An Annotation-aware JVM Implementation

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

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

July 1999.
To Submit to ACM CPandE 1999

Abstract

The Java Bytecode language lacks expressiveness for traditional compiler optimizations, making this portable, secure software distribution format inefficient as a program representation for high performance. This inefficiency results from the underlying stack model, as well as the fact that many bytecode operations intrinsically include sub-operations (e.g., `iaload` includes the address computation, array bounds checks and the actual load of the array element). The stack model, with no operand registers and limiting access to the top of the stack, prevents the reuse of values and bytecode reordering. In addition, the bytecodes have no mechanism to indicate which sub-operations in the bytecode stream are redundant or subsumed by previous ones. As a consequence, the Java Bytecode language inhibits the expression of important compiler optimizations, including common sub-expression elimination, register allocation and instruction scheduling.

The bytecode stream generated by the Java front-end is a significantly under-optimized program representation. The most common solution to overcome this aspect of the language is the use of a Just-in-Time (JIT) compiler to not only generate native code, but perform optimization as well. However, the latter is a time consuming operation in an already time-constrained translation process. 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 optimization. During the translation process, an annotation-aware JVM system then uses this information to produce high-performance native code without performing much of the necessary analyses or transformations. We describe the implementation of our first prototype of an annotation-aware JVM system consisting of a JIT compilation system. We also discuss basic ideas on how to verify annotated class files. We conclude the paper showing performance results comparing our system with other Java Virtual Machines (JVMs) running on SPARC architecture.

[*]This work supported in part by CAPES.

1 Introduction

The Java Bytecodes are emerging as a software distribution language for both its portability and safety features. The portability property of the language is ensured by the platform-independent stack machine model targeted by Java compilers. On the target machine, this intermediate code representation is either interpreted [18], or compiled into native code using traditional ahead-of-time [15] or just-in-time compilers [1, 2, 4, 17, 19, 20, 27]. The safety features of the language are based on the security violation checks performed at load and run-time [12]. Such checks include enforcement of methods and variables access modifiers, strict type-checking and array bounds checking. Many of these checks are implicit in the bytecodes, forcing the JVM to perform them unless it can prove at load-time (via analysis) that the checks are unnecessary.

In the design of the Java Bytecode language, a great deal of effort was spent to make it secure and portable. However, in order to be widely accepted, it must also yield efficient execution on a wide range of machine architectures. Unfortunately this is the weakest aspect of the language and is currently the focus of much research. The inefficient execution of Java Bytecode programs lies with the definition of the bytecodes themselves. The language is poor for conveying the result of many common and important compiler optimizations that are traditionally expressed in the native code generated by optimizing compilers. The direct translation of a bytecode stream generated by a Java front-end into target machine code results in low-quality code.

The first limitation in expressing compiler optimization is the stack model of the Java Bytecodes. This model provides no registers and restricts access to only the top element of the stack. Restricting access to the top of the stack prevents the reordering of bytecodes, a necessary transformation during instruction scheduling. And without registers to hold values, the stack model sequentializes computation and prevents the reuse of values (since again, only the top is accessible). Obviously, the lack of registers also prevents the expression of register allocation, a critical and potentially time-consuming optimization.

The second limitation of the Java Bytecodes as a program representation is the fact that many bytecodes intrinsically encapsulate many machine sub-operations (e.g., `iaload` includes the address computation, array bounds checks and the actual load of the array element). The Java front-end can detect when sub-operations are redundant or subsumed by preceding sub-operations, and can try to apply traditional code-improving transformations in order to eliminate these sub-operations. However, the compiler is still limited by the stack-based nature of the Java Bytecodes, in which sub-operations cannot easily be separated, eliminated or rearranged. Furthermore, there is no mechanism in the language to disable sub-operations when deemed unnecessary. For this reason, straighforward compiler optimizations such as common sub-expression elimination, array bounds check elimination and loop-invariant code removal have limited expressiveness in Java Bytecodes.

To demonstrate these limitations, consider the example in Figure 1. This example assumes that a RISC-like, three address code Intermediate Representation (IR) is used in the Java front-end to bytecode compiler.

The leftmost column shows the unoptimized IR[1] corresponding to the Java code at the top of Figure 1. The middle column shows the result of performing some simple optimizations, such as loop invariant removal of expression `offset1 + offset2` and the array size reference. After optimizing this IR, the compiler is then able to produce the optimized bytecode stream shown in the last column. However, additional optimizations are possible that cannot be expressed in the final bytecode. For example, the sub-operations comprising array element accesses represent common sub-expressions and thus one could be eliminated (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 leftmost column are redundant). Likewise, given the bounds on the loop, all array bounds checks involving `a` are unnecessary (and those involving `b` could be reduced to a single check before the loop starts). Clearly, the resulting bytecode has room for improvement.

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, "arraySizeOffset", T1
 3 : ild (T1), T2
 4 : icmpge i, T2, T3
 5 : br T3 (18)
 6 : ishl i, "ishift", T5
 7 : iadd T5, "arraySizeOffset", T6
 8 : aadd b, T6, T7
 9 : ild (T7), T4
10 : iadd T4, offset1, T8
11 : iadd T8, offset2, T9
12 : ishl i, "ishift", T10
13 : iadd T10, "arraySizeOffset", T11
14 : aadd a, T11, T12
15 : ist  T9, (T12)
16 : iadd  i, 1 , i
17 : jmp (2)
18 : return
```

Optimized IR

```
 1 : iadd offset1, offset2, T1
 2 : smovi 0 , i
 3 : aadd a, "arraySizeOffset", T2
 4 : ild (T2), T3
 5 : icmpge i, T3, T4
 6 : br T4 (16)
 7 : ishl i, "ishift", T6
 8 : iadd T6, "arraySizeOffset", T7
 9 : aadd b, T7, T8
10 : ild (T8), T5
11 : iadd T5, T1, T9
12 : aadd a, T7, T10
13 : ist T9, (T10)
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

The implication is that even though the Java front-end can compile a program into a clean and optimized sequence of bytecodes, a JIT compiler will still need to perform significant optimization in order to generate high-quality native code. This in turn implies that a JIT compiler will have to perform bytecode analysis to extract information about the program for the purposes of optimization. This introduces a potentially significant overhead in an already time-constrained JIT system. In this paper we present an alternative to the traditional 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 [16]; in this paper we present the details of the

[1] Array bound checks have been omitted.

implementation of our annotation-aware JIT system. In particular, we show how annotations are effective in carrying information concerning register allocation, common sub-expressions and value propagation. We also discuss the basic security checks that are necessary to validate untrusted annotated Java class files. We finalize the paper presenting some initial results on the performance of the code generated by our AJIT system, demonstrating that our approach outperforms other JVM implementations on the SPARC architecture.

The format of this paper is as follows. In the next section we present the structure of our annotation generating Java front-end and discuss the types and formats of the annotations implemented in our first prototype. We also provide details concerning our compile-time register allocator that produces the annotations in support of 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 produce native code. In Section 4 we talk about the VRA annotations verification process. In in Section 5 we discuss related work. Finally Section 6 presents some preliminary results on the performance of our AJIT system, followed by our conclusions and discussion of future work in Section 7.

2 Annotation-Generating Compilation System

The idea of annotating a program representation with analysis information produced by a front-end compiler stems from the need to reduce the workload of run-time code optimizing systems. We have chosen to annotate the Java Bytecode representation, given its commerical success and widespread availability induced by its write-once-run-anywhere capability. However, the concept of annotations is a general one, and thus can be applied to any program representation.

Our annotation types and formats vary with the kind of information that needs to be conveyed to the run-time code optimizing system. For example, it may consist of high-level program information that is not expressible in the lower-level program representation, or compile-time analysis information that is too time consuming to produce at run-time. Figure 2 gives an overview of a general annotation-generating compilation system with a number of different annotations that we are currently working on. 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 form each bytecode. On this IR traditional code-improving techniques (e.g., copy propagation, common sub expression elimination, loop invariant code removal and register allocation) are applied and an optimized IR produced. Once this stage has been reached, each operation (or sequence of operations) is translated into an optimized stream of Java Bytecodes. Next, an annotation generator also reads the optimized IR, along with the data provided by various compiler analyses, and produces a set of annotations. Finally, the compiler performs a mapping phase in which the bytecode operations are paired with their corresponding IR operations and annotations, and then stores the annotated bytecode into the appropriate class file.

For example, in the case of the Virtual Register Allocation (VRA) annotations (to be explained shortly), each bytecode is annotated with the source and destination registers allocated to the operands of that Java IR operation. Then, the bytecode stream is copied into the code attribute section of the class file together with the annotations, the latter being stored as an extra code attribute. Storing annotations in this way guarantees backward compatibility with existing JVMs, which by definition must ignore unknown code attributes [12].

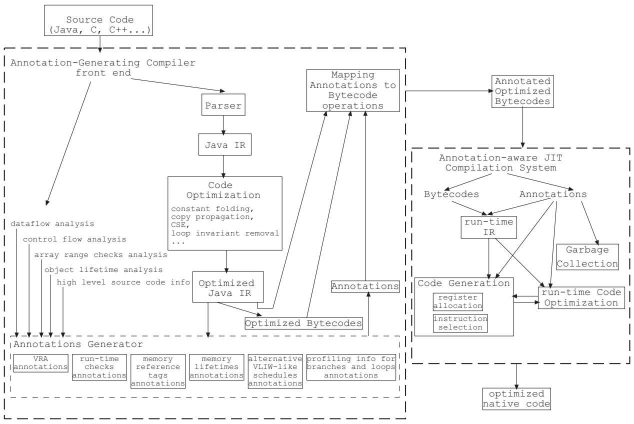

Figure 2: Annotation generating compiler and annotation aware JIT (AJIT) system

Our annotation-generating compiler was built on the freely available Java Bytecode compiler *guavac* version 0.3.1 [25]. From the Java source code, this compiler generates a parse tree and produces bytecodes. We augmented the compiler by (a) introducing functions for building and manipulating a three-address code IR, (b) implementing compiler optimizations for common sub-expression elimination, copy propagation and virtual register allocation, and (c) designing a Virtual Register Allocation annotation generator. This paper focuses in particular on the VRA annotations. The remaining annotations, as presented in Figure 2, are discussed in [16] or represent future work (see Section 7).

Virtual Register Allocation annotations represent the result of performing register allocation assuming an infinite number of *virtual* registers. The information provided by the VRA annotations can then be used by a JVM engine, either an interpreter or a JIT compiler, to perform a fast and efficient dynamic register allocation and also to indicate which bytecodes (or bytecode sub-operations) are redundant [2] or subsumed by preceding operations; such operations need not be translated into native code. In Section 3 we show in detail how a JIT compiler interprets these annotations, does register allocation, and produces native code. In the remainder of this current section we discuss the format of VRA annotations and how the Java front-end compiler produces them.

Each instruction defined in the Java Bytecode language is mapped into operations in our Java IR. Anno-

[2] As discussed earlier, redundant bytecodes appear in the optimized bytecode stream due to the stack machine model.

tations 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 values implicitly calculated by the bytecode sub-operations (e.g., array index calculation in an array load operation). For each bytecode operation type there is a distinct VRA annotation format. A format may have variations at different program points indicating how a particular bytecode sub-operation should be translated: where to read its input operands, where to write the result, and perhaps whether or not this operation should be skipped entirely (e.g. when a previous operation has already computed the needed value).

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 Case 1 of Figure 3, 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 `SRC-SRC-EXTRA-EXTRA-DEST` as annotation header format. 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; and the `DEST` field represents the virtual register holding the array element read from memory. If the address computation has already been computed, as in Figure 3 Case 2, 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.

Case 1: Array element address calculation and array load

Bytecode	Java IR
iaload	V0 holds array address V1 holds index 1 : ishl V1, "ishift", V2 2 : iadd V2, "arraySizeOffset", V2 3 : aadd V0, V2, V3 4 : ild (V3), V4

Annotated Bytecode

```
opcode SRC   SRC   EXTRA   EXTRA   DEST
iaload  V0    V1      V2      V3     V4
```

Case 2: Array load

Bytecode	Java IR
iaload	V0 holds array element address 4 : ild (V0), V1

Annotated Bytecode

```
opcode SRC   DEST
iaload  V0     V1
```

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

In Figure 4, we show how local variables and class member variables are represented in our Java IR. Local variables are directly mapped to virtual registers. Local variable accesses (e.g, `iload` and `istore`) are represented in our Java IR as `nop` operations or move operations, annotated as `SRC-DEST`, `CONST-DEST`, or

Bytecode	Java IR	VRA Annotation Formats
iload	nop	SRC
istore	imov V1, V2	SRC DEST
	imov CONST, V1	CONST DEST
	nop	SRC

Figure 4: Example of Java IR and VRA annotations for local variables accesses

SRC, depending on the result of optimizing the Java IR via copy propagation. When the JVM interprets the annotation format SRC it has the information that the local variable is in a virtual register indicated by the byte following the format header but no machine code is generated for the bytecode. Class member variables are kept as variables in memory in our front-end compiler and accessed via load and store operations, as shown in Figure 5 for bytecodes getstatic, putstatic, getfield and putfield. As a consequence, these variables are also kept in memory in our AJIT system. To enable some optimization on accesses to class member variables, we devised annotations that make explicit the variable address calculation, just like those in array references. For example, bytecode getfield has the different annotation formats SRC-DEST and SRC-EXTRA-EXTRA-DEST which state whether or not the variable's address has already been computed.

A complete listing of all Java Bytecode operation types, the corresponding Java IR operations and VRA annotations formats can be found in the Appendix (Section 8).

The choice of which virtual register to hold an operation's operands is crucial to the register allocation done at run-time. In order to enable a fast and efficient dynamic register allocation, the VRA annotations must convey the order in which variables should be allocated to physical registers (and thus which should be spilled if necessary). This is accomplished by assigning, at compile-time, the lowest virtual register numbers to the most important variables in the code. Then, at run-time, the register allocator should assign the lowest virtual register numbers to the physical 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 for the run-time system to decode so that processing the annotations would incur minimal overhead. The general VRA annotation formats include a byte-long header followed by a variable number of bytes representing the virtual register numbers. The header indicates how the subsequent annotation bytes should be interpreted. In our first prototype, we did not try to optimize the space consumed by the annotations, and thus we found that our annotations can double the size of the bytecode stream [16]. Another, potentially more significant problem is that of verification: to maintain security, a scheme is needed to verify the safety of the annotations in the class file, as malicious or incorrect annotations can lead to unsafe native code. A discussion on verifying annotated class file is carried out in Section 4.

Bytecode	Java IR	VRA Annotation Formats
getstatic	amovi “addressOfClassField”, V1 {b,c,s,i,l,d,f,a}ld (V1), V2	EXTRA DEST
	{b,c,s,i,l,d,f,a}ld (V1), V2	SRC DEST
	nop	SRC
putstatic	amovi “addressOfClassField”, V2 {b,c,s,i,l,d,f,a}st V1, (V2)	SRC EXTRA
	amovi “addressOfClassField”, V2 {b,c,s,i,l,d,f,a}st CONST, (V2)	CONST EXTRA
	{b,c,s,i,l,d,f,a}st V1, (V2)	SRC SRC
	{b,c,s,i,l,d,f,a}st CONST, (V2)	CONST SRC
	{b,c,s,i,l,d,f,a}mov V1, V2	SRC DEST
	{b,c,s,i,l,d,f,a}mov CONST, V2	CONST DEST
	nop	SRC
getfield	amovi “addressOfObject, V1 amovi “offsetOfField”, V2 aadd V1, V2, V3 {b,c,s,i,l,d,f,a}ld (V3), V4	EXTRA EXTRA EXTRA DEST
	amovi “offsetOfField”, V2 aadd V1, V2, V3 {b,c,s,i,l,d,f,a}ld (V3), V4	SRC EXTRA EXTRA DEST
	aadd V1, V2, V3 {b,c,s,i,l,d,f,a}ld (V3), V4	SRC SRC EXTRA DEST
	{b,c,s,i,l,d,f,a}ld (V1), V2	SRC DEST
	nop	SRC
pufield	amovi “addressOfObject, V2 amovi “offsetOfField”, V3 aadd V2, V3, V4 {b,c,s,i,l,d,f,a}st V1, (V4)	SRC EXTRA EXTRA EXTRA
	amovi “addressOfObject, V2 amovi “offsetOfField”, V3 aadd V2, V3, V4 {b,c,s,i,l,d,f,a}st CONST, (V4)	CONST EXTRA EXTRA EXTRA
	amovi “offsetOfField”, V3 aadd V2, V3, V4 {b,c,s,i,l,d,f,a}st V1, (V4)	SRC SRC EXTRA EXTRA
	amovi “offsetOfField”, V3 aadd V2, V3, V4 {b,c,s,i,l,d,f,a}st CONST, (V4)	CONST SRC EXTRA EXTRA
	aadd V2, V3, V4 {b,c,s,i,l,d,f,a}st V1, (V4)	SRC SRC SRC EXTRA
	aadd V2, V3, V4 {b,c,s,i,l,d,f,a}st CONST, (V4)	CONST SRC SRC EXTRA
	{b,c,s,i,l,d,f,a}st V1, (V2)	SRC SRC
	{b,c,s,i,l,d,f,a}st CONST, (V1)	CONST SRC
	{b,c,s,i,l,d,f,a}mov V1, V2	SRC DEST
	{b,c,s,i,l,d,f,a}mov CONST, V1	CONST DEST
	nop	SRC

Figure 5: Example of Java IR and VRA annotations for class member variables accesses

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, 6], 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 [7] uses heuristics and cost analyses to determine the ordering of live ranges and guarantees that the most important live ranges are assigned colors first. In our compiler, variables (including method parameters, method local variables, class variables and compiler generated temporaries) are prioritized by their static reference counts, having references inside loops counting 10 times more and scaled by the nesting level. After the generation of the Java IR, the compiler runs data-flow analyses and performs copy propagation and common sub-expression elimination. At this point loop structures are also identified and static reference counts are calculated. The first step of our register allocator is to build a priority list of variables using this information. In case of matching static reference counts, the priority of a variable is dictated by the order in which it was declared in the code. As we want to keep the number of virtual registers as small as possible, we 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 register (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 the case of conflicts.

In our register allocation algorithm, when assigning virtual register numbers we associate each virtual register number with the Java type of the variable it is allocated to, and we do not allow, for example, a virtual register holding an integer to later be re-used to hold a floating-point value. This restriction, although it has the counter effect of increasing the number of virtual registers, serves two main purposes. It guarantees that the mapping of a virtual register to a physical register is fixed in the run-time compilation system. Otherwise, the frequent re-mapping of virtual registers to physical registers to comply with variable types and machine register assignment restrictions will conflict with the virtual register priorities, potentially leading to an increase in spills and lower performance. Associating virtual registers to Java types also facilitates the annotations verification process. The run-time data-flow analysis to check Java type properties for normal bytecode verification may be enough to identify wrongly annotated class files.

3 Annotation-Aware JIT (AJIT) Compilation System

The rightmost portion of Figure 2 depicits our annotation aware JIT (AJIT) system. We modified the public domain JIT compiler system *Kaffe* [27] (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*'s internal intermediate representation, and changes to its SPARC code generator.

Both the original and new functionality coexist in the system, allowing the processing of annotated methods and non-annotated methods within the same class file.

As VRA annotations are derived from translating bytecodes into a RISC-like three address code, one wonders whether they are general, flexible and helpful enough to produce optimized code for different target architectures. We experimented with the Intel architecture in [16], and now with the SPARC architecture in this paper — two distinct architectures (CISC and RISC respectively). Our annotation scheme has proven to suffice the needs for generating code for these two platforms. As we experiment with other architectures our annotation types and formats will be refined accordingly.

In our AJIT system, when a class method is first called, the bytecode stream is read into a table buffer. If there is an annotation code attribute, the annotations are also read into an annotations table. Then 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. As each bytecode and its annotations bytes are read, 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 entirely skipped, or that some sub-operations be eliminated or simplified. Figure 6 shows a code example of how an `iaload` bytecode operation is translated using annotation information. The translated *Kaffe* IR operation operands are specified by virtual register numbers, extracted from the annotations bytes. Once the entire bytecode stream has been processed, SPARC native code is produced from the *Kaffe* IR. At this point, as each *Kaffe* IR operation is translated into native code, the register allocator is invoked to replace virtual register numbers with machine registers.

Our VRA annotation scheme does not need any form of run-time intermediate representation in order to produce register allocation. In our implementation we could have skipped building the *Kaffe* IR. This intermediate representation does not capture any control or data flow information. Its basic functionality is to separate the low level details of all target machines *Kaffe* can produce machine code for from the interpreter and JIT compiler translation code. Keeping the IR enabled us to write code that can be shared in the compilation and interpretation of annotated bytecodes to any target machine supported by *Kaffe*, which was very convenient at the moment of validating our ideas on annotations.

The run-time register allocator is a fast and effective algorithm that essentially maps each virtual register to a machine register, prioritizing 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 the number of physical registers is exhausted, virtual registers are mapped to temporaries on the stack. In the case of the SPARC architecture, the register allocator reserves four registers of each type (four of the global integer registers `g4-g7` and four of the floating point registers `f28-f31`) for evaluating expressions that involve variables that are not mapped into machine registers. It uses local registers `l0-l7` , global registers `g1-g3`, any unused input register `i0-i5` and floating point registers `f0-f27` during allocation. Registers `o0-o7` are not available for the allocator and are reserved for passing parameters

```
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 6: AJIT translation process for an `iaload` bytecode operation

to method calls. Our register allocation 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 method's 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.

In our experiments we observed that machine calling conventions can complicate the simple mapping-based register allocation, as it forces virtual register assignments to specific machine registers. This may break virtual register priorities, and the register allocator fixes it by spilling lower priority physical registers in case a higher priority virtual register needs a physical register and none are available. We are currently studying the effect of different calling conventions in our mapping-based dynamic register allocator.

Our current register allocation scheme does not try to minimize the cost of subroutine calls. 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 generate better native code more efficiently than traditional optimizing JIT compilers, we have shown that our framework is a good solution for improving the overall execution speed of Java programs. Annotation overhead results from many factors: (1) the larger class file size (which increases download time), (2) the interpretation of the information conveyed in the annotation bytes (see the extra processing required to build the *Kaffe* JIT IR in Figure 6), (3) additional work done by the run-time register allocator, and (4) the demand for extra resources (memory for storing annotations). Network applications are sensitive to the download time overhead, but other types of applications that do not depend on annotated class files being downloaded are not affected. In our AJIT system, the *Kaffe* run-time IR is simple to build and manipulate. Other optimizing JIT systems will need a more complex IR to enable more advanced compiler transformations. We believe that the overhead of processing the annotations, storing them and building a simple run-time IR will ultimately be 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. As a result, our register allocator is simple and fast. No time is spent on conflict graph construction, coloring nor dataflow analysis — tasks routinely performed by traditional register allocators.

4 VRA Annotations Verification Scheme

Validation of VRA annotations can be done by the Java class file verifier module in the JVM [22]. This module adds an extra security level to Java Bytecodes execution by verifying whether the class file satisfies certain defined static constraints for legal Java class files, avoiding problems such as Java operand stack overflow and underflow, invalid use of local variables, invalid types of arguments and class file version errors.

The class file verification process is performed during both class loading and linking and is broken into four phases. One of such phases is called bytecode verification in which the JVM verifier checks the code array of the code attribute for each method of the class file. The main task performed at this point is type checking to test compliance with Java properties (e.g., Gosling property [22]). For each instruction in the code the verifier records information about (1) the operand stack size and the type of each value on it; (2) the type contents of local variables. This information is collected by running a data-flow analyzer loop on a simple linear list of bytecode operations basic blocks. This algorithm performs an abstract interpretation of the program. At control flow change points the algorithm merges type information collected along different paths. As a result, new type information may be generated and more loop iterations are needed until the point where the collected type information becomes stable and is available at each instruction. Any type violation detected during the process makes the verifier report failure and no attempt is made to translate the method code.

As our VRA annotations are information generated per bytecode operation we can verify such information by extending the abstract interpretation in the bytecode verifier. Besides recording type information, the annotated bytecode verifier also records the contents of the operand stack and the local variables in terms of VR numbers assigned to them. At this same time type information for the virtual registers is also collected. The fact that in our AJBC we do not allow variables of different types share the same virtual register number makes VRA annotations easier to be checked. Any reuse of a virtual register in distinct instructions requiring different operand types characterizes an invalid annotation information. These pieces of information are necessary to validate annotations, but are not sufficient. An abstract interpretation only models the effect of the different types of operations on the operand stack and local variables array, giving information on types and information on which virtual registers are used and produced by operations. Other rules for legal annotated class files have to be incorporated, for example, to evaluate if a certain re-use of a virtual register number to represent a local variable, an operand stack value or an implicit bytecode sub-operation operand is valid.

We classify untrusted annotations as potentially valid or invalid. Figure 7 helps us to understand this classification. The top side of the figure shows the method code for the subtraction of two numbers. Figure 7(b) shows the annotated bytecode stream as it would be generated by an AJBC. An abstract interpretation of this method code up to the `isub` operation yields type and VR information as recorded in the same figure. The verification process for the `isub` operation VRA annotations compares the given annotations to the expected VRs numbers for operand stack positions and local variables and their types, all derived from the abstract interpretation. If there is no source operand or type violation, the annotations are considered as valid up to this point. The lower portion of the figure shows possible malicious changes in the bytecode stream. Figure 7(c) shows a situation in which annotations have been maliciously changed but still characterize valid annotations. Changing virtual registers does not always cause the verification process to fail and native code generated from such annotations will be correct. In cases like this, variables priorities dictated by the virual registers numbers may change and the only side effect is a less efficient native code (not noticed in this piece of code though). The situation described above is similar to malicious change in the bytecode stream that does not result in illegal class files, as for example, in Figure 7(c). In this figure we see that exchanging the `iload` operations in the second given bytecode stream makes the method compute `b-a` and not `a-b`. The bytecode stream is still legal though.

Figure 7(d) shows another situation of untrusted annotated code. In this case the annotations are consistent with the operand stack and local variables contents and seem valid. However, the reuse of `VR1` for both load of variables `a` and `b` shows a clear violation, as if these two variables are live at the same point in the bytecode stream they cannot have been assigned to the same virtual register. This extra rule that checks for registers reuse helps identifying invalid annotated bytecode streams.

The examples in Figure 7 are very simple and were used to illustrate the basic idea of our VRA annotations

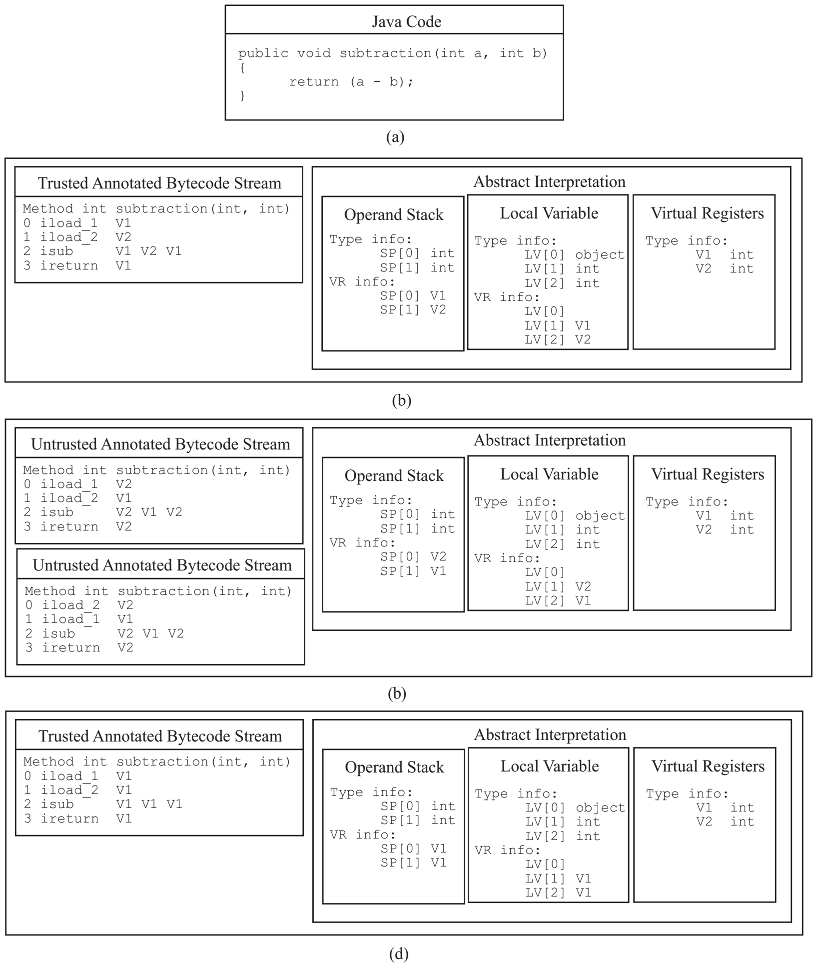

Figure 7: Valid and invalid annotated Java Bytecode streams

verification scheme. The verification process is complicated by the different annotations formats for each bytecode operation type and the formats variations.

In section 2 we discussed that VRA annotations contain virtual registers specifying the source and destination registers for the bytecode operations but also include extra registers for intermediate values calculated by some bytecode sub-operations. For example: annotations for an array element load would make explicit a register for holding the array element address computation; annotations for constants load would make explicit a register for the address of the constant. The benefit of having complex annotations formats like this is that information on redundant bytecode implicit sub-operations are exposed and the run-time code generator does not have to perform redundant computation elimination optimizations. The drawback is that VRA annotations verification gets more complicated as these sub-operations are not explicit in the abstract interpretation of bytecodes. We can solve this difficulty by creating new bytecode operations that represent the sub-operations made explicit in the annotation scheme and including their effect during the abstract interpretation (by remembering the basic block and instruction number where they are generated). These sub-operations will have no effect on the Java operand stack nor on the local variables array. Their operands are intermediate values that do not belong to any of these structures. As a solution, we decided to create an auxiliary local variable array that keeps track of such values.

These extra rules we included in our verifier help anwering the question whether the annotated bytecode stream is invalid. If an error is detected while performing the basic block checks as described above, the annotated class file can be reported as invalid. If after such checks no error was detected, it is still not safe to proceed with the translation of the annotated bytecode. Further checks are needed to completely validate the annotations. These checks have to do with checking use-definition chains (UD-chains) and definition-use chains (DU-chains) for local variables and sub-operations intermediate values. In our AJBC compiler a local variable may be assigned to different virtual registers in its different lifetime intervals. To validate VRA annotations it is not enough to to have a local variable to virtual register mapping table. Local variable to virtual register mapping is a type of information that may change from program point to program point. The complexity of checking UD-chains and DU-chains is comparable to computing the chains. A control flow graph needs to be constructed, UD-chains and DU-chains formed. For each definition of a variable, the corresponding virtual register annotation must be checked against the virtual register annotation in the use of the variable and they have to match. Furthermore, different definitions of a same variable that have a use in common must also have matching virtual register annotations, as in the virtual register allocation we use the concept of webs of variables. Other definitions of the same variable that have no use in common with a particular definition do not have to abide to this rule. These tests check whether a particular variable's VRA annotations are correctly used throughout the code. However, variables which are not live at the same time can reuse the same virtual register assignment. This implies that we need to check liveness of variables in each basic block to be able to detect wrong virtual register reuse. This information can be collected while in the abstract interpretation loop. Then, for each variable (including local variables and auxiliary local variables) we check its VRA annotations with all other variables alive in the same basic blocks. The information collected above is also enough to check VRA annotations for skipping bytecode sub-operations.

In this section we discussed the basic checks we believe necessary to validate VRA annotations. We have not implemented an annotated bytecode verifier yet. A better algorithmical description of the verification process detailing the steps and order of checks needs to be defined as well as an evaluation of is run-time complexity. We plan to keep working on this subject to present a more elaborated scheme for the next version of this paper.

5 Related Work

Various approaches are being proposed to overcome the inefficiency of translating the Java Bytecodes to native code, and thus increase the execution speed of Java programs. When compilation time is not a constraint, the most common approach is to translate the bytecodes into some higher-level intermediate form [8, 15] or language [24], and then back to native code (perhaps using an existing compiler, as in [24]). When speed of compilation is an issue, optimizing JIT compilers [1, 2, 17, 19, 27] try to improve the quality of the native code generated on the fly by adapting traditional optimization techniques to run-time code generation. Optimizations can also be applied during load-time, i.e. after bytecode generation yet before run-time translation to native code; [9] is an example of such a bytecode optimizer. Our annotation scheme is a hybrid approach in that most work is done at compile-time to retain important high-level program and optimization information, while at run-time lightweight code-improving transformations accomplish the task of generating high-quality native code.

Research in the area of developing fast run-time algorithms for traditional compiler optimizations is very active [2, 4, 20]. In the following paragraphs we overview commercial and academic systems, some of which make use of annotation schemes to aid code optimization. We also discuss how they 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. In all optimizing JIT compilers there is an attempt to develop compiler optimizations with linear time algorithms with respect to some parameter (e.g., the number of bytecode instructions, or the number of local or stack variables). Our annotation-based approach has also been designed with this in mind; our VRA annotation scheme allows run-time register allocation in linear time.

Several researchers exploit the idea of code annotations and relate to our approach. Though not designed to specifically overcome the Java Bytecode language inefficiency, these approaches could potentially be applied to this problem. In the context of dynamic code generation, code annotations in the form of programmer hints [13] or high-level language constructs extensions [23] serve as guide to where (and on what) dynamic compilation should take place. These code annotations help to build optimizing just-in-time compilers by extending to run-time the applicability of traditional compiler optimizations. Using these schemes researchers have built different algorithms for copy propagation, dead code elimination, register allocation and even advanced cross-module optimizations. Different strategies are applied to balance the tradeoff between dynamic compilation speed and the quality of the generated code.

Most directly related to our VRA annotation scheme is the work of Wall [26] on cross-module link-time register allocation. In his approach, link-time register allocation 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, Wall's approach has the overhead of building the call graph and carrying out local data flow analysis at link-time, and it depends on good usage estimates (profiling information). However, it performs global register allocation while our current implementation only works intraprocedurally.

The Intel *VTune* JIT compiler described in [2] implements register allocation of local variables, stack slots and temporaries in separate phases. Local variables are pre-allocated using a priority-based algorithm while the others are locally allocated. A technique of lazy code generation with mimic stack to keep track of the Java operand stack optimizes the code by avoiding copy operations and allowing a limited form of CSE. Our mapping-based register allocation is also a priority-based scheme and allows global register allocation of both method local variables and compiler temporaries representing values generated by bytecodes sub-operations. It is faster to implement at run-time as it dispenses with any form of code analysis. In addition, our VRA scheme can be expanded along the ideas presented in [26] to allocate class variables, while the Intel JIT compiler would need interprocedural data-flow analysis to accomplish the same, implying in an expensive run-time algorithm. In our scheme traditional copy propagation and CSE algorithms can be implemented at compile-time as annotations convey the information on how to translate a bytecode or when to skip its translation, having the further advantage in revealing redundancies implicit in the bytecode operations. A very simple array bounds check elimination algorithm was implemented in the Intel JIT compiler, handling only constant indexes. As described in [16], our run-time check annotations allow powerful subscript analysis to be performed at compile-time and easily convey this analysis information to the run-time system.

Another efficient JIT compilation system is CACAO [1]. CACAO implements pre-coloring of local variables relying on the efficient coloring of local variables done by the Java front-end (i.e., assigning the same local variable number to variables which are not active at the same time). Like Intel's JIT compiler, CACAO implements lazy code generation with operand stack simulation but carries out further analysis in the intermediate code that helps reducing copy operations and keeping temporaries in registers rather than in spill locations. Compared to our scheme, the stack analysis information that has to be computed by their algorithm is provided for free by our VRA annotations. On the other hand, their run-time register allocator takes into account the cost of subroutine calls by efficiently using the calling convention registers. This is lacking in our current scheme. Other optimizations such as instruction scheduling, method inlining and array bounds check removal are planned for CACAO, but the run-time cost of these optimizations has not been reported yet.

Kaffe [27] is a freely available JVM that runs on several platforms; it serves as the basis for our implementation work. *Kaffe*'s native code translation process identifies basic blocks and 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 based on a simple algorithm that maps Java operand stack slots and local variable slots from memory positions to machine registers as these values are referenced in the code. When it runs out of registers, the least recently used register is spilled and freed for allocation. There is no special treatment to reduce subroutine calling costs, or to exploit machine calling conventions, as CACAO does. Upon a call, copy operations are introduced to guarantee values are in the correct register and all modified slots are spilled. No other compiler optimizations are implemented. In Section 6 we demonstrate that our AJIT system outperforms *Kaffe* in terms of the quality of the generated native code.

An interesting optimizing run-time compilation system is the Slim Binary project [11, 21]. This approach proposes an architecture-neutral intermediate representation for software distribution, called *slim binaries*, that can be seen as an alternative to Java Bytecodes. The dynamic compiler for slim binaries implements code optimizations as background processes. Just like the dynamic compilation systems discussed in [13, 23], this system tries to utilize run-time information (e.g., values of variables and run-time profiling information) to perform customized optimizations. Slim binaries incorporate a more complex tree-based intermediate representation, conveying control flow information but also incurring some run-time overhead to manipulate it. Much like our annotation scheme extends the Java Bytecodes with extra information that is collected during traditional compilation, the Slim Binary representation could benefit from our annotations scheme to decrease run-time optimization costs, such as carrying extra information to aid in register allocation.

6 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 [10]; `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 JVMs available on the SPARC platform: Sun's JDK version 1.1.1 [18] and *Kaffe* JVM version 0.9.2 [27]. The execution time results are shown in Table 1. Note that the timings do not include translation nor compile time, and thus 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, the *Kaffe* JIT compiler, and our AJIT system.

The results presented in Table 1 reflect the sole effect of our VRA annotations scheme. From the two speedup columns of Table 2 we see that our annotation based approach offers speedups varying from 1.38 to 4.83 over direct interpretation, and is 17% to 100% faster than *Kaffe*'s JIT technology. Also notice that the

Benchmarks	**SUN Interpreter** (in secs)	**kaffe JIT** (in secs)	**AJIT** (in secs)
`Neighbor` 256X256 array Iterations = 1500	553.03	162.73	115.31
`EM3D` 1250 tree nodes Iterations = 200	359.84	149.86	74.51
`Bitonic Sort` 1024 tree nodes Iterations = 512	167.05	141.23	120.96
`Huffman` 30000 array nodes Iterations = 288	4690.00	1856.00	1487.00

Table 1: Benchmarks execution times (in seconds)

Benchmarks	**SpeedUp AJIT/SUN**	**SpeedUp AJIT/Kaffe**
`Neighbor`	4.80	1.41
`EM3D`	4.83	2.01
`Bitonic Sort`	1.38	1.17
`Huffman`	3.15	1.25

Table 2: Benchmarks speedups

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, along with the propagation of values and elimination of move operations. These transformations correspond to optimizations that could not be expressed in the Java Bytecodes directly. The annotations also ensured that the most important variables, such as loop index variables, were permanently assigned to machine registers throughout method execution. The smallest performance gain was observed for the code with the highest number of subroutine calls — `Bitonic Sort`, a recursive algorithm. This result is explained by the way our AJIT system, and *Kaffe* as well, handle subroutine calls during dynamic register allocation. In short, both JIT compilers do not take advantage of SPARC register windows. All active registers are saved across method calls, introducing significant overhead. Thus, it is important to note that this overhead is not an intrinsic limitation of the algorithms, but an artifact of the current implementations.

The encouraging observation we have obtained from these preliminary results is that despite many limitations of the first implementation, our AJIT system is capable of producing machine code that executes up to twice as fast as current JIT technology. By extending our VRA annotations scheme with extra information, such as registers to be saved across subroutine calls, and improving the implementation of the dynamic register allocator itself, we believe the impact of our VRA annotations will be even more significant.

We are conscious that a real validation of our annotation scheme would involve comparisons with other JIT compilers besides *Kaffe*. For this current version of the paper we thought the only fair comparison we could make was with the original *kaffe* code, as all other features of the JVM are maintained the same across AJIT and *Kaffe*, the only difference being in the code generator. To be able to compare our AJIT implementation with other JIT compilers (e.g., SUN's) we would need to isolate the effect of other JVM features implemented in the last that can interfere in the quality of the generated code and on the compilation time of the run-time register allocation algorithm. For the final version of this paper we want to define a better evaluation method for our annotation scheme. The possibilities we have been poundering are (1) to include a comparison with SUN's JIT compiler using machine code example to compare code quality in terms of register allocation; (2) implement a graph coloring like register allocation in the original *kaffe* code and show compilation cost compared to AJIT implementation. Any suggestion from the reviewer for yet another scheme to evaluate AJVM is welcome.

7 Conclusions and Future Work

Most approaches for speeding up Java execution resort to dynamic compilation (and even dynamic code re-optimization [14]). In this scenario, run-time costs must be minimized and thus it is desirable that the bulk of the compilation process be done statically at compile time. Having a rich program representation conveying, for example, dependence information to allow instruction scheduling and support for dynamic register allocation, will decrease 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 Bytecode language is a poor choice for a high-performance program representation, since it demands a more time consuming code generation process (at run-time!) in order to produce high-quality native code. We presented an approach based on code annotations that helps overcome this problem, and discussed the implementation details of our resulting annotation-aware JIT system.

Our first prototype implements the VRA annotation scheme that conveys information for dynamic register allocation. It also enables 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 minimize the cost of subroutine calls (e.g., values to be saved across procedure calls and values passed as subroutine parameters) and allows cross-module register allocation. We started with the implementation of the VRA annotations scheme because register allocation is the most important compiler optimization on today's architectures. We initially selected scientific benchmarks to test our approach given their higher sensitivity to such optimization. We will be also refining our VRA annotations verification process presented in section 4. Figure 2 mentions a number of annotation possibilities that we plan to explore in the future. These annotations support more sophisticated compiler optimizations, such as instruction scheduling and lifetime analysis for reducing garbage collection. To help evaluate these annotations we will study non-numeric Java benchmarks as well.

8 Appendix

This section lists out the Java Bytecode operation types, their corresponding Java IR sub-operations and VRA annotations formats.

8.1 Scalar Load and Store Instructions

Table 3 summarizes how scalar load and store instructions are represented. Local variable loads are represented as `nop` operations in our Java IR and the variable is directly allocated to a virtual register. Local variable stores can be represented as `move` operations or `nop` operations depending whether the store operation defines a new live range for the local variable or not. Loading of constants can be represented as loading a constant from a memory location where it is defined or as a `nop` operation, depending on the primitive type of the constant. Bytecode operations that have constants as their operands are annotated with such constant value, for integer type constants, or are annotated with the virtual register that contains the constant value, for all other types of constants. The option for further constant folding is left for the JVM if the target architecture supports operations with immediate values.

Bytecode	Java IR	VRA Annotations Format
[i,l,f,d]load [i,l,f,d]load_<n>	nop	SRC
[i,l,f,d]store [i,l,f,d]store_<n>	[i,l,f,d]mov V1, V2 [i,l,f,d]mov CONST, V1 nop	SRC-DEST CONST-DEST SRC
bipush sipush iconst_<n> iconst_m1	nop	NONE
aconst_null ldc, ldc_w, ldc2_w [l,f,d]const_<n>	amovi address-of-const, V1 [a,l,f,d]ld (V1), V2 [a,l,f,d]ld (V1), V2 nop	EXTRA-DEST SRC-DEST SRC

Table 3: Java IR and VRA annotations formats for scalar loads and stores

8.2 Arithmetic Instructions and Type Conversion Instructions

Table 4 summarizes how arithmetic, type conversion and local variable increment instructions are represented. Bynary and unary operations are represented as add, subtract, multiply, divide, remainder, negate, shift, bitwise OR, bitwise AND and bitwise exclusive OR operations defined in our Java IR. They are annotated with constant values or up to three virtual registers, representing an operation operands and result. Local variable increments are represented as add operations in the Java IR and are annotated with the virtual register allocated to the local variable. Type conversion instructions are represented in the same way as unary operations.

Bytecode	Java IR	VRA Annotations Format
[i,l,f,d]binaryOp [i,l,f,d]unaryOp	[i,l,f,d]binaryOp CONST, V1, V2 [i,l,f,d]binaryOp V1, CONST, V2 [i,l,f,d]binaryOp V1, V2, V3 [i,l,f,d]unaryOp CONST, V1 [i,l,f,d]unaryOp V1, V2	CONST-SRC-DEST SRC-CONST-DEST SRC-SRC-DEST CONST-DEST SRC-DEST
iinc	add V1, CONST, V1	SRC

Table 4: Java IR and VRA annotations formats for arithmetic operations, type conversion operations and local variable increment operation

8.3 Object Creation and Manipulation

Bytecode instructions that manipulate class instances are represented as shown in Table 5. Fields of a class are variables kept in memory and explicit load and store operations are used in our Java IR for accessing such variables. The address computation for field accesses is made explicit via Java IR operations and in the VRA annotations formats. When interprocedural virtual register allocation is implemented, class field accesses, for some safe program points (e.g., not a program point where an exception may be thrown or a method call for which the effect on the class instance is not known) can be represented the same way we do for local variables accesses (i.e., move operations or `nop` operations).

How to map array elements load and stores into our Java IR and the corresponding VRA annotations formats are shown in Table 7. For these bytecode operations me made explicit the array index calculation and the array address computation besides the actual array load or store instructions. Virtual registers representing the base array address, the array index and the element to be stored are passed as parameters to the sub-operations. The result value and intermediate values are also represented using virtual registers and correspond to the sub-operations operands. In case a load or store operation may be omitted, either because the element load has been computed before or a store back to memory is not necessary, the array access is represented as a `nop` operation and the VRA annotations bytes contain the virtual register where the array element can be found.

Creating a new instance of a class is represented as a method call in our Java IR, as shown in Table 6. There is one virtual register for representing the address of the method for creating the class instance and another for representing the newly created class object. The call to the class instance initialization method that follows an object creation is handled by another Java IR method call instruction representing the method invocation bytecode. Instructions for checking properties of class instances or array objects such as `checkcast` and `instanceOf` are also represented as Java IR method calls, as shown in Table 6.

The operation to get the length of an array is represented as a load operation in our Java IR, as shown in Table 7. One virtual register is used for the array base address, and another for the result.

Operations for creating a new array object are represented as method calls in our Java IR. These call operations take as parameters a virtual register containing the address of the method, the array dimensions represented as constants or values in virtual registers and the place where to store the newly created array reference, as summarized in Table 7.

8.4 Control Transfer Instructions

Conditional jump bytecodes are translated as a Java IR comparison operation followed by a conditional jump operation, as shown in Table 8. These sub-operations take one or two virtual register arguments as input and produce a condition value as result.

Conditional jumps that manipulate long, float and double values are represented as special Java IR

Bytecode	Java IR	VRA Annotations Format
getstatic	amovi addressOfClassField, V1 [b,c,s,i,l,d,f,a]ld (V1), V2	EXTRA-DEST
	[b,c,s,i,l,d,f,a]ld (V1), V2	SRC-DEST
	nop	SRC
putstatic	amovi addressOfClassField, V1 [b,c,s,i,l,d,f,a]st CONST, (V1)	CONST-EXTRA
	amovi addressOfClassField, V2 [b,c,s,i,l,d,f,a]st V1, (V2)	SRC-EXTRA
	[b,c,s,i,l,d,f,a]st CONST, (V1)	CONST-SRC
	[b,c,s,i,l,d,f,a]st V1, (V2)	SRC-SRC
	[b,c,s,i,l,d,f,a]mov V1, V2	SRC-DEST
	[b,c,s,i,l,d,f,a]mov CONST, V1	CONST-DEST
	nop	SRC
getfield	amovi offsetOfField, V2 aadd V1, V2, V3 [b,c,s,i,l,d,f,a]ld (V3), V4	SRC-EXTRA-EXTRA-DEST
	aadd V1, V2, V3 [b,c,s,i,l,d,f,a]ld (V3), V4	SRC-SRC-EXTRA-DEST
	[b,c,s,i,l,d,f,a]ld (V1), V2	SRC-DEST
	nop	SRC
putfield	amovi offsetOfField, V3 aadd V2, V3, V4 [b,c,s,i,l,d,f,a]st V1, (V4)	SRC-SRC-EXTRA-EXTRA
	amovi offsetOfField, V2 aadd V1, V2, V3 [b,c,s,i,l,d,f,a]st CONST, (V3)	CONST-SRC-EXTRA-EXTRA
	aadd V2, V3, V4 [b,c,s,i,l,d,f,a]st V1, (V4)	SRC-SRC-SRC-EXTRA
	aadd V1, V2, V3 [b,c,s,i,l,d,f,a]st CONST, (V3)	CONST-SRC-SRC-EXTRA
	[b,c,s,i,l,d,f,a]st V1, (V2)	SRC-SRC
	[b,c,s,i,l,d,f,a]st CONST, (V1)	CONST-SRC
	[b,c,s,i,l,d,f,a]mov V1, V2	SRC-DEST
	[b,c,s,i,l,d,f,a]mov CONST, V1	CONST-DEST
	nop	SRC

Table 5: Java IR and VRA annotations format for accessing class fields

Bytecode	Java IR	VRA Annotations Format
new	amovi addressOfNew, V1 acall V2, classType, V1, V3 acall V2, classType, V1, V3	EXTRA-DEST SRC-DEST
checkcast	amovi addressOfCheckCast, V2 call V2, classType, V1 call V2, classType, V1	EXTRA SRC
instanceof	amovi addressOfInstanceOf, V2 call V2, classType, V1, V3 call V2, classType, V1	EXTRA-DEST SRC-DEST

Table 6: Java IR and VRA annotations format for manipulating object instances

comparison operations, also shown in table 8. In this case, the virtual register annotations include the two input arguments and also the condition value result. These bytecodes could have been broken into the simple compare and branch Java IR operations however we did not find any advantage in making explicit the compare operations implicit in these bytecodes.

The unconditional branch bytecodes as `goto`, `goto_w` and `return` have counterpart Java IR operations. The unconditional branch bytecodes associated with the `finally` keyword implementation are represented by Java IR operations for method call and unconditional indirect jump. The method call specifies the address of the `finally` block and has as argument the address of the instruction following the `jsr` bytecode. The `ret` bytecode is represented as a `goto` operation which indirect label is specified by a virtual register. All these bytecodes are shown in Table 8.

Compound conditional branch bytecodes as `tableswitch` and `lookupswitch` are broken into Java IR conditional jump operations. When annotating these bytecodes, the virtual register corresponds to the key argument been tested. These bytecodes are also shown in Table 8.

8.5 Method Invocation and Return Instructions

Method return bytecodes are represented by counterpart Java IR return operations and are annotated with the virtual register containing the value to be returned. Method invocation bytecodes are mapped into Java IR method call instructions. These method calls take as argument as many virtual registers or constant values as the number of method parameters. We include other sub-operations that make explicit the computation of the address of the method. In case method calls occur refering to the same object and the same virtual registers, annotations bytes can suggest the omission of the method address computation. The method invocation bytecodes are further annotated with a virtual register containing the calculated method address and when necessary, also virtual registers for the object whose method is being invoked and the return

Bytecode	Java IR	VRA Annotations Format
[b,c,s,i,l,d,f,a]aload	ishl V2, [b,c,s,i,l,d,f,a]shiftValue, V3 iadd V3, arraySizeOffset, V3 aadd V1, V3, V4 [b,c,s,i,l,d,f,a]ld (V4), V5	SRC-SRC-EXTRA-EXTRA-DEST
	ishl CONST, [b,c,s,i,l,d,f,a]shiftValue, V3 iadd V3, arraySizeOffset, V3 aadd V1, V3, V4 [b,c,s,i,l,d,f,a]ld (V4), V5	CONST-SRC-EXTRA-EXTRA-DEST
	aadd V1, V2, V3 [b,c,s,i,l,d,f,a]ld (V3), V4	SRC-SRC-EXTRA-DEST
	[b,c,s,i,l,d,f,a]ld (V1), V2	SRC-DEST
	nop	SRC
[b,c,s,i,l,d,f,a]astore	ishl V2, [b,c,s,i,l,d,f,a]shiftValue, V4 iadd V4, arraySizeOffset, V4 aadd V1, V4, V5 [b,c,s,i,l,d,f,a]st V3, (V5)	SRC-SRC-SRC-EXTRA-EXTRA
	ishl V2, [b,c,s,i,l,d,f,a]shiftValue, V3 iadd V3, arraySizeOffset, V3 aadd V1, V3, V4 [b,c,s,i,l,d,f,a]st CONST, (V4)	CONST-SRC-SRC-EXTRA-EXTRA
	aadd V1, V2, V4 [b,c,s,i,l,d,f,a]st V3, (V4)	SRC-SRC-SRC-EXTRA
	aadd V1, V2, V3 [b,c,s,i,l,d,f,a]st CONST, (V3)	CONST-SRC-SRC-EXTRA
	[b,c,s,i,l,d,f,a]st V1, (V2)	SRC-SRC
	[b,c,s,i,l,d,f,a]st CONST, (V1)	CONST-SRC
	nop	SRC
arraylength	aadd V1, arraySizeOffset, V2 ild (V2), V3	SRC-DEST
newarray	amovi addressOfNewArray, V2 acall V2, arrayType, V1, V3	SRC-EXTRA-DEST
	acall V2, arrayType, V1, V3	SRC-SRC-DEST
anewarray	amovi addressOfANewArray, V2 acall V2, arrayType, V1, V3	SRC-EXTRA-DEST
	acall V2, arrayType, V1, V3	SRC-SRC-DEST
multianewarray	amovi addressOfMultiANewArray, V1 acall V1, arrayType, V2, V3..., CONST,... Vn	[SRC/CONST]-EXTRA-DEST
	acall V1, arrayType, V2, V3..., CONST,... Vn	[SRC/CONST]-SRC-DEST

Table 7: Java IR and VRA annotations formats for accessing array elements

Bytecode	Java IR	VRA Annotations Format
if_<eq,ne,lt,le,ge,gt>	icmp_<eq,ne,lt,le,ge,gt> V1, 0, V2 br V2 trueLabel falseLabel	SRC
if_<null,nonnull>	acmp_<eq,ne> V1, null, V2 br V2 trueLabel falseLabel	SRC
if_icmp<eq,ne,lt,le,ge,gt>	icmp_<eq,ne,lt,le,ge,gt> V1, V2, V3 br V3 trueLabel falseLabel icmp_<eq,ne,lt,le,ge,gt> CONST, V1, V2 br V2 trueLabel falseLabel icmp_<eq,ne,lt,le,ge,gt> V1, CONST, V2 br V2 trueLabel falseLabel	SRC-SRC CONST-SRC SRC-CONST
if_acmp<eq,ne>	acmp_<eq,ne> V1, V2, V3 br V3 trueLabel falseLabel	SRC-SRC
lcmp fcmpl fcmpg dcmpl dcmpg	[l,f,d]cmp V1, V2, V3 [l,f,d]cmp CONST, V1, V2 [l,f,d]cmp V1, CONST, V2	SRC-SRC-DEST CONST-SRC-DEST SRC-CONST-DEST
goto gotow	goto label	none
return	return	none
jsr jsr jsr_w	amovi addressOfFinally, V1 amovi addressOfNextInstruction, V2 call V1, V2 call V1, V2	EXTRA-DEST SRC-DEST
ret	goto V1	SRC
tableswitch lookupswitch	icmp V1, CONST, V2 br V2 trueLabel falseLabel	SRC

Table 8: Java IR and VRA annotations formats for control transfer instructions

value. Table 9 shows the correspondence between these bytecodes, Java IR operations and VRA annotations formats.

Bytecode	Java IR	VRA Annotations Format
invokevirtual	aadd V1, methodTableOffset, V2 ald (V2), V2 aadd V2, methodOffset, V3 ald (V3), V3 [b,c,s,i,l,d,f,a]call V3, V1, V4, ... CONST ... Vn	SRC-EXTRA-EXTRA-[[SRC/CONST][DEST]]
	aadd V1, methodOffset, V2 ald (V2), V2 [b,c,s,i,l,d,f,a]call V2, V1, V3,..CONST..Vn	SRC-EXTRA-[[SRC/CONST][DEST]]
	[b,c,s,i,l,d,f,a]call V2, V1, V3,..CONST..Vn	SRC-SRC-[[SRC/CONST][DEST]]
invokestatic	amovi methodAddress, V1 [b,c,s,i,l,d,f,a]call V1, V2,..CONST..Vn	EXTRA-[[SRC/CONST][DEST]]
	[b,c,s,i,l,d,f,a]call V1, V2,..CONST..Vn	SRC-[[SRC/CONST][DEST]]
invokespecial	amovi methodAddress, V2 [b,c,s,i,l,d,f,a]call V2, V1,..CONST..Vn	EXTRA-[[SRC/CONST][DEST]]
	[b,c,s,i,l,d,f,a]call V2, V1,..CONST..Vn	SRC-[[SRC/CONST][DEST]]
invokeinterface	amovi methodAddress, V2 [b,c,s,i,l,d,f,a]call V2, V1,..CONST..Vn	EXTRA-[[SRC/CONST][DEST]]
	[b,c,s,i,l,d,f,a]call V2, V1,..CONST..Vn	SRC-[[SRC/CONST][DEST]]
[b,c,s,i]return [l,d,f,a]return	[b,c,s,i,l,d,f,a]return V1	SRC

Table 9: Java IR and VRA annotations formats for method invocation and return instructions

8.6 Operand Stack Management Instructions

These bytecodes manipulate the Java stack and represent copy, elimination or swap of values placed on the Java operand stack. They are basically employed as fast bytecode operations. There is no need to represent them in Java IR operations or annotate them.

8.7 Throwing and Handling Exceptions

The `athrow` bytecode is annotated with a virtual register representing the reference to the object being thrown. We map the `throw` keyword into a Java IR method call. Methods that can throw exceptions have an extra return value that represents the thrown object. Catch clauses are mapped into conditional jumps. We are modifying our current implementation of AJBC and AJIT to support exception handling as summarized above. A more detailed explanation of how we deal with exceptions will be provided in the final version of this paper.

8.8 Synchronization

We represent synchronization at statement level by mapping the `monitorenter` and `monitorexit` bytecodes into a Java IR method call. This method call takes a virtual register representing the object requiring synchronized access as argument. Our current implementation processes methods ignoring synchronization at method level. A more detailed study on how to represent synchroniztaion and the implications on our annotation scheme will be provided in the final version of this paper.

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] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, and Michael Hind. The Jalapeno Dynamic Optimizing Compiler for Java. *In Proceedings of the ACM Java Grande Conference*, pages 129–141, June 1999.

[5] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices*, 17(6):201–107, June 1982.

[6] 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.

[7] F. C. Chow and J. L. Hennessy. A Priority-based Coloring Approach to Register Allocation. *ACM TOPLAS*, 12(4):501–536, October 1990.

[8] M. Cierniak and W. Li. Optimizing Java Bytecodes. *Concurrency: Practice and Experience*, 9(11), November 1997.

[9] L. R. Clausen. A Java Bytecode Optimizer Using Side-effect Analysis. *Concurrency: Practice and Experience*, 9(11), November 1997.

[10] 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.

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

[12] J. Gosling, Bill Joy, and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[13] 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.

[14] David Griswold. The Java HotSpot Virtual Machine Architecture, March 1998. See whitepaper at http://www.javasoft.com/products/hotspot/.

[15] 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.

[16] 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.

[17] Microsoft Inc. The Microsoft Virtual Machine for Java.
See http://www.microsoft.com/java/sdk/.

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

[19] Symantec Inc. Just in Time Compiler for Windows 95/NT. See http://www.symantec.com.

[20] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, Implementation and Evaluation of Optimizations in a Just-In-Time Compiler. *In Proceedings of the ACM Java Grande Conference*, pages 119–128, June 1999.

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

[22] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1997.

[23] 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.

[24] 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.

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

[26] D. W. Wall. Global Register Allocation at Link-Time. *In Proc. ACM SIGPLAN'86 Symp. on Compiler Construction*, pages 264–275, June 1986.

[27] Tim Wilkinson. Kaffe: A Free JIT virtual machine to run Java code.
See http://www.transvirtual.com.