CONCURRENCY: PRACTICE AND EXPERIENCE
*Concurrency: Pract. Exper.* 2000; **12**:389–406

# Annotating Java class files with virtual registers for performance

Joel Jones*,† and Samuel Kamin

*Department of Computer Science, University of Illinois at Urbana-Champaign, 3270 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801, U.S.A.*

**SUMMARY**

**The Java `.class` file is a compact encoding of programs for a stack-based virtual machine. It is intended for use in a networked environment, which requires machine independence and minimized consumption of network bandwidth. However, as in all interpreted virtual machines, performance does not match that of code generated for the target machine. We propose verifiable, machine-independent annotations to the Java class file to bring the quality of the code generated by a 'just-in-time' compiler closer to that of an optimizing compiler without a significant increase in code generation time. This division of labor has expensive machine-independent analysis performed off-line and inexpensive machine-dependent code-generation performed on the client. We call this phenomenon 'super-linear analysis and linear exploitation.' These annotations were designed mindful of the concurrency features of the Java language. In this paper we report results from our machine-independent, prioritized register assignment. We also discuss other possible annotations. Copyright © 2000 John Wiley & Sons, Ltd.**

KEY WORDS: runtime code generation; register allocation; load/store elimination; pragma generation

## 1. INTRODUCTION

The Java `.class` file is a compact encoding of programs for a stack-based virtual machine. It is intended for use in a networked environment, which requires machine independence and minimized consumption of network bandwidth. However, as in all interpreted virtual machines, performance does not match that of code generated for the target machine. To ameliorate this problem, many implementations of the Java Virtual Machine (JVM) use 'just-in-time' (JIT) compilers, in which Java bytecodes are translated into machine code.

We propose machine independent annotations to the Java class file [1] to bring the quality of the code generated by a 'just-in-time' compiler closer to that of an optimizing compiler without a significant

*Correspondence to: Joel Jones, Department of Computer Science, University of Illinois at Urbana-Champaign, 3270 Digital Computer Laboratory, 1304 West Springfield Avenue, Urbana, IL 61801, U.S.A.
†E-mail: jjones@uiuc.edu

*Received 1 August 1999*
*Revised 15 November 1999*

increase in code generation time. The annotations proposed are a specification of a prioritized register assignment (virtual registers, VRs), load-store elimination (remove if physical, RIPs), and register spilling (swaps). Our notion is to provide a division of labor between the class file annotator on the server and the code generator on the client.

This environment differs from both conventional batch compilation and compilation for interactive languages [2,3]. In conventional compilation, to a first order of approximation, compilation costs are ignored, and the target machine is known; no holds are barred in terms of aggressive analyses and machine-dependent trickery. In the interactive environment, the target machine is known, but the cost of compilation is constrained by the need to maintain interactive responsiveness. By contrast, in the network client/server model, compilation to bytecode is done off-line and is, to a first-order of approximation, unconstrained. However, it must generate portable code and therefore cannot perform machine dependent optimizations. Compilation of bytecode to target machine code is performed on the client and may therefore be very machine-specific. However, since this compilation time is added to the overall response time seen by the user, it must be minimized. As a result, the quality of the code generated by 'just-in-time' compilers does not match that of an optimizing compiler for the same machine.

The key idea is that the machine-independent analysis phase performed by the Java compiler may be expensive, but the results of that analysis may be expressed compactly and used inexpensively. We call this phenomenon 'super-linear analysis and linear exploitation.'

One guiding feature of our annotations, and indeed of any annotations of the Java class file, is that the annotations can be safely ignored by those implementations of the Java VM that do not recognize them. This precludes any transformation on the bytecode that results in a `.class` file which is unverifiable or incorrect when considered without the annotations.

The following sections describe in more detail the environment we are trying to produce code for followed by a concrete example showing Java source, Java bytecodes, and SPARC machine code. Then, the properties of the machine code generated by our 'just-in-time' compiler, the semantics of our register assignment annotation, and annotation generation are discussed. Following this are more examples of Java source code, the corresponding Java byte code, and the machine code generated with our register assignment annotation, including timing information. We then cover related work in solving this sort of problem and then conclude with a summary of our contributions.

## 2. THE OVERALL ENVIRONMENT

In Figure 1 we see the environment in which our annotated `.class` file must operate. A client makes a request for some Java code, packaged either as a `.class` file, or as a `.jar` file containing a `.class` file. The server sends the file to the client. This annotated code is used by our modified JVM to produce machine code, as in any just-in-time (JIT) compilation system. In the figure, $t_1$ represents the time from when the client requests a Java `.class` file to the time that the client first sees results, while $t_2$ represents the time from the `.class` file request to when the computation completes. We are interested in minimizing both $t_1$ and $t_2$. If we were interested in minimizing just $t_1$, then we could use an interpreter over the bytecodes and see our first results quickly. If we were interested in minimizing just $t_2$, then for a program which was compute intensive, we could tolerate a large amount of traditional optimizing compiler analysis being done by the JVM.

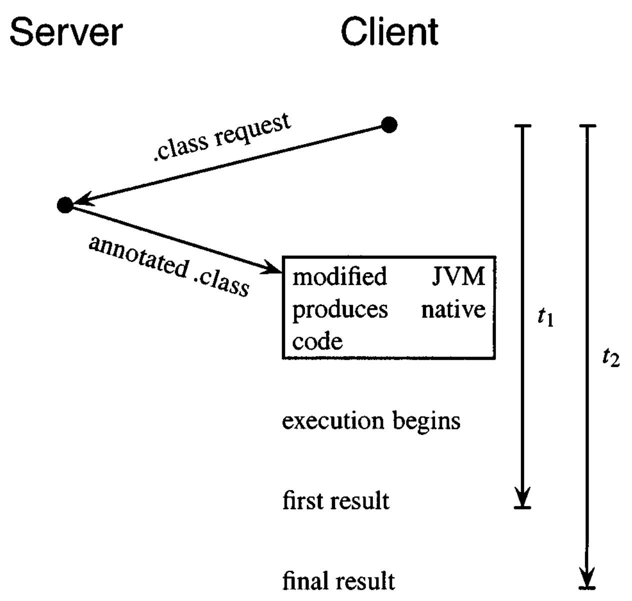


Figure 1. Java virtual machine execution.

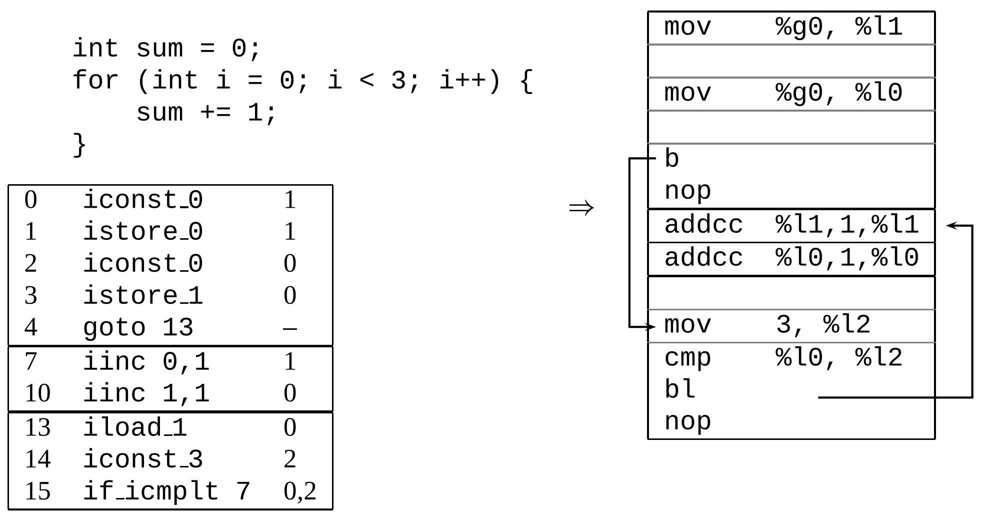
Obviously, the client-side code generator could perform the same control and data flow analysis that the server-side annotator does. But our approach is to do as much work up front as possible. There are two advantages to having the analysis done by the server. First, time is saved by having the analysis done once, not every time a class file is compiled to machine code. More importantly, many optimizations can be divided into a super-linear analysis phase and a linear exploitation phase. By performing the expensive analysis phase and recording the results as annotations, we can then very quickly produce high-quality code in the code generator.

Note that adding annotations does not preclude the VM from doing additional optimizing transformations on the bytecodes. It merely changes the intermediate representation from bytecodes for a stack-based virtual machine to bytecodes for an infinite register virtual machine.

The first part of our system, the annotator, takes as input `.class` files and produces as output annotated `.class` files. These annotations are produced by using the Java bytecodes as an intermediate representation and applying modifications of traditional optimizing compiler algorithms to them. We then add the results of these algorithms as annotations to the output `.class` file. The size of these annotations must not be overly large, as this would increase the amount of time needed to transfer the annotated `.class` file across a network.

## 3. SIMPLE EXAMPLE

Before going into detail about how annotations are generated and the exact nature of the mechanisms in the code generator to exploit them, let us examine a very simple example first.



```
int sum = 0;
for (int i = 0; i < 3; i++) {
    sum += 1;
}
```

| | | |
|---|---|---|
| 0 | iconst_0 | 1 |
| 1 | istore_0 | 1 |
| 2 | iconst_0 | 0 |
| 3 | istore_1 | 0 |
| 4 | goto 13 | – |
| 7 | iinc 0,1 | 1 |
| 10 | iinc 1,1 | 0 |
| 13 | iload_1 | 0 |
| 14 | iconst_3 | 2 |
| 15 | if_icmplt 7 | 0,2 |

⇒

```
mov    %g0, %l1

mov    %g0, %l0

b
nop
addcc  %l1,1,%l1
addcc  %l0,1,%l0

mov    3, %l2
cmp    %l0, %l2
bl
nop
```

Figure 2. For loop with accumulator.

In Figure 2, we have an example of the Java source for a ‘for’ loop. This loop has three values, `sum`, `i`, and the constant `3`. Below it, we have the Java bytecodes corresponding to the source. The Java bytecode is formatted with the first column the bytecode location, the second column the Java bytecode and any arguments, and the third column the ‘VR’ annotation. The rightmost column of the Java bytecodes block is a register assignment for the bytecode to its left. We call this register assignment ‘virtual registers’ or VRs. On the right-hand side, we have the SPARC machine code generated from the bytecodes. For the SPARC machine code, instructions for a single Java bytecode are separated by single gray horizontal lines. For both Java bytecodes and SPARC machine code, basic blocks are separated by thick horizontal lines.

The assignment for `sum` is $\mathsf{vr}_0$; `i`, $\mathsf{vr}_2$; and `3`, $\mathsf{vr}_1$. Also, the assignment of physical registers for virtual registers starts at `%l0`. The SPARC architecture divides the general-purpose register set into four parts, the global registers, labeled `%g0`–`%g7`, the input registers, labeled `%i0`–`%i7`, the output registers, labeled `%o0`–`%o7`, and the local registers, labeled `%l0`–`%l7`. The first piece of machine code starts with a SPARC machine idiom of using global register `%g0`, which is always zero, to initialize a register with an integer value. Note that on the SPARC, the destination of an instruction is the rightmost argument. Next, we see an empty slot in our machine code diagram, indicating that no machine code was generated for the `istore_0` bytecode. The next interesting set of instructions occurs in the code for the `iinc` bytecodes. The numbers after the `iinc` bytecodes indicate which local slot is being incremented, and what it is being incremented by, respectively. This is part of the normal encoding for the `iinc` bytecodes. Each `iinc` bytecode is translated into a single SPARC add instruction, since the values referenced by both bytecodes are being stored in physical registers.

In this example, each VR corresponds directly to a physical register. The advantage of the VR annotation is that it provides a register assignment at no cost. Of course if there were always a sufficient

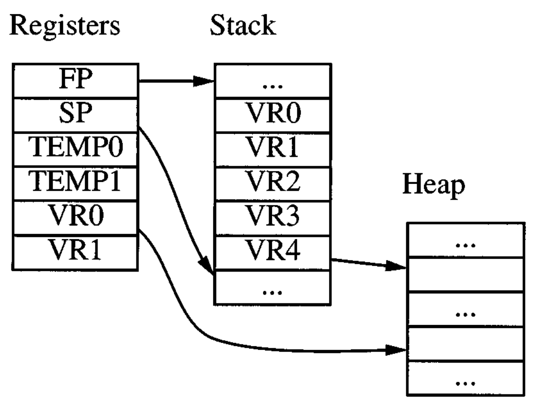


Figure 3. Run-time environment. The stack location of $vr_0$ is reserved for spilling register $vr_0$; it may or may not contain the same value at other times.

number of physical registers, the register assignment problem would be completely conventional. In fact, the number of VRs is usually greater than the number of physical registers. VR annotations also provide a fast way to determine where register spill code is needed.

## 4. THE CODE GENERATION ENVIRONMENT

To make things more concrete, we will use the SPARC as our example machine in the following sections. With the exception of its register windows, the SPARC is like most modern RISC architectures. Furthermore, since all architectures perform better when registers are used as much as possible, most of the principles guiding the SPARC implementation hold everywhere.[‡]

Figure 3 gives the overall structure of the machine environment for which we are generating code. The bounds of the current activation record (stored on the stack) are maintained by two registers, the stack pointer (SP) and the frame pointer (FP). Temporaries used by the code generator are also set aside in the registers TEMP0 and TEMP1. Values kept primarily in registers are shown in the table labeled 'Registers.' Values which primarily reside in memory are shown only in the table labeled 'Stack.' Objects and arrays are kept in the heap and references to them are kept in registers or on the stack. The following sections give details related to the environment sketched in Figure 3.

### 4.1. Simple properties

Our central annotation is the virtual register (VR) annotation. The basic notion is that there is a set of virtual register numbers for each Java bytecode which correspond to the operands of the bytecode. The

[‡]Including the popular Intel x86 and embedded versions of the Motorola 68K.

Table I. VR location table; MAXREGS = 2.

| Virtual register | Physical register | Memory |
|---|---|---|
| 0 | %l0 | [%fp - 12] |
| 1 | %l1 | [%fp - 16] |
| 2 | – | [%fp - 0] |
| 3 | – | [%fp - 4] |
| 4 | – | [%fp - 8] |

code generator uses the VR annotation to generate a register assignment for the machine instructions. These VRs are arranged in *priority* order, meaning that the lower the VR is, the more likely it is to be assigned to a physical register in the code generator. We will deal with the details of the VR annotation in more detail in Section 5.1.

Below are some simple properties that guide our code generation process. We use the term 'physical register(s)' to refer to the machine registers and 'physical location' to refer to some location on the machine, either in a physical register or in memory. The memory that is used for virtual registers is contained in a run-time stack, which also contains the stack and frame pointer. We will also use the term 'primarily stored' to indicate the physical location where a VR's value can be found most of the time.

- All virtual registers have at least one physical location. Virtual registers which primarily reside in physical registers have two physical locations, their physical register and their stack location. Other virtual registers will reside only in one physical location, on the stack. We will see the need for all values having a place on the stack when we deal with spilling.
- Constants are either folded into the machine instructions or are statically allocated alongside the code for each method. The memory for the code and constants is allocated in the heap, but is otherwise treated statically by our code generation system.
- The operand stack is not mimicked. All VRs not assigned to physical registers are loaded into physical registers if necessary, then written out, using physical registers dedicated to such temporary use. A more detailed description of this process is given in the next section.
- A simple data structure, the VR location table, provides the mapping from VRs to physical locations. Conceptually, the code generator keeps two sets of mapping, one for virtual registers residing only in memory, and one for virtual registers additionally residing in a physical register. This table is mostly static, and is built before code generation begins. Imagine we have a machine where only two physical registers remain after setting aside registers for use as temporaries. Further, assume that the method we are compiling has five VRs, numbered 0–4, specified in all its bytecodes. An example of this data structure can be seen in Table I. This indicates that virtual registers 0 and 1 will be primarily stored in physical registers `%l0` and `%l1` respectively and that virtual registers 2, 3, and 4 will be primarily stored in memory at 0, 4, and 8 off the frame pointer.

### 4.2. Non-register resident values

To manage situations where the number of physical registers is insufficient to contain all the virtual registers, a scheme to place the excess virtual registers into memory has been devised. There are several things to note.

- Several physical registers are reserved for temporary use. The lifetime of the values placed into these registers typically does not extend beyond the span of the machine code generated for one Java bytecode.
- For a given virtual register, its location on the stack is at a fixed offset from the frame pointer.
- On machines which allow reading and writing of a single physical register in an instruction (e.g. `add %l0,%l1,%l0`), only two temporary registers are needed (actually, two temporary registers for integral values and another two for floating-point values.)

### 4.3. Code generation

The general scheme for generating machine code is as follows, doing the steps below for each bytecode:

1. For all input VRs to the bytecode which are not assigned a physical register, according to the current state of the VR location table, generate a load from their stack location into a temporary physical register.
2. If the output VR from the bytecode is not assigned a physical register, set aside a temporary physical register.
3. Generate the appropriate machine instruction using temporary and/or permanent registers.
4. If the output VR from the bytecode is not assigned a permanent physical register, generate a store from the temporary physical register to its location on the stack.

The VR location table is determined by the code generator by using information gathered during the bytecode and VR annotation verification process. The verification step is used by the JVM to ensure that the bytecodes of a downloaded program do not corrupt the virtual machine. We have modified this step to additionally verify the VR annotation associated with each method. As a side effect of this process, the type of each virtual register is determined. For machines with a split general purpose register set and floating point register set, object references and integral types are assigned to the general purpose register set, and floating point types are assigned to the floating point register set. There may be some VRs which will not be assigned to physical registers. Any virtual registers not assigned to physical registers are only assigned locations on the stack. This process is also responsible for generating procedure entry prologues.

The verification process allows us to make an important optimization. Since the verification process ensures that every use of a local JVM frame slot (or equivalently a local variable) is preceded along all paths by a definition of that slot, we can logically eliminate most load/store bytecodes that reference locals. As an example of the machine code generated using this scheme, consider this bytecode and annotation: '`iadd` 2,3→4'. This signifies that the `iadd` bytecode is annotated with three virtual registers, 2, 3, and 4. The `iadd` bytecode, in the semantics of the stack-oriented JVM, removes the top two integer operands off the bytecode operand stack, adds them, and pushes the resulting integer result onto the stack. Our VR annotation has the semantics: take the values contained in virtual registers 2

Table II. Mapping from VRs to physical registers; MAXREGS = 4.

| Virtual register | Physical register | Memory |
|---|---|---|
| 0 | %l0 | [%fp - 4] |
| 1 | %l1 | [%fp - 8] |
| 2 | %l2 | [%fp - 12] |
| 3 | %l3 | [%fp - 16] |
| 4 | – | [%fp - 0] |

and 3, add them, and place the result in virtual register 4. If we have a SPARC-like machine with only two allocatable physical registers for holding integers, then we generate the following, assuming the mapping from Table I:

```
! virtual registers > vr1 live in the activation record
    ld [%fp - 0], %g1  ! copy vr2 -> g1
    ld [%fp - 4], %g2  ! copy vr3 -> g2
    add %g1,%g2,%g1    ! compute result
    st %g1, [%fp - 8]  ! copy result to vr4
```

(We are using `%g1` and `%g2` as temporary registers.) If we can allocate four physical registers to virtual registers, then the mapping would become that shown in Table II. We would generate:

```
! virtual registers > vr3 live in the activation record
    add %l2,%l3,%g1      ! compute result
    st %g1, [%fp + 36]   ! copy result to vr4
```

## 5. ANNOTATIONS

We have four sets of annotations: register assignment (VRs), redundant load/store elimination (‘remove if physical,’ RIPs), register spills (swaps), and copies. Below is a description of the VR annotation. Currently, the VR annotation is implemented in both the annotator and in the code-generator. The RIP and swap annotations, which have not been implemented yet, are described in Section 7. We have a ‘copy’ annotation which indicates where a load from one local followed by a store to another local corresponds to a copy assignment from one local to another. This annotation deals with the situation where the value of a local variable is placed on the stack and stored into another local. As noted in the previous section, we do not generate code for most bytecodes which do loads/stores of local variables. We do, however, if a particular load/store bytecode is annotated with a copy annotation. The generation and use of the copy annotation is very straightforward. The copy annotation is generated by simply noting where such a pattern exists. It is used in the code generator to generate an appropriate register-register or register-memory move instruction. We will not deal with the copy annotation any further in this presentation.

## 5.1. VRs

The VR annotation is an assignment for each Java bytecode of a set of virtual register numbers which correspond to the operands of the bytecode. The number of virtual registers per bytecode varies. For example, as we have seen above, a binary arithmetic operator will have three virtual registers—two for the input and one for the result. A method call bytecode will have a variable number of virtual registers—one for the object, one for each argument to the method, and one for the return value.

The salient characteristics of this annotation are that VRs are assigned their priority based upon their importance and that the number of distinct virtual registers is minimized. The priority is equivalent to the inverse of the virtual register number. Disjoint live ranges of the same type may be assigned to the same virtual register. This makes each VR *monotyped*, i.e. a VR can only 'carry' one type of value throughout the entire method. This includes reference types, taking into account the least-upper bound along the inheritance and interface hierarchies induced by the data-flow algorithm of the bytecode verification procedure [1]. For example, if $\mathsf{vr}_0$ is used as a reference to an object of class A at one point in a program, it may not be used later as an integer or even as a reference to an object of class B, unless class A and class B have a type-compatible superclass.

This annotation uses an unsigned one byte quantity for each virtual register. The values 0–254 indicate a valid virtual register number and the value 255 indicates no virtual register assignment. More information on the VR usage for every bytecode can be found in [4].

## 5.2. Generating VRs

The process of generating our VR annotation is similar to that of other register allocators. Once we have discovered the values that are actually used by a method, we proceed using standard graph-coloring techniques, with an important distinction—we do not know the number of physical registers. Therefore, our algorithm for finding a register assignment consists of the standard algorithm modified to operate without this vital piece of information. We will begin by describing the Chaitin graph coloring register allocator, which forms the basis for our allocator [5]. We will then proceed by giving our modifications to the Chaitin algorithm.

Finding a register allocation can be viewed as a graph-coloring problem. An interference graph is constructed where each node represents a value (live range) from the program and the edges are between values which are simultaneously 'live.' The goal is to find a $k$-coloring of the interference graph, where $k$ is the number of physical registers. We can view this process as involving an oracle that tells us whether or not a $k$-coloring is possible (using a particular heuristic). If it is, then the coloring of the interference graph indicates which values should be assigned to which physical registers. If a $k$-coloring is not possible, then a value is chosen to be spilled (i.e. to reside in memory), effectively removing its node from the graph, and another attempt to $k$-color the graph is made. This process continues until the graph is colored and thereby a register assignment obtained.

In our allocator, we also view the register assignment problem as graph coloring of the interference graph. However, we have to change our abstract notion of the problem to deal with the fact that we do not know the value of $k$, the number of physical registers. Therefore, instead of attempting to find a $k$-coloring, we are actually trying to find the minimum $k$, such that a $k$-coloring of the graph exists. This gives us a coloring that tends to reduce the number of registers; although this number may well be larger than the actual number of registers of a particular machine, it is still a logical place to start.

```
int test(boolean mode, int U[], int V[]) {
    int SUM;
    if (mode) {
        A1 = U[0]; A2 = U[1];
    } else {
        B1 = V[0]; B2 = V[1];
    }
    if (mode) {
        SUM = A1 + A2;
    } else {
        SUM = B1 + B2;
    }
    return SUM;
}
```

Figure 4. Example from Chaitin (size = 2).

Another aspect of our environment that we must deal with is the need for verification. We do this by introducing edges into the interference graph between nodes that have differing types. This ensures that every VR is *monotyped*, as discussed above. A final concern that arises in our environment is that we must prioritize our colors—in a very abstract sense, turning our colors into a gray-scale. In the normal Chaitin allocator, which physical register holds which live ranges does not matter. In our prioritized VR scheme, on the other hand, we need to decide beforehand which registers should be spilled first, and it is simplest to use the register number to indicate its priority. Thus there may be no difference between using register 1 and using register 10 (if the machine has ten registers), but there is a difference if the machine has fewer registers and we are forced to spill one; register 10 has the lower priority, so it will be the one spilled. Therefore, we use the so-called 'Haifa heuristics' [6] to order our colors so that the colors that hold the most important live ranges are numbered lower. These heuristics take into account the number of uses of the live range, and how many other live ranges are simultaneously live with this live range. Once this prioritization has been calculated, the VR annotation is added to the `.class` file.

## 6. EXAMPLES

In the following pages are two examples of code generation for the SPARC using the annotation scheme described above. The annotations were produced by our compiler, which accepts `.class` files as input and produces annotated `.class` files as output. The SPARC machine code shown was produced by our code generator embedded within the kaffe JVM [7]. The general format of these examples follows that first shown in Figure 2.

In Figure 7 we have the code for a complete method, which therefore includes machine code for setting up the activation record on the stack—the `save` instruction in the SPARC machine code on the right. We will return to this example again in Section 7.2

```
Java bytecode
     ...
42 istore_3
43 goto 54
     ...
53 istore_3
54 iload_3
55 ireturn
```

```
Without VRs                          With VRs
                                        ...
                                     88: addcc %l0, %o4, %l0
     ...
5dc: st %l5, [ %fp + -92 ]           8c: b b4
5e0: b 614                           90: nop
5e4: nop                                ...
                                     b0: addcc %l0, %o4, %l0
     ...
610: st %i5, [ %fp + -92 ]
614: ld [ %fp + -92 ], %l7
618: mov %l7, %i4
61c: mov %i4, %i0
620: ret                             b4: ret
624: restore                         b8: restore %g0, %l0, %o0
```

Figure 5. Bytecodes and SPARC machine code generated from Chaitin (size = 2).

In Figure 4 we have the equivalent Java source code for the example from the Chaitin *et al.* 1981 paper [5]. What makes this example interesting is that the separation into two if-then-else statements makes local register allocation ineffective. To wit, local register allocation forces the store of `sum` at the end of both branches of the second if statement and the subsequent load of `sum` at the beginning of the 'return' basic block.

In Figure 5, we have the bytecodes and the SPARC machine code generated by a code generator for a portion of the source code of Figure 4. On the bottom left, we have the code generated by the kaffe code generator, which does not use our VR annotation. On the bottom right, we have the code generated by our code generator, using our VR annotations. The code shown is that for the very end of the then-branch of the second if statement and for the return statement. Note that the code for line 614 of the 'Without VRs' SPARC code is preceded along both of its predecessors, lines 5e0 and 610, with stores. The stores in lines 5dc and 610 and the following load in line 614 are necessary because

Table III. Results of VR annotation for Chaitin (SPARCstation 20).

| Problem size | | `.class` file size | Code gen time | Execution time | Verification time |
|---|---|---|---|---|---|
| 2 | w/o VRs | 1252 | 1 ms | 3 ms | <1 ms |
| | w VRs | 1645 | 2 ms | 3 ms | <1 ms |
| 10 | w/o VRs | 1800 | 6 ms | 6 ms | <1 ms |
| | w VRs | 2354 | 5 ms | 5 ms | <1 ms |

Table IV. Results for quicksort (SPARCstation 20).

| | `.class` file size | Code gen time | Execution + compilation time | Verification time |
|---|---|---|---|---|
| w/o VRs | 619 | 5 ms | 132 405 ms | <1 ms |
| w VRs | 855 | 5 ms | 124 856 ms | <1 ms |

the kaffe code generator is only doing a local register allocation. In the ‘With VRs’ SPARC code, the predecessors to the basic block containing the return do not contain unnecessary stores, and the return block itself does not contain unnecessary loads.

In Table III we see performance numbers based upon the example in Chaitin *et al.*’s 1981 work [5]. The lines for size $= 2$ correspond exactly to the source code in Figure 4. The lines for size $= 10$ correspond to the example as given in Chaitin *et al.* [5]. We see that for larger problem sizes, i.e. 10 vs. 2, that the VR annotation provides a speedup, while not increasing code generation time, or the part of code generation time spent on verification. These numbers were obtained on a SPARCstation 20. The lines marked ‘w/o’ are for code generated by the standard kaffe code generator, which uses a local register allocator, run at code generation time. The lines marked ‘w’ are for code generated by our code generator, using the VR annotations. Similarly, we see in Table IV similar results for an integer quicksort routine.

Overall, we see an 8% speedup, without an increase in code generation or verification time. We calculate the speedup using the normalized geometric mean equation from [8]:

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

Since the total execution time includes all methods, not just those annotated, this represents a very conservative estimate of the potential speedups.

```
AClass anObj;
anObj.aField = 3;
for (i = 0; i < a.length; i++) {
    a[i] = a[i] + anObj.aField;
}
```

Figure 6. Example of utility of RIP annotation.

## 7. OTHER ANNOTATIONS

As noted earlier, we have defined two annotations other than the VR annotation. The first of these, the 'RIP' annotation, is used for redundant load-store elimination. The second of these, the 'swap' annotation, is used to indicate where register spills should be performed if needed.

### 7.1. Semantics of 'RIP' annotation

The VR annotation removes most of the redundant loads and stores that would result from a naive JIT implementation for Java bytecodes. However, VRs do not address redundant load/store elimination for heap resident values. In situations dealing with class and instance variables, machine code for multiple loads (i.e. `getfield` and `getstatic` bytecodes) not reached by another definition after the first load, can be eliminated by marking all subsequent loads as 'remove if physical' if the virtual register is being stored in a physical register. A similar situation holds for stores. The analysis must take note of possible changes to an object that may take place via function calls. A conservative approach is to define any live range as ending when it reaches a function call. An example will help in clarification. Suppose we have the Java source code in Figure 6. The resulting annotated bytecode would have a `putfield` bytecode, for the assignment to `anObj.aField`. As long as we arrange for the VR used by the `getfield` bytecode for `anObj.aField` inside the loop to be the same as the VR used by the `putfield` bytecode, we can mark the `getfield` bytecode as a RIP.

Due to the 'precise exception' semantics of Java, it is anticipated that a few store instructions will not be marked 'remove if physical' using the above guidelines. For example, suppose we have the following Java code as the entire body of a method:

```
int b = 0;
obj.a = 0;
try { obj.a = 3; b = 4; obj.a = 5; }
finally { obj.f(b); }
```

It might seem that the first `putfield` bytecode in the `try` block for `obj.a` could be marked 'remove if physical'. However, assume that some asynchronous exception is thrown before the assignment of `b`, in the `try` clause. Since `obj.f()` may contain a `getfield` bytecode, the `putfield` may not be marked 'remove if physical'.

It is this sort of issue that makes optimizing for Java and C++, languages with exceptions, different from optimizing for languages like C and FORTRAN, which do not. There has been little published work in the area of optimizing for languages with exceptions. One important work, however, is

Hennessy's 1981 paper [9]. Other important work in the area of implementing exceptions has been done by Tiemann [10] for C++, by Chase [11,12] for Ada, Modula2+, C++, Module-3 and Eiffel, and by Goodenough [13] for general exception mechanisms.

Threading also must be examined carefully when generating 'remove if physical' annotations. Java has a shared-memory model for the sharing of information between threads of execution. Access to this shared-memory area may or may not be synchronized, with synchronization obtained either through explicit `monitor_enter` and `monitor_exit` bytecodes, or through methods marked as synchronized.

From a safety viewpoint, RIPs do not pose a threat. It may appear that the elimination of a store can lead to a violation of the requirement for definite assignment before use, which is normally guaranteed by the bytecode verification process. Since RIPs only apply to loads and stores of heap allocated values, the requirement is met by the class, array, and object creation semantics. When a class or instance is created, any variables associated with it are assigned an appropriate default value.[§] For numeric types, this default value is zero. More importantly, for reference types, the default value is `null`. Therefore, so long as the bytecode and VR are type-compatible, 'RIPs' cannot be used to grab a rogue pointer and break the sanity of the JVM.

### 7.2. Semantics of 'swap' annotation

When every variable is used with about the same frequency throughout a program, deciding which are most important to keep in physical registers (or equivalently in our case, to assign to lower numbered virtual registers) does not matter much. Similarly, if there are enough physical registers, then the priority does not matter. The reason is that, no matter what the assignment, an equal number of operations will result in accesses to memory. The point is that the non-uniform use of a variable throughout a method opens up opportunities for that variable to be spilled and allow a more important value to have a chance at a physical register.

Using the swap annotation to guide changes in physical register assignment is crucial in achieving our goal of having machine-independent annotations. There is a great disparity in the number of registers available in popular microprocessors. Those in the RISC family, such as the SPARC, have a large number of registers, typically 32 or greater. In the CISC family, there is a great deal of variance. The embedded version of the Motorola 68000 has only 16 registers, and the most popular non-embedded processor, the Intel x86 family, has even fewer. Given this disparity in the number of registers, swap annotations provide a means for improving the quality of the register assignment by allowing spill code to be generated more optimally, but in a machine-independent fashion.

To do this, we mark regions of the program with 'swaps' between two virtual registers. This indicates that until otherwise indicated, the roles of the two registers are swapped. This is accomplished in the code generator by changing the VR location table. In a traditional global register allocator, if the number of registers is insufficient to contain all live ranges, then code is inserted to 'spill' values from registers into memory and load values from memory into registers. The swap annotation has

[§]Assuming that an initializer is called after object creation [14].

```
int sum1 = 0, sum2 = 0;
for (int i = 0; i < 3; i++) {
    sum1 += i;
}
System.out.println(sum1);
for (int i = 1; i < 4; i++) {
    sum2 += 1;
}
return sum1;
```

| | | |
|---|---|---|
| 0 | `iconst_0` | 0 |
| 1 | `istore_0` | 0 |
| 2 | `iconst_0` | 3 |
| 3 | `istore_1` | 3 |
| 4 | `iconst_0` | 1 |
| 5 | `istore_2` | 1 |
| 6 | `goto 16` | |
| 9 | `iload_0` | 0 |
| 10 | `iload_2` | 1 |
| 11 | `iadd` | 0,1 → 0 |
| 12 | `istore_0` | 0 |
| 13 | `iinc` | 1 |
| 16 | `iload_2` | 1 |
| 17 | `iconst_3` | 2 |
| 18 | `if_icmplt 9` | 1,2 |
| 21 | `getstatic` | 4 |
| 24 | `iload_0` | 0 |
| 25 | `invoke_virtual` | 4,0 |
| 28 | `iconst_1` | 1 |
| 29 | `istore_3` | 1 |
| 30 | `goto 39` | |
| 33 | `iinc` | 3,1 |
| 36 | `iinc` | 1,1 |
| 39 | `iload_3` | 1 |
| 40 | `iconst_4` | 2 |
| 41 | `if_icmplt 33` | 1,2 |
| 44 | `iload_0` | 0 |
| 45 | `ireturn` | 0 |

⇒

| | |
|---|---|
| *`save`* | *`%sp, -200, %sp`* |
| `mov` | `%g0, %l0` |
| | |
| `mov` | `%g0, %l3` |
| | |
| `mov` | `%g0, %l1` |
| | |
| `b` | `xxx` |
| `nop` | |
| | |
| | |
| `add` | `%l0, %l1, %l0` |
| | |
| `addcc` | `%l1,1,%l1` |
| | |
| `mov` | `3, %l2` |
| `cmp` | `%l1, %l2` |
| `bl` | `0x10aea8` |
| `nop` | |
| `sethi` | `%hi(xxx), %g4` |
| `or` | `%g4, 0x324, %g4` |
| `ld` | `[%g4], %l4` |
| | |
| `mov` | `%l4, %o0` |
| `ld` | `[%o0], %g2` |
| `ld` | `[%g2 + 0x8c], %g2` |
| `mov` | `%l0, %o1` |
| `call` | `%g2` |
| `nop` | |
| `mov` | `1, %l1` |
| | |
| `b` | `xxx` |
| `nop` | |
| `addcc` | `%l1,1,%l1` |
| `addcc` | `%l3,1,%l3` |
| | |
| `mov` | `4, %l2` |
| `cmp` | `%l1, %l2` |
| `bl` | `xxx` |
| `nop` | |
| | |
| `ret` | |
| `restore` | `%g0, %l0, %o0` |

Figure 7. Utility of swaps.

the effect that spill code generation has in a traditional allocator. As covered in more detail below, the presence of a swap annotation may or may not result in spill code being generated by the code generator.

One can see a simple example of the utility of swaps in Figure 7. On the top left is the body of a Java function. On the bottom left is the corresponding annotated Java bytecode. The value corresponding to `sum1` is assigned to $\mathtt{vr}_0$, and the value for `sum2` is assigned to $\mathtt{vr}_3$. Although $\mathtt{vr}_3$ is not used in the first for loop and $\mathtt{vr}_0$ is not used in the second, we cannot assign either to the same virtual register since they are simultaneously live or interfere. If we were generating code for a machine with two allocatable registers, and used only our VR annotation, then $\mathtt{vr}_3$ would not be assigned to a physical register. Note that the machine code here is generated assuming the normal number of registers for the SPARC. However, we can improve the performance of this code by annotating the `goto` bytecode at PC 30 by saying that the relative priority of $\mathtt{vr}_0$ and $\mathtt{vr}_3$ are swapped when doing code generation. We annotate at this location since this is a natural loop header. This is done by modifying the VR location table discussed in Section 4.1.

The 'swap' annotation consists of information indicating the PC where the swap is located, and which virtual registers are having their priorities swapped. What the code generator does with the swap annotation is dependent upon the number of registers available to be allocated. The 'swap' annotation has the form $A \leftrightarrow B$, where $A$ and $B$ are the VRs that are to have their priorities swapped. In this annotation, we always have $A < B$, so that $B$ is the VR whose priority is being increased. The rule is simple: if $A$ is in a physical register and $B$ is not, then insert spill code and modify the VR location table accordingly; otherwise do nothing—that is, neither generate any machine code nor change the VR location table.

To devise an algorithm for determining where swaps should be inserted, we will use the intuition provided by the aggressive live range splitting algorithm of Briggs [15]. There, live ranges are aggressively split before coloring by inserting spill code based upon the single assignment (SSA) form of the method. Extraneous copies are eliminated before coloring is attempted. We will simplify the Briggs algorithm by simply generating the SSA form and then inserting the appropriate swap annotations to correspond to the location and values of the $\phi$-functions of the SSA form.

It may prove necessary to have two kinds of swaps—before and after. An after swap placed on a bytecode indicates that the swap should logically take place after the effect of the instruction has taken place. For conditional branches, this will have the additional semantics that the swap takes place when the branch is not taken. This is necessary to ensure that there is always a place to put the swap—which logically appears on a control-flow edge, not an instruction.

Swaps present no integrity problem, as long as the two values being swapped are type-compatible.

## 8. RELATED WORK

Many current implementations of the JVM use JIT technology. We discuss a few of these below.

The open source virtual machine which we use to implement our work is kaffe, from Transvirtual [7]. This VM can be deployed either as an all-interpreted system or an all-JIT system. Their JIT system does a very weak local register allocation, and no other optimizations.

Sun's 'Java HotSpot$^{TM}$ performance engine' [16] employs an optimizing compiler on code that has been determined to be performance critical. The optimizing compiler in their VM includes implementations of dead-code elimination, loop invariant hoisting, common subexpression elimination, constant propagation and a graph-coloring global register allocator.

Another JVM developed in a commercial environment is one at IBM Tokyo Research Laboratory [17]. It applies a similar set of optimizations as HotSpot does in its JIT. However, it does not use a global register allocator, but uses a region-based local one instead. They claim 'Since the JIT compiler requires fast compilation, expensive register allocation algorithms, such as graph coloring, cannot be used.' We prove this assertion is limited to non-annotation-aware JITs by doing the graph coloring when making the `.class` file, not when executing it.

Most closely related to our work is the AJIT system [18,19]. They also add annotations to the `.class` file, and then use an 'annotation-aware' JVM to generate machine code. They break up some of the more complicated Java bytecodes into suboperations (e.g. `iaload`, load an element from an array of integers) and produce annotations specific to the suboperations. By in effect transforming their bytecodes into a sort of microcode, it seems likely that the verification process is made more difficult. Furthermore, they do not have the swap annotation, which we argued is important for increasing the portability of any annotated code.

## 9. OPEN QUESTIONS

While we have shown the potential for a useful technique in the implementation of portable virtual machines, there are important questions still open.

1. Are the virtual register assignments worth the transmission cost versus doing global register allocation in the JVM?
2. How much does monotyping a virtual register cost in terms of lost opportunities to allocate the register for typical Java functions?
3. How effective are these techniques on machines with fewer and more restricted registers than the SPARC?

We intend to aggressively address these issues as our implementation matures. It is our intention to answer these questions in an environment where the benefits of portability and security of the JVM architecture are not compromised.

## 10. CONCLUSIONS

We have defined and implemented a register allocation annotation for Java `.class` files which is both machine-independent and safe. In addition, we have defined two other annotations which are machine-independent and safe, 'remove if physical' and swaps which should also prove to be performance enhancing.

**ACKNOWLEDGEMENTS**

We would like to thank Mike Lake for coining the phrase and clarifying the concept of 'superlinear analysis and linear exploitation' and for providing a version of Figure 1 suitable for LaTeX.

## REFERENCES

1. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (*The Java Series*). Addison-Wesley, 1997.
2. Johnson R, Graver JO, Zurawski LW. TS: An optimizing compiler for smalltalk. *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, November 1988, vol. 23, no. 11. ACM, 1988; 18–26.
3. Holzle U. Adaptive optimization for SELF: Reconciling high performance with exploratory programming. *Technical Report STAN//CS-TR-94-1520*, Department of Computer Science, Stanford University, August 1994.
4. Jones J. Annotating Java class files for performance. http://mtdoom.cs.uiuc.edu/AnnotationSemantics.ps [November 1997].
5. Chaitin GJ, Auslander MA, Chandra AK, Cocke J, Hopkins ME, Markstein PW. Register allocation via coloring. *Journal of Computer Languages* 1981; **6**:45–57.
6. Bernstein D, Goldin DQ, Golumbic MC, Krawczyk H, Mansour Y, Nahshon I, Pinter RY. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices* 1989; **24**(7):258–263. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*
7. Transvirtual Technologies. Kaffe OpenVM$^{TM}$.
8. Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach* (2nd edn). Morgan Kaufmann: San Mateo, CA, 1996.
9. Hennessy J. Program optimization and exception handling. *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Williamsburg, Virginia, 26–28 January 1981. ACM SIGACT-SIGPLAN, ACM Press, 1981; 200–206.
10. Tiemann MD. An exception handling implementation for C++. *USENIX C++ Conference*, Berkeley, CA, USA, 1990. USENIX Association, 1990; 215–232.
11. Chase D. Implementation of exception handling, Part I. *The Journal of C Language Translation* 1994; **5**(4):229–240.
12. Chase D. Implementation of exception handling; Part II: Calling conventions, asynchrony, optimizers, and debuggers. *The Journal of C Language Translation* 1994; **6**(1):20–32.
13. Goodenough JB. Exception handling: Issues and a proposed notation. *Communications of the ACM* 1975; **18**(12):683–696.
14. Freund SN, Mitchell JC. A type system for object initialization in the Java TM bytecode language. *Technical Report CS-TN-98-62*, Department of Computer Science, Stanford University, April 1998.
15. Briggs P. Register allocation via graph coloring. *PhD Thesis*, Rice University, April 1992.
16. Sun Microsystems Incorporated. The Java HotSpot$^{TM}$ performance engine architecture: A white paper about Sun's second generation performance technology, April 1999.
17. Ishizaki K, Kawahito M, Yasue T, Takeuchi M, Suganuma T, Onodera T, Komatsu H, Nakatani T. Design, implementation, and evaluation of optimizations in a just-in-time compiler. *ACM 1999 Java Grande Conference*, 1999.
18. Hummel J, Azevedo A, Kolson D, Nicolau A. Annotating the java bytecodes in support of optimization. *Concurrency: Practice and Experience* 1997; **9**(11):1003–1016.
19. Azevedo A, Nicolau A, Hummel J. Java annotation-aware just-in-time (AJIT) compilation system. *ACM 1999 Java Grande Conference*, 1999.