An Annotation-aware Java Virtual Machine Implementation

Ana Azevedo[*], Alex Nicolau

University of California, Irvine
aazevedo, nicolau@ics.uci.edu

Joe Hummel

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

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 language has no mechanism to indicate which sub-operations in the Java bytecode stream are redundant or subsumed by previous ones. As a consequence, the Java bytecode language inhibits the expression of important compiler optimizations, including register allocation and instruction scheduling.

The Java bytecode stream generated by a Java bytecode compiler is a significantly under-optimized program representation. The most common solution to overcome this inefficiency 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 a Java bytecode compiler. These annotations carry information concerning compiler optimizations. During the translation process, an annotation-aware Java Virtual Machine (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 a prototype of an annotation-aware JVM consisting of an annotation-aware JIT compilation system. We conclude the paper showing performance results comparing our system with other JVMs running on SPARC architecture.

1 Introduction

The Java bytecode language is 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 [16], or compiled into native code using traditional ahead-of-time [14] or just-in-time

[*]This work supported in part by CAPES.

compilers [1, 2, 4, 13, 17, 26, 27]. The safety features of the language are based on the security violation checks performed at load and run time [19]. Such checks include enforcement of method and variable 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, the language must also yield efficient execution on a wide range of machine architectures. Unfortunately this is the weakest aspect of Java 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 compiler front-end into target machine code results in low-quality code.

The first limitation in expressing compiler optimization is the stack model that the Java bytecode language implements. 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 serializes 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 bytecode language 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 bytecode compiler 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 language, 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, straightforward compiler optimizations such as common sub-expression elimination, array bounds check elimination and loop-invariant code removal have limited expressiveness in Java bytecode.

To demonstrate these limitations, consider the example in Figure 1. This example assumes that a RISC-like, three address code Intermediate Representation (IR) [1] is used in the Java bytecode compiler. The leftmost column shows the unoptimized intermediate code [2] 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 intermediate

[1] This example is based on the Java IR we designed. More details on the Java IR can be found in the Appendix Section 7.

[2] Array bound checks have been omitted.

code, 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 stream. For example, the sub-operations comprising array element accesses represent common sub-expressions and thus 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 implicit 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 bytecode as a language for program representation

The implication is that even though the Java bytecode compiler 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 [15]; in this paper we present the details of the implementation of our AJIT compilation 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 potential run-time overhead generated by our annotation scheme including an explanation on the basic security checks that are necessary to validate untrusted annotated Java class files. We conclude 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 Bytecode Compiler (AJBC), discuss the types and formats of the annotations already implemented, and give an overview of other annotations been designed in our compiler framework. We provide details about our annotations for register allocation and present the compile-time register allocatation algorithm that produces the annotations in support of dynamic register allocation. In Section 3 we discuss our AJIT compilation system and show how it uses annotations to implement run-time register allocation and produce native code. In this section we also talk about the sources of potential run-time overhead associated with our annotation scheme. Section 4 presents some preliminary results on the performance of our AJIT system. In Section 5 we discuss related work, followed by our conclusions and future work directions in Section 6.

2 Annotation-Generating Java Bytecode 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. Our annotation types and formats vary with the kind of information that needs to be conveyed to the run-time system. For example, it may consist of high-level program information that is not expressible in the language chosen for software distribution or compiler analysis information that is too time consuming to produce at run-time. Figure 2 gives an overview of a general annotation-generating Java bytecode 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 is produced. Once this stage has been reached, each operation (or sequence of operations) is translated into an optimized Java bytecode stream. In this process, the annotation generator component of the compiler framework uses the optimized IR, along with the data provided by various compiler analyses, to produce a set of annotations. Finally, the compiler performs a mapping phase in which the IR operations and annotations are paired with the bytecodes forming the optimized annotated class files.

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 the corresponding Java IR operation (or sequence of operations). 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 [19].

Our annotation-generating compiler was built using *guavac* Open Source (version 0.3.1) Java bytecode compiler [23]. From the Java source code, this compiler generates a parse tree and produces Java bytecode. We augmented the compiler by (a) introducing functions for building and manipulating our three-address code IR, (b) implementing compiler optimizations for common sub-expression elimination, copy propagation and register allocation, and (c) designing a VRA annotation generator. This paper focuses in particular on the VRA annotations. The remaining annotations, as presented in Figure 2 represent work in progress.

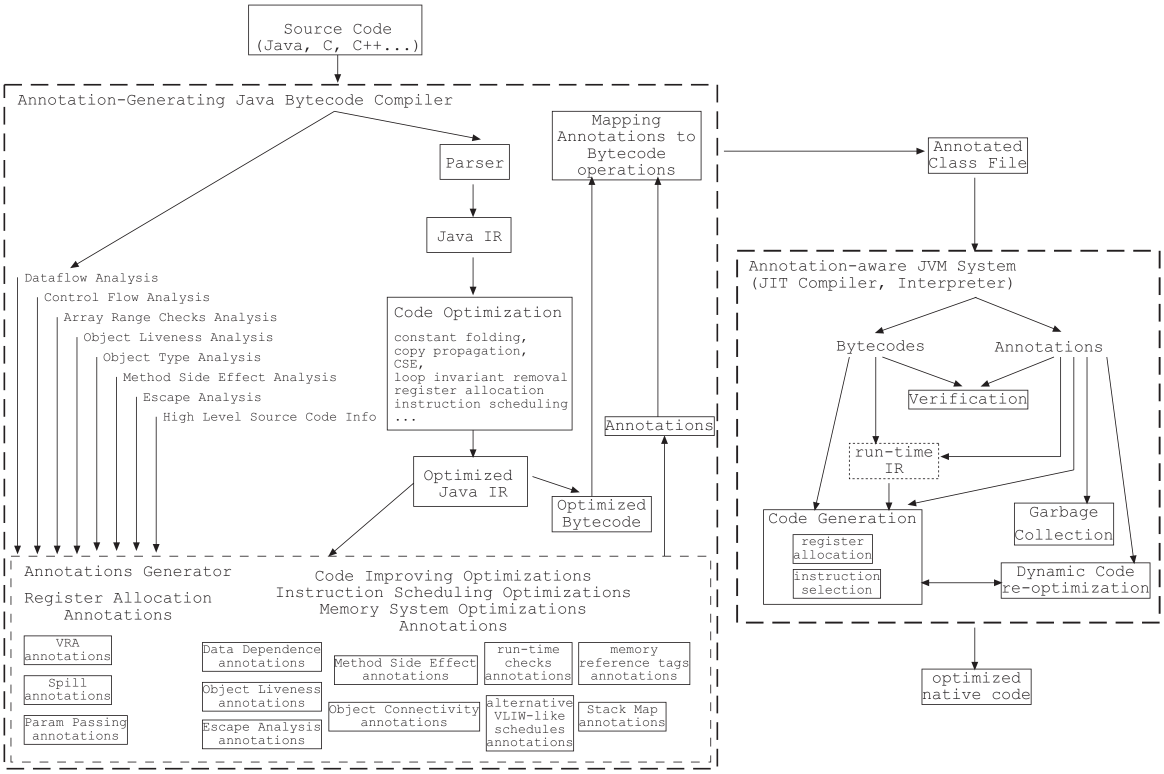

Figure 2: Annotation-generating Java Bytecode Compiler (AJBC) and Annotation-aware JVM system

Virtual Register Allocation annotations represent the result of performing register allocation assuming an infinite number of symbolic 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 [3] 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.

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

Each instruction defined in the Java bytecode 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 values implicitly calculated by the bytecode sub-operations (e.g., array address calculation in an array element access operation). Each bytecode operation type has a distinct VRA annotation format. A format may have multiple variations that indicate 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 sub-operation should be skipped (e.g. when a previous operation has already computed the needed value).

Figure 3 shows an example of correspondence among bytecodes, Java IR and VRA annotation formats. Each `SRC`, `EXTRA` and `DEST` fields holds 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 Java IR and VRA annotations for `iaload` bytecode

Figures 5 and 7 from Appendix Section 7 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, annotated as `SRC`, or move operations (move between symbolic registers or move of a constant value into a symbolic register) annotated as `SRC-DEST`, `CONST-DEST`, 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 Java bytecode compiler and accessed via load and store operations, as shown in Figure 7 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 designed 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. For some types of class member variables and for some safe program points we attempt a better virtual register allocation and we also designed annotations that express such cases (see Appendix Section 7.3).

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 7. In the design of the Java IR and VRA annotations we made some assumptions on object layout and dynamic dispatching (e.g., see Java IR operations for array element access operations and Java method calls) based on the most used conventions. However such assumptions do not compromise the portability of the annotations (i.e., annotations for the intermediate bytecode sub-operations can be ignored by the underlying JVM).

2.1 Compile-Time Register Allocation

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. In our annotation-generating compiler we implement a modified priority-based graph-coloring algorithm. A priority-based coloring algorithm [5] 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 (method local variables, class variables, stack slots and compiler generated temporaries) are prioritized by their static reference counts, having references inside loops counting 10 times more and scaled by the loop 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 low 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 virtual register assignment algorithm picks variables from the priority list and assigns virtual register numbers to them, reusing lowest virtual register numbers or creating a new virtual register number in 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 annotation verification process. We use the run-time data-flow analysis that checks Java bytecode type properties for normal bytecode verification to identify incorrect annotated class files.

When trying to do machine independent optimizations we lose opportunity to be as aggressive as traditional native compilers. In particular, our virtual register allocation scheme does not allocate registers taking into account call costs or spill code minimization. Both interfere in the quality of the register allocation and most variations of the traditional graph coloring algorithm try to address these factors. In our compile-time register allocation we do not produce spill code as we work with an infinite number of symbolic registers and we do not make any assumption on calling conventions.

3 Annotation-aware Java Virtual Machine System

The rightmost portion of Figure 2 depicts our annotation-aware JVM which consists of an annotation-aware JIT (AJIT) compiler. To implement our annotation scheme we modified *Kaffe* (version 0.9.2) code which is distributed freely under the GNU Public License [26]. The changes concentrated on a few number of files and consisted of the design of a new register allocator, modifications to the generation of *Kaffe*'s internal pseudo machine instructions 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 [15], and now with the SPARC architecture in this paper — two distinct architectures (CISC and RISC respectively). Our annotation scheme has proven to be sufficient for generating code for these two platforms. As we experiment with other architectures our annotation types and formats will be further validated and refined.

The AJIT compiler invokes different translation routines depending on the presence of the annotation

```
define_insn(IALOAD)
{
/*
   ..., array ref, index -> ..., value
   VRA annotation format: header (byte) + data (sequence of bytes)
*/

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

  if (a.header == SRC_SRC_EXTRA_EXTRA_DEST){
     index = *(a.VRAData); objref = *(a.VRAData+1);
     extra1 = *(a.VRAData+2); extra2 = *(a.VRAData+3); dest = *(a.VRAData+4);

     annotated_lshl_int_const(vrslots[extra1].slots, vrslots[index].slots, SHIFT_jint);
     if (object_array_offset !=0)
        annotated_add_int_const(vrslots[extra1].slots, vrslots[extra1].slots, object_array_offset);
     annotated_add_ref(vrslots[extra2].slots, vrslots[objref].slots, vrslots[extra1].slots);
     annotated_load_int(vrslots[dest].slots, vrslots[extra2].slots);

  }else if (a.header == CONST_SRC_EXTRA_EXTRA_DEST){
     const = *(a.VRAConst); objref = *(a.VRAData);
     extra1 = *(a.VRAData+1); extra2 = *(a.VRAData+2); dest = *(a.VRAData+3);

     annotated_move_int_const(vrslots[extra1].slots, (const<<SHIFT_jint), NULL);
     if (object_array_offset !=0)
        annotated_add_int_const(vrslots[extra1].slots, vrslots[extra1].slots, object_array_offset);
     annotated_add_ref(vrslots[extra2].slots, vrslots[objref].slots, vrslots[extra1].slots);
     annotated_load_int(vrslots[dest].slots, vrslots[extra2].slots);

  }else if (a.header == SRC_SRC_EXTRA_DEST){
     objref = *(a.VRAData); src2 = *(a.VRAData+1); extra = *(a.VRAData+2); dest = *(a.VRAData+3);

     annotated_add_ref(vrslots[extra].slots, vrslots[objref].slots, vrslots[src2].slots);
     annotated_load_int(vrslots[dest].slots, vrslots[extra].slots);

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

Figure 4: AJIT translation process for `iaload` bytecode

attribute allowing both annotated and non-annotated methods to be invoked and translated during the execution of a Java application. The process of producing native code from annotated Java bytecode is done in a single pass over the bytecode stream. As each bytecode and its annotation 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 skipped entirely, or that some sub-operations be eliminated or simplified. Figure 4 shows how the original *Kaffe* code has been modified to handle an annotated `iaload` bytecode operation. 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 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 possible target machine code from the interpreter and JIT compiler translation functions. 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 when validating our ideas on annotations.

The run-time register allocation is a mapping-based 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 physical registers are exhausted, virtual registers are mapped to temporaries on the stack. As can be noticed, the complexity of our run-time algorithm is linear in the number of virtual registers.

The first task performed in the translation of an annotated method is the initialization of a mapping table used as an auxiliary data structure. The mapping table stores information on a virtual register number, the corresponding machine register, and the stack pointer offset that should be used in case of spilling. There are some details in the initialization of the mapping table to correctly handle the SPARC calling conventions. These details are taken care of in the method's prologue and in the translation of bytecodes for accessing method local variables. Dealing with such details results in changing the mapping table as we force virtual register to machine register mapping. As a consequence virtual register priorities may break which may require further fixing later in the translation process by spilling a lower priority virtual register that has been mapped to a physical register to free the register for a higher priority virtual register.

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 to method calls.

Our current register allocation scheme does not try to minimize the cost of method calls. At method calls, copy operations are generated to move values into the correct SPARC output registers and all active registers are spilled. Our annotation scheme can be modified 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 method 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 method calls by spilling only the registers that are later referenced in the program.

3.1 Annotations Run-Time Overhead

To prove that our AJIT system is an acceptable engineering solution we need to quantify the overhead of manipulating the annotated bytecode stream and the overhead of our mapping-based register allocation in the process of generating optimized native code on the fly. Annotations potential overhead results from many factors: (1) the larger class file size which increases download time; (2) the time spent verifying non-trusted annotated class files; (2) the time spent in the interpretation of the information conveyed in the annotation bytes (see the extra processing required to build the *Kaffe* JIT IR in Figure 4); (3) the time spent in the mapping-based dynamic register allocation 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, *Kaffe* run-time IR is simple to build and manipulate. Other optimizing JIT compilers 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. Besides, our run-time register allocation algorithm is an algorithm that obeys a defined mapping rule and only manipulates a mapping table. 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.

Among the above listed potential sources of overhead, the annotation verification cost is the one of most concern. Annotation verification implies checking virtual registers type and checking virtual registers use and re-use. To collect virtual registers type information can be done by extending the abstract interpretation in the traditional Java bytecode verifier. Besides collecting the type of the contents of operand stack slots and local variables, the Java bytecode verifier can also compute the type of the contents of virtual registers (including the virtual registers from bytecode implicit sub-operations). The fact that in our AJBC we do not allow variables of different types to share the same virtual register number makes VRA annotations easier to be checked. Any reuse of a virtual register in distinct bytecode operations requiring different operand types indicates an invalid annotation. Virtual registers type information is cheaply obtained as a byproduct from the traditional bytecode verification.

Other rules for legal annotated class files have to be defined to evaluate if a certain re-use of a virtual register to represent a new local variable, an operand stack value or an implicit bytecode sub-operation operand is valid. Checking such rules is the expensive part of the annotation verification process. The different annotation formats carrying information on redundant sub-operations also makes annotation verification more complicate. To accomplish this verification task compiler analyses that would be necessary to compute at run-time are UD-chains, DU-chains and liveness analyses. An alternative solution to avoid computing such analyses would be to define proofs for annotations in a way similar to the proofs generated by certifying compilers [11, 20] to validate optimized native code. Annotation verification is still work in progress in our research group.

4 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 [8]; `Huffman`, a character string compression and decompression application; and `Bitonic Sort`, which builds a binary tree and then performs bitonic sorting (recursively) [3]. [26]. The speedup results are shown in Table 1. When collecting the timings and measuring the speedups we did not include translation nor compile time, and thus the results represent the quality of the generated code. The programs were compiled using our annotation-generating Java bytecode compiler and then executed using JVMs available on the SPARC platform: SUN *JDK* interpreter version 1.1.6, SUN *HotSpot* JVM version 1.2.2, *Kaffe* JIT compiler version 0.9.2, and our AJIT system.

Benchmarks	SpeedUp AJIT/Sun Interpreter 1.1.6	SpeedUp AJIT/Kaffe JIT 0.9.2	SpeedUp HotSpot 1.2.2/Kaffe JIT 0.9.2	Class File Size Increase
Neighbor 256x256 array 1500 iterations	5.16	1.41	1.40	1.63
EM3D 1250 tree nodes 200 iterations	4.86	2.01	1.50	1.42
Bitonic Sort 1024 tree nodes 512 iterations	1.25	1.17	2.66	1.49
Huffman 30000 array nodes 288 iterations	3.19	1.25	2.73	1.58

Table 1: Benchmark speedups and class file size increase

From the two first speedup columns in Table 1 we notice that our annotation based approach offers speedups varying from 1.25 to 5.16 over direct interpretation, and is 17% to 100% faster than *Kaffe* JIT technology. Both our AJIT system and *Kaffe* are baseline compilers that only do register allocation. We think the most fair comparison we can make is 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. The third speedup column in Table 1 compares *Kaffe* system performance with SUN *HotSpot*. SUN *HotSpot* optimizing compiler implements global register allocation via graph coloring, a number of traditional compiler optimizations (e.g., common sub-expression elimination, loop invariant removal, constant propagation, and dead-code elimination) and possibly some object oriented language optimizations (e.g., method inlining). The individual effect of each of these optimizations could not be filtered out and the quality of the generated code reflects the combined effect.

In relation to *Kaffe*, the best AJIT system speedups were achieved for codes consisting of basic loops iterating over array-based or pointer-based data (`Neighbor`, `EM3D` and `Huffman`). The smallest performance gain was observed for the code with the highest number of method calls — `Bitonic Sort`. This result is explained by the way our AJIT system, and *Kaffe* as well, handle method calls during dynamic register allocation. In short, both JIT compilers do not take advantage of SPARC calling conventions and do not try to minimize spill code. At method calls, copy operations are generated to place values in the correct parameter passing registers and all live registers are spilled. Looking at SUN *HotSpot* performance when compared with *Kaffe*'s we notice that our AJIT compiler performs as good as SUN's JVM in two cases and presents much lower performance for the other two benchmarks, which are those that involve frequent method calls. This last result indicates a point of performance bottleneck in our scheme. By extending our VRA annotation scheme with extra information, such as virtual registers to be saved across method calls and which virtual registers are used as method call parameters can improve the impact of our VRA annotation scheme by reducing the number of copy operations and amount of spilling.

The last column in Table 1 shows that under the current encoding scheme annotated class files have an average of 53% size increase. We have not optimized our encoding scheme and we believe more compact representation is possible.

5 Related Work

Various approaches are being proposed to overcome the inefficiency of translating the Java bytecode 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 Java bytecode into some higher-level program representation [6, 14, 22] and then finally to native code (perhaps using an existing compiler, as in [22]). When speed of compilation is an issue, optimizing JIT compilers [1, 2, 4, 13, 17, 26, 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; [7, 9] are examples of Java bytecode optimizers. In the following paragraphs we overview commercial and academic systems, some of which make use of annotation schemes to aid code optimization.

Several researchers exploit the idea of code annotations and relate to our approach. In the context of selective dynamic compilation, code annotations in the form of programmer hints [12] or high-level language constructs extensions [21] serve as guide to how to handle language constructs during run-time code specialization and where (and on what) dynamic compilation should take place. Just like in our annotation scheme these systems use annotations to balance the tradeoff between dynamic compilation speed and the quality of the generated code. However we are designing annotations that can be automatically generated by the compiler, instead of relying on skilled programmers. Our annotation-aware run-time system is also simpler in terms code generation complexity and system footprint.

Most directly related to our VRA annotation scheme is Wall's work [25] on cross-module link-time register allocation. His objective is to solve conflicts in register assignment caused by the separate compilation of individual modules composing an application program. He corrects such conflicts via global register allocation at link-time using annotations encoded in the native code to convey register assignment relocation information.

Looking at the information publicly available for most current JVM implementations none of these systems, with the exception of SUN *HotSpot* [13], attempt to apply graph coloring heuristics for run-time register allocation. The cost of graph coloring heuristics can be very expensive reaching complexity potentially quadratic in the number of register candidates (for Java bytecode programs the register candidates include local variables, stack slots and compiler temporaries). JVM systems employ cheaper register allocation algorithms resorting to either simple local register allocation as in [26] or region-based register allocation as in [4, 27].

Kaffe [26] is the virtual machine that serves as basis for our implementation. Its register allocation is a simple algorithm that maps Java operand stack slots and local variable slots to memory positions on the stack of the translated method. When the allocator runs out of machine registers, the least recently used register is spilled and freed for allocation. There is no special treatment to reduce method call costs, or to exploit machine calling conventions. Upon a method call, copy operations are introduced to guarantee values are in the correct registers and all modified registers are spilled. The Intel's 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 as in *Kaffe*. Another interesting JVM implementation is *CACAO* JIT compiler [1]. *CACAO* also implements fixed pre-coloring of local variables relying on the efficient coloring of local variables done by the Java bytecode compiler. Both Intel's JIT compiler and *CACAO* implement lazy code generation with operand stack simulation to keep track of the Java operand stack contents. This information helps to combine instructions, eliminate copy operations and spilling, better use the machine calling conventions and reduce method call costs. The IBM *Latte* [27], IBM *Jalapeno* [4] and IBM Japan JIT compiler [17] systems implement an interval coloring algorithm and linear scan register allocation algorithms [21, 24], respectively. These algorithms are faster than graph coloring heuristics because they compute simplified liveness analysis. For small regions of code with a lot of variables these algorithms perform well, but as the size of the region increases, the performance degrades as compared to graph coloring approaches.

An interesting research work on dynamic compilation is the *Slim Binary* project [10, 18]. It proposes an architecture-neutral intermediate representation for software distribution that can be seen as an alternative to Java bytecode. *Slim Binary* incorporates a more complex tree-based intermediate representation as compared to Java bytecode, incurring extra run-time overhead to manipulate it. *Slim Binary* code is dynamically compiled to native code just like Java bytecode. Much like our annotation scheme extends the Java bytecode with extra information that is collected during off-line traditional compilation, the *Slim Binary* representation could benefit from our annotation scheme for reducing run-time optimization costs.

6 Conclusions and Future Work

Most approaches for speeding up Java execution resort to dynamic compilation and run-time code re-optimization. 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 and we presented an approach based on code annotations that helps to overcome this problem. We discussed in detail the implementation of our resulting annotation-aware JIT system.

Our first annotation-aware JVM prototype implements a virtual register allocation annotation scheme that conveys information for dynamic register allocation. It also enables some basic code improving optimizations by identifying and eliminating redundant computation and allowing propagation of values. Preliminary results show that we produce code that can be competitive with the performance of the best commercial JVM implementation in the market and indicate where the performance bottleneck of this type of machine independent optimization is. We plan to extend our VRA annotation scheme by incorporating information that minimizes call costs and the amount of spilling. We are also working on a VRA annotation verification process. We have interest in other Java performance problems, besides register allocation optimization, and we have identified a number of annotation candidates that we plan to exploit in the future using our annotation scheme.

7 Appendix

This section lists out the Java Bytecode operation types, their corresponding Java IR sub-operations and VRA annotation formats as implemented in the first prototype of our AJIT system.

7.1 Scalar Load and Store Instructions

Figure 5 summarizes how scalar load and store instructions are represented. Local variable load is represented as `nop` operation in our Java IR and the variable is directly allocated to a virtual register. Local variable store can be represented as copy or `nop` operations depending on 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 stored 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 the other types of constants. The option for further constant folding is left for the JVM which depends on the target architecture support for operations with immediate values.

7.2 Arithmetic Instructions and Type Conversion Instructions

Figure 6 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 a constant value and/or up to three virtual registers, representing the operation operands and result. Local variable increment is represented as an add operation in the Java IR and is annotated with the virtual register allocated to the local variable. Type conversion instructions are represented in the same way as unary operations.

7.3 Object Creation and Manipulation

Bytecode instructions that manipulate class instances are represented as shown in Figure 7. Class fields 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 annotation formats. For some class variable types and safe program points (e.g., unsafe program points are places where an exception may be thrown or there is a method call for which the effect on the class field is not known) class field accesses can be represented the same way we represent local variable accesses (i.e., copy or `nop` operations).

Creating a new instance of a class is represented as a method call in our Java IR, as shown in Figure 8. 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 Figure 8.

How to map array element load and store bytecodes into our Java IR and the corresponding VRA annotation formats are shown in Figure 9. For these bytecode operations we made explicit the array index calculation and the array address computation besides the actual array load or store sub-operation. 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 annotation bytes contain the virtual register where the array element can be found. The operation to get the length of an array is represented as a load operation in our Java IR, as shown in Figure 10. 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 Figure 10. Observe that we made assumptions about object layout when representing the bytecode sub-operations for manipulating array objects. These assumptions are based on most used conventions and do not compromise the portability of the annotations. (e.g., annotations for the intermediate bytecode sub-operations can be ignored by the underlying JVM).

7.4 Control Transfer Instructions

A Conditional jump bytecode is translated as a Java IR comparison operation followed by a conditional jump operation, as shown in Figure 11. These sub-operations take a constant value and/or virtual registers as operands and produce a condition value in a virtual register as a result. Conditional jumps that manipulate long, float and double values are represented as special Java IR comparison operations, also shown in Figure 11. 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. In all these cases, the virtual register annotations include up to two operand arguments.

The unconditional branch bytecodes as `goto`, `goto_w` and `return` have counterpart Java IR operations. 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 being tested. Bytecode instructions associated with the implementation of `finally` keyword (`jsr` and `ret`) are represented as jump operations in our Java IR and are not annotated. These bytecodes are also shown in Figure 11.

Bytecode	Java IR	VRA Annotation Formats
[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,d,f,a}mov V1, V2	SRC DEST
	{i,l,d,f,a}mov CONST, V1	CONST DEST
	nop	SRC
bipush sipush iconst_<n> iconst_m1	nop	NONE
aconst_null ldc, ldc_w, ldc2_w [l,f,d]const_<n>	amovi “addressOfConst”, V1 {l,d,f,a}ld (V1), V2	EXTRA DEST
	{l,d,f,a}ld (V1), V2	SRC DEST
	nop	SRC

Figure 5: Java IR and VRA annotation formats for scalar loads and stores

Bytecode	Java IR	VRA Annotation Formats
[i,l,f,d]binaryOp [i,l.f,d]unaryOp	{i,l,d,f}binaryOp CONST, V1, V2	CONST SRC DEST
	{i,l,d,f}binaryOp V1, CONST, V2	SRC CONST DEST
	{i,l,d,f}binaryOp V1, V2, V3	SRC SRC DEST
	{i,l,d,f}unaryOp CONST, V1	CONST DEST
	{i,l,d,f}unaryOp V1, V2	SRC DEST
iinc	iadd V1, CONST, V1	SRC

Figure 6: Java IR and VRA annotation formats for arithmetic, type conversion and local variable increment operations

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 “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”, 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 7: Java IR and VRA annotation formats for class instance field accesses

Bytecode	Java IR	VRA Annotation Formats
new	amovi addressOfNew, V1 acall V1, classType, V2	EXTRA DEST
	acall V1, classType, V2	SRC DEST
checkcast	amovi addressOfCheckCast, V1 call V1, classType, V2	SRC EXTRA
	call V1, classType, V2	SRC SRC
instanceof	amovi addressOfInstanceOf, V1 icall V1, classType, V2	SRC EXTRA DEST
	icall V1, classType, V2	SRC SRC DEST

Figure 8: Java IR and VRA annotation formats for manipulating object instances

Bytecode	Java IR	VRA Annotation Formats
[b,c,s,i,l,f,d,a]aload	ishl V2, [b,c,s,i,l,f,d,a]shiftV, V3 iadd V3, arraySizeOffset, V3 aadd V1, V3, V4 [b,c,s,i,l,f,d,a]ld (V4), V5	SRC SRC EXTRA EXTRA DEST
	ishl CONST, [b,c,s,i,l,f,d,a]shiftV, V3 iadd V3, arraySizeOffset, V3 aadd V1, V3, V4 [b,c,s,i,l,f,d,a]ld (V4), V5	CONST SRC EXTRA EXTRA DEST
	aadd V1, V2, V3 [b,c,s,i,l,f,d,a]ld (V3), V4	SRC SRC EXTRA DEST
	[b,c,s,i,l,f,d,a]ld (V1), V2	SRC DEST
	nop	SRC
[b,c,s,i,l,f,d,a]astore	ishl V2, [b,c,s,i,l,f,d,a]shiftV, V4 iadd V4, arraySizeOffset, V4 aadd V1, V4, V5 [b,c,s,i,l,f,d,a]st V3, (V5)	SRC SRC SRC EXTRA EXTRA
	ishl CONST, [b,c,s,i,l,f,d,a]shiftV, V3 iadd V3, arraySizeOffset, V3 aadd V1, V3, V4 [b,c,s,i,l,f,d,a]st CONST, (V4)	CONST SRC SRC EXTRA EXTRA
	aadd V1, V2, V4 [b,c,s,i,l,f,d,a]st V3, (V4)	SRC SRC SRC EXTRA
	aadd V1, V2, V3 [b,c,s,i,l,f,d,a]st CONST, (V3)	CONST SRC SRC EXTRA
	[b,c,s,i,l,f,d,a]st V1, (V2)	SRC SRC
	[b,c,s,i,l,f,d,a]st CONST, (V1)	CONST SRC
	nop	SRC

Figure 9: Java IR and VRA annotation formats for array element accesses

Bytecode	Java IR	VRA Annotation Formats
arraylength	aadd V1, arraySizeOffset, V2 ild (V2), V3	SRC EXTRA DEST
	ild (V1), V2	SRC DEST
	nop	SRC
newarray	amovi addressOfNewArray, V1 acall V1, arrayType, V2, V3	SRC EXTRA DEST
	amovi addressOfNewArray, V1 acall V1, arrayType, CONST, V2	CONST EXTRA DEST
	acall V1, arrayType, V2, V3	SRC SRC DEST
	acall V1, arrayType, CONST, V2	CONST SRC DEST
anewarray	amovi addressOfAnewArray, V1 acall V1, classType, V2, V3	SRC EXTRA DEST
	amovi addressOfAnewArray, V1 acall V1, classType, CONST, V2	CONST EXTRA DEST
	acall V1, classType, V2, V3	SRC SRC DEST
	acall V1, classType, CONST, V2	CONST SRC DEST
multianewarray	amovi addressOfMultiAnewArray, V1 acall V1, classType, V2, V3... Vn	[SRC/CONST] EXTRA DEST
	acall V1, classType, V2, V3... Vn	[SRC/CONST] SRC DEST

Figure 10: Java IR and VRA annotation formats for creating and manipulating array objects

7.5 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 arguments 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 re-occur refering to the same object and the same method, annotation bytes can suggest the omission of the method address computation. Note that we have made assumptions about dynamic dispatching conventions. However, portability of the annotations has not been affected. 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 value. Figure 12 shows the correspondence between these bytecodes, Java IR operations and VRA annotation formats.

Bytecode	Java IR	VRA Annotation Formats
`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`	SRC SRC
	`icmp_<eq,ne,lt,le,ge,gt> CONST, V2, V3` `br V2 trueLabel falseLabel`	CONST SRC
	`icmp_<eq,ne,lt,le,ge,gt> V1, CONST, V2` `br V2 trueLabel falseLabel`	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`	SRC SRC DEST

	`[l,f,d]cmp V1, CONST, V2`	SRC CONST DEST
	`[l,f,d]cmp CONST, V1, V2`	CONST SRC DEST
`goto` `gotow`	`goto label`	none
`return`	`return`	none

`jsr`
`jsr`
`jsr_w`
	`goto label`	none

`ret`	`goto label`	none
`tableswitch` `lookswitch`	`icmp_<eq> V1, CONST, V2` `br V2 trueLabel falseLabel`	SRC
`athrow`	`amovi addressOfThrow, V2` `call V2, V1`	SRC EXTRA
	`call V1, V2`	SRC SRC
`monitorenter` `monitorexit`	`amovi addressOfMonitorenter, V2` `call V2, V1`	SRC EXTRA
	`call V2, V1`	SRC SRC

Figure 11: Java IR and VRA annotation formats for control transfer, exception handling and synchronization operations

7.6 Operand Stack Management Instructions

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

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

Figure 12: Java IR and VRA annotation formats for method invocation and return operations

7.7 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. These operations are listed in Figure 11. We are modifying our current implementation of AJBC and AJIT to support exception handling as summarized above.

7.8 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 an argument. Our current implementation does not handle synchronization at statement nor method levels.

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] M. G. Burke, J. Choi, S. Fink, D. Grove, and M. Hind. The Jalapeno Dynamic Optimizing Compiler for Java. *In Proceedings of the ACM Java Grande Conference*, pages 129–141, June 1999.

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

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

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

[8] 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.

[9] P. Sweeney F. Tip, C. Laffra. Practical Experience with an Application Extractor for Java. *ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, November 1999.

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

[11] P. Lee G. Necula. The Design and Implementation of a Certifying Compiler. *ACM Conference on Programming Language Design and Implementation*, June 1998.

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

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

[17] 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.

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

[19] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, 1997.

[20] G. Necula. Proof-Carrying Code. *ACM Symposium on Principles of Programming Languages*, January 1997.

[21] 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.

[22] 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.

[23] Effective Edge Technologies. guavac open source.
Sources available at ftp://ftp.unicamp.br/pub/languages/java/guavac.

[24] O. Traub, G. Holloway, and M. D. Smith. Quality and Speed in Linear-scan Register Allocation. *Proceedings of ACM Programming Languages Design and Implementation*, pages 142–151, 1998.

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

[26] T. Wilkinson. Kaffe Open Source Java Virtual Machine. See http://www.transvirtual.com.

[27] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. *International Conference on Parallel Architectures and Compilation Techniques*, pages 128–138, October 1999.