results using the JVMs available on SPARC machines: Sun's JDK version 1.1.1 [17] and *Kaffe* JVM version 0.9.2 [24]. The execution time results are shown in Table 1.

Note that the timings do not include translation nor compile time and they only represent the quality of the generated code. All codes were compiled using our annotation-generating Java Bytecode compiler and then executed using Sun's interpreter, *Kaffe* JIT compiler and our AJIT system.

Benchmarks	SUN Interpreter (in secs)	kaffe JIT (in secs)	AJIT (in secs)	SpeedUp AJIT/SUN	SpeedUp AJIT/Kaffe

`Neighbor`
`Array Size = 256x256`
`Iterations = 1500`
	553.03	162.73	115.31	4.80	1.41

`EM3D`
`Tree Size = 1250 nodes`
`Iterations = 200`
	359.84	149.86	74.51	4.83	2.01

`Bitonic Sort`
`Tree Size = 1024 nodes`
`Iterations = 512`
	167.05	141.23	120.96	1.38	1.17

`Huffman`
`Array Size = 30000 nodes`
`Iterations = 288`
	4690.00	1856.00	1487.00	3.15	1.25

Table 1: Benchmarks execution times (in seconds) and speedups

The results presented on Table 1 reflect the sole effect of our VRA annotations scheme. From the last two columns of Table 1 we can see that our annotation-based approach offers speedups varying from 1.38 to 4.83 over direct interpretation and it is 17% to 100% faster than *Kaffe* JIT technology. We can notice that the best speedups were achieved for codes consisting of basic loops iterating over array-based or pointer-based data (`Neighbor`, `EM3D` and `Huffman`). For such codes, the VRA annotations helped to identify common subexpressions and eliminate them and allowed propagation of values avoiding move operations. These correspond to optimizations that could not be expressed in the Java Bytecodes language. It also guaranteed that the most important variables, such as loop index variables, were permanently assigned to machine registers throughout the method execution. The smallest performance gain was observed for the code with high number of subroutine calls - `Bitonic Sort` is a recursive algorithm. This behavior is explained by the way our AJIT system, and *Kaffe* system as well, handle call costs in their dynamic register allocation algorithms. Both JIT compilers do not take advantage of SPARC register windows. All active registers are saved across method calls introducing high overhead. The call overhead is not an intrinsic limitation of the algorithms, it is just a limitation in the current implementations of the compilers. Both implementations can be modified to better exploit the machine architecture features.

The encouraging observation we obtained from these preliminary results is that, despite the limitations of the first implementation, our AJIT system was capable of producing machine code that executes up to twice as fast as JIT technology. By extending our VRA annotations scheme with extra information, such as registers to be saved across subroutine calls and information for basic code scheduling optimizations, and improving on the parts of the coding of the dynamic register allocator that are machine specific, we believe the impact of our run-time register allocation will be more noticeable.

6 Conclusions and Future Work

Most approaches for speeding up Java execution resort to dynamic compilation (and even dynamic code re-optimization [13]). In this scenario, run-time costs must be kept down and it is desirable that the bulk of the compilation process be done at static compile time. Having a rich program representation conveying, for example, dependence information to allow instruction scheduling and support for dynamic register allocation, decreases the time spent on run-time code generation, by cutting down the time spent on program analysis and transformation. In this paper we discussed how the Java Bytecodes language is poor as a program representation, demanding a more time consuming run-time code generation process in order to produce high quality native code. We presented an approach based on code annotations that helps to overcome this limitation. We showed the details of the implementation of our annotation-generating compilation system and our annotation-aware JIT system.

Our first prototype implements the VRA annotation scheme that conveys information for dynamic register allocation and some basic code scheduling by identifying and eliminating redundant computation and allowing propagation of values. Preliminary results show that we outperform JIT technology, producing code that runs up to twice as fast. We plan to extend our VRA annotation scheme by incorporating information that helps minimizing call costs (e.g., values to be saved across procedure calls and values passed as subroutine parameters). We started with the implementation of the VRA annotations scheme because register allocation is the most important compiler optimization to exploit today's CPUs. We also initially selected scientific benchmarks to test our approach for their higher sensitivity to such optimization. In Figure 2 we showed planned extensions of annotations types. These other annotations types aim at more sophisticated compiler optimizations, such as instruction scheduling, and garbage collection optimizations. To help devising these other annotations we will be studying non-numeric Java applications as well.

References

[1] R. Grafl A. Krall. Efficient javavm just-in-time compilation. *In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, 1998.

[2] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *Proceedings of ACM Programming Languages Design and Implementation*, pages 280–290, 1998.

[3] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical Report TR86-769, Cornell University, 1986.

[4] G. J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):201–107, June 1982.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[6] F. C. Chow and J. L. Hennessy. A priority-based coloring approach to register allocation. *ACM TOPLAS*, 12(4):501–536, October 1990.

[7] M. Cierniak and W. Li. Optimizing java bytecodes. *Concurrency: Practice and Experience*, 9(11), November 1997.

[8] L. R. Clausen. A java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11), November 1997.

[9] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing 1993*, pages 262–273, November 1993.

[10] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[11] J. Gosling, Bill joy, and G. Steele. The java language specification. Addison-Wesley, 1996.

[12] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in c. *In Proc. of PEPM*, June 1997.

[13] David Griswold. The java hotspot virtual machine architecture, March 1998.
See http://www.javasoft.com/products/hotspot/whitepaper.html.

[14] C. Hsieh, J. Gyllenhaal, and W. Hwu. Java bytecode to native code translation: The caffeine prototype and preliminary results. *Proceedings of the 29th Annual Workshop on Microprogramming*, December 1996.

[15] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, November 1997.

[16] Microsoft Inc. The microsoft virtual machine for java.
See www.microsoft.com/java/sdk/default.htm.

[17] SUN Inc. Sun interpreter.
See http://www.javasoft.com.

[18] Symantec Inc. Just in time compiler for windows 95/NT.
See http://www.symantec.com/region/can/eng/product/jit/jitreadme.html.

[19] T. Kistler and M. Franz. Dynamic runtime optimization. *In Proceedings of the Joint Modular Languages Conference, JMLC'97*, pages 53–66, March 1997.

[20] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible and high-level dynamic code generation. *Proceedings of ACM Programming Languages Design and Implementation*, 1997.

[21] T. Proebsting, J. Hartman, G. Townsend, P. Bridges, T. Newsham, and S. Watterson. Toba: A java-to-c translator.
See http://www.cs.arizona.edu/sumatra/toba.

[22] Effective Edge Technologies. guavac. See summit.stanford.edu:/pub/guavac/.

[23] D. W. Wall. Global register allocation at link-time. *In Proc. ACM SIGPLAN'86 Symp. on Compiler Construction*, pages 264–275, June 1986.

[24] Tim Wilkinson. Kaffe: A free JIT virtual machine to run java code.
See http://www.transvirtual.com.