Java Bytecode Annotations Types and Formats

Ana Azevedo*, Alex Nicolau

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

Joe Hummel

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

UCI-ICS Technical Report No. 99-27

June 4, 1999

Abstract

This paper summarizes the Java Bytecode Annotations currently supported or under implementation in our **A**nnotation-generating **J**ava **B**ytecode **C**ompiler (AJBC) and in our **A**nnotation-aware **JIT** compilation system (AJIT). We explain the meaning of the annotations types, how they are generated by the Java Bytecode compiler, the formats for encoding annotations in the class file and how annotations are employed by a JVM engine (an interpreter or a JIT compiler) to produce high performance code. We analyze the potential benefits and costs of the different types of annotations and identify possible improvement and extensions.

1 Introduction

We designed the Java Bytecode Annotations [3, 7, 8] as an engineering solution to improve the speed of Java interpretation or JIT compilation and the quality of the code generated by JVM engines. We are investigating a complete set of annotations that can simplify the work of an interpreter and JIT compiler in producing high performance code by overcoming the Java stack language lack of expressiveness for traditional compiler optimizations and by passing on to the JVMs compiler analyses free of run-time costs. We have an initial list of annotations that can support building fast and efficient run-time algorithms for register allocation, code improving transformations, simple and advanced instruction scheduling optimizations and memory system optimizations. Our annotations carry information at bytecode level and at the level of the intrinsic implicit sub-operations that compose some bytecodes.

The annotations are extra information that offer benefits but do have a cost associated with them. Annotation overhead results from many factors: (1) the larger class file size (which increases download time), (2) the interpretation of the information conveyed in the annotation bytes, and the demand for

*This work supported in part by CAPES.

extra resources (memory for storing annotations). However, given the complexity of the information they can convey, annotations require simple run-time intermediate representation (IR) and 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 optimizing JVM engines that need complex IR to enable advanced compiler transformations. Another aspect of our annotations scheme is that annotations need to be verified before employed, just like untrusted bytecodes do. This is a parallel research direction we are pursuing.

Annotations types and formats have been presented along our previous works [3, 7, 8] and this current paper we lead a detailed discussion on each of them. Figure 1 summarizes the various annotations types we have thought as important and in the sections to follow shortly we explain each of the components integrating the Annotation Generator in our AJBC annotation-generating Java Bytecode Compiler.

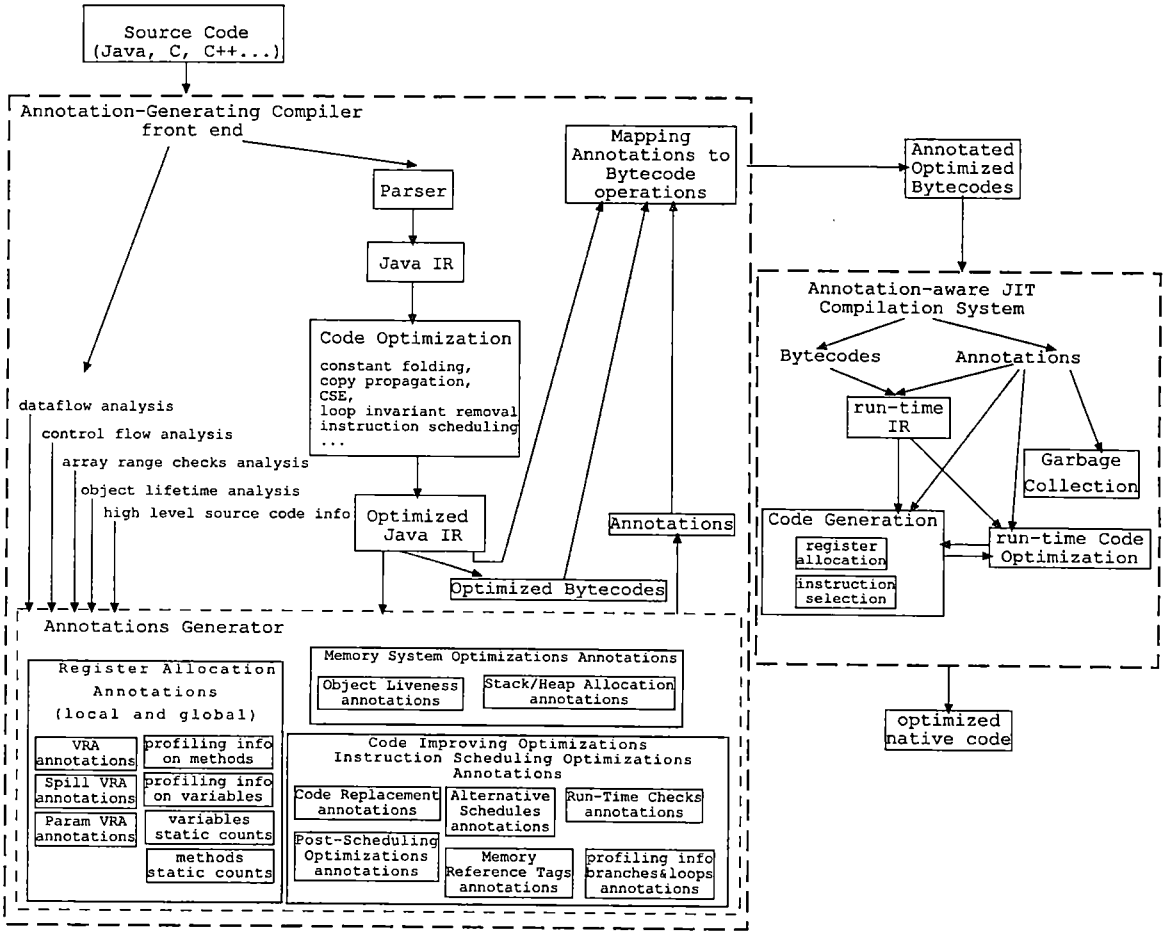

Figure 1: Annotation-generating compiler (AJBC) and annotation-aware JIT (AJIT) system

2 Annotations Types and Formats

2.1 Annotations for Register Allocation Optimizations

Register allocation is the most important optimization to exploit today's CPUs. Therefore annotations for register allocation were the first ones we designed and implemented in our annotation-aware compilation system [3]. We divide our register allocation annotations in 3 different types: **V**irtual **R**egister **A**llocation (VRA) annotations, **S**pilling **V**irtual **R**egister **A**llocation (SVRA) annotations and Parameter Passing **V**irtual **R**egister **A**llocation (PVRA) annotations. VRA annotations provide the minimum information to guide an efficient method level register allocation at run-time. SVRA annotations can be combined with the first to reduce the amount of spilling at method call boundaries. PVRA annotations are a more specific type of annotations that can be added to take advantage of calling conventions on machines that have such feature. An important aspect to remark is that these annotations are all machine independent optimization information.

2.1.1 Virtual Register Allocation (VRA)

Annotations Semantics

Virtual Register Allocation annotations represent the result of performing register allocation assuming an infinite number of registers, therefore *virtual* registers. The information provided by the VRA annotations is then used by the JVM engine to

1. perform a fast and efficient dynamic mapping-based register allocation
2. perform elimination of common sub-expressions that could not be removed due to the Java stack language restrictions. The annotations indicate which bytecodes (or bytecode sub-operations) are redundant or subsumed by preceding operations; such operations need not be translated into native code.

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 index calculation in an array load operation). For each bytecode instruction one or more VRA annotation formats exist. Each format indicates how a particular bytecode sub-operation should be translated: where to read its input operands, where to write the result, and perhaps whether or not this operation should be skipped entirely (e.g. when a previous operation has already computed the needed value).

Figure 2 shows an example of correspondence between bytecodes, Java IR and VRA annotations formats. Each `SRC`, `EXTRA` and `DEST` fields hold virtual register numbers representing the operands for the

sub-operations. In Case 1 of Figure 2, 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 2 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 2: Example of VRA annotations for `iaload` operation

Local variables and compiler generated temporaries (e.g., the operands for bytecode sub-operations in Figure 2) are treated the same way. They are directly mapped to virtual registers. As shown in Figure 3 local variable accesses (e.g, `iload` and `istore`) are represented in our Java IR as `nop` operations or move operations, annotated as `SRC-DEST`, `CONST-DEST`, `CONST` or `SRC`, depending on the result of optimizing the Java IR via copy propagation. When the JIT interprets the annotation formats `SRC` or `CONST`, it has the information that either (a) the local variable is in a virtual register indicated by the byte following the format header, or (b) it is a constant. In both cases, no machine code is generated for the bytecode.

In our VRA annotations class member variables are kept as variables in memory in our front-end compiler and are accessed via load and store operations, as shown in Figure 4 for bytecodes `getstatic`, `getfield`, `putstatic` and `putfield`. As a consequence, these variables are also kept in memory in our AJIT system. To enable some optimization on accesses to class member variables, we devised annotations that make explicit the variable address calculation, just like those in array references. For example, bytecode `getfield` has

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

Figure 3: Example of VRA annotations for local variables accesses

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 SRCADDR
	{b,c,s,i,l,d,f,a}st CONST, (V2)	CONST SRCADDR
	{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
pufield	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 SRCADDR
	{b,c,s,i,l,d,f,a}st CONST, (V1)	CONST SRCADDR
	{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 4: Example of VRA annotations for class member variables accesses

the different annotation formats `SRC-DEST` and `EXTRA-EXTRA-EXTRA-DEST` which state whether or not the variable's address has already been computed. We allow another optimization where class member variables, for some specific pieces of code have a chance of being directly mapped into physical registers skipping the load and store operations above mentioned. This information is conveyed to the JVM engine by annotating the bytecode `getfield` with the `SRC`, `CONST`, just like we do for local variables. Bytecodes `putfield`, `getstatic` and `putstatic` have similar annotations. Without interprocedural analyses, this optimization can be done for sequences of code where accesses to class member variables are not intercalated with method calls or the accesses occur at program points where exceptions cannot be thrown. In the presence of certain method calls it is still possible to allow this optimization if it can be checked by the compiler that the method call has no side effect on the variable. Figures 5, 6, 7 and 8 show several optimizations that are possible with class member accesses. They vary from simple common sub-expression elimination of addresses computation to mapping of class members directly to virtual registers. The figures show the Java source code, the Java IR generated in our AJBC system and the resultant annotated bytecode stream. Figure 5 shows the annotated bytecode with no optimizations. Figure 6 shows the result of eliminating addresses computation. Figure 7 shows what happens to the code when there is a method call we do not know its side effects. Optimizations could not be applied in this case. Finally, in Figure 8 all class member variables were directly allocated to virtual registers and at run-time will become candidates for register allocation.

Annotations Encoding Format

Each bytecode annotation has a header that indicates how the following annotation bits should be read and interpreted. In our initial encoding strategy all operations have a byte-long header followed by a variable number of bytes representing each virtual register number. An alternative encoding strategy is to let the header vary in size as for some operations few bits are enough to encode the header format. In this encoding scheme up to 255 virtual registers can be labeled. This is a fairly high amount of virtual registers to represent long Java method codes given that the register allocation algorithm always try to reuse virtual registers (see the details on our compile-time virtual register allocation algorithm in [3]).

Generating Annotations in a Java Bytecode Compiler

The choice of which virtual register to hold an operation's operand is crucial to the register allocation done at run-time. In order to enable a fast and efficient dynamic register allocation, the VRA annotations must convey the order in which variables should be allocated to physical registers (and thus which should be spilled if necessary). This is accomplished by assigning, at compile-time, the lowest virtual register numbers to the most important variables in the code. Then, at run-time, the register allocator should assign the lowest virtual register numbers to the physical machine registers. Our compile-time register allocation algorithm [3] implements a modified priority-based graph-coloring algorithm.

Employing Annotations in a JVM engine

The run-time register allocator is a fast and effective algorithm that essentially maps each virtual register to a machine register, prioritizing the assignment of lower virtual register numbers. This guarantees that

high priority values (program variables represented by lower virtual register numbers) have preference in the register assignment. When the number of physical registers is exhausted, virtual registers are mapped to temporaries on the stack. The register allocator reserves some of the machine registers for evaluating expressions that involve such variables that are not mapped into machine registers. Our register allocation algorithm uses a mapping table as an auxiliary data structure. The mapping table stores information on a virtual register number, a pointer to the corresponding physical register table entry, and the stack offset value it should use in case of spilling. In [3] we explain the details of the implementation of a mapping-based dynamic register allocator for the SPARC machine.

Annotations Benefits and Cost

The benefit of VRA annotations is that they provide a scheme that allows a graph-coloring quality register allocation with very low run-time cost as no time is spent on conflict graph construction, coloring nor dataflow analysis. It also conveys information on common sub-expression elimination and copy propagation. The cost of this type of annotation is high and is mainly determined by the cost of encoding virtual registers.

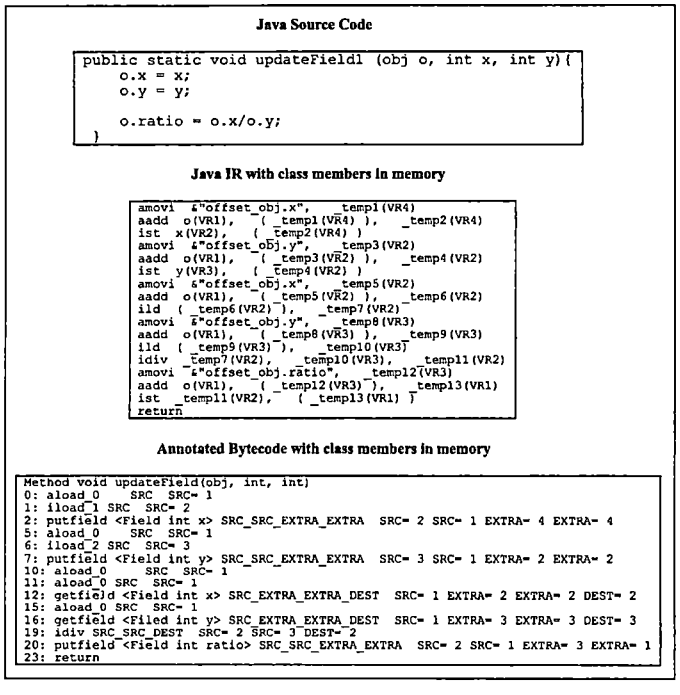

Figure 5: Resultant annotated bytecodes when class member variables are in memory

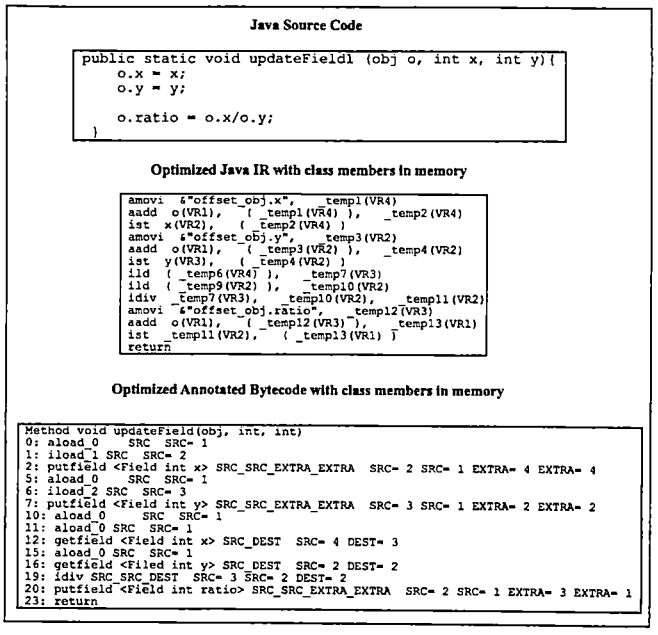

Figure 6: Resultant annotated bytecodes when class member variables are in memory and redundant addresses computations are eliminated

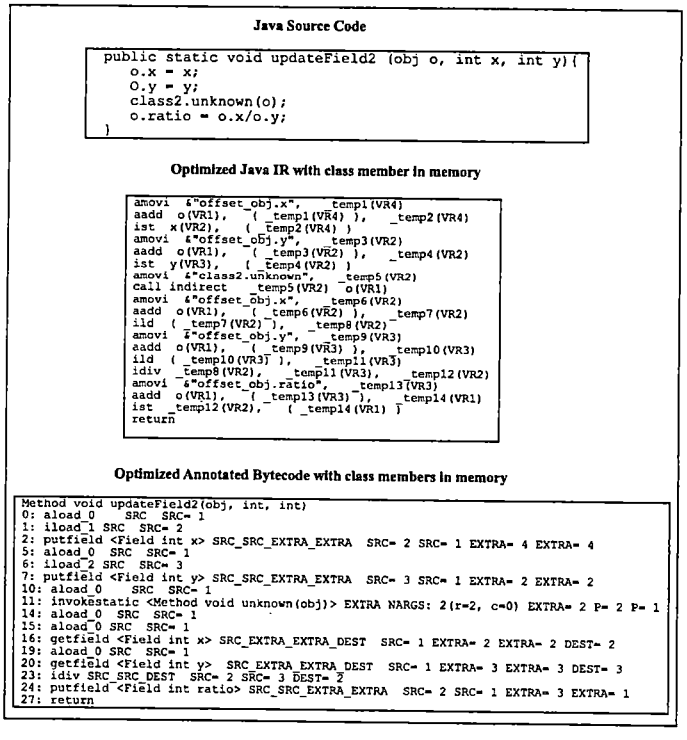

Figure 7: Resultant annotated bytecodes when class member variables are in memory and no optimization is possible

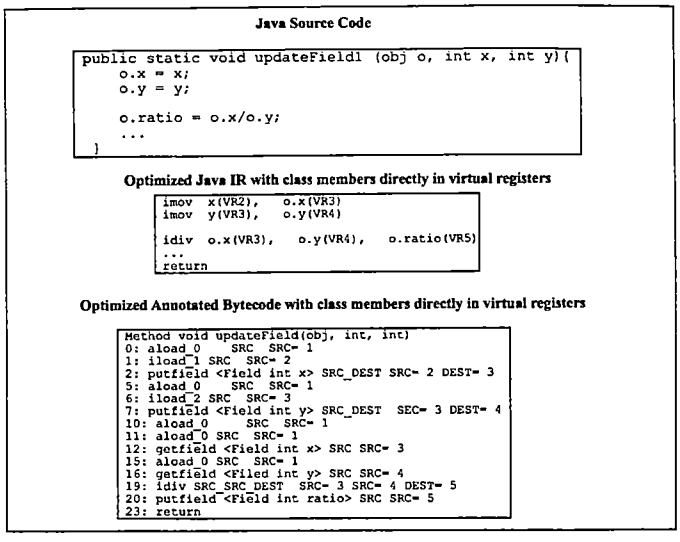

Figure 8: Resultant annotated bytecodes when class member variables are directly allocated to virtual registers

2.1.2 Spilling Virtual Registers Allocation Annotations (SVRA)

Annotations Semantics

When using VRA annotations to generate run-time register allocation at method call boundaries all virtual registers that got previously mapped into a physical register and are still active at the method call point have to be spilled and later reloaded to maintain correctness of the mapping-based register allocation. An improvement to our basic VRA scheme is to provide spill code annotations. This is accomplished with our SVRA annotations that indicate which virtual registers should be spilled at each method call.

Annotations Encoding Format

Each method call bytecode is annotated with a sequence of virtual registers numbers representing the values that should be spilled across the method call. Figure 9 shows the SVRA annotations format.

Bytecode	SVRA Annotation Formats
invokevirtual invokestatic invokespecial invokeinterface	TOTAL SRC SRC ... SRC 1 byte 1 byte ...

Figure 9: Example of SVRA annotations

Generating Annotations in a Java Bytecode Compiler

Traditional dataflow analysis can be used to generate SVRA annotations.

Employing Annotations in a JVM engine

When translating method call bytecodes the corresponding SVRA annotations are checked and spill code

is generated for each referenced virtual register that has been mapped to a machine register.

Annotations Benefits and Cost

The obvious benefit of this annotation is to reduce the amount of spilling code at method calls. The cost of this kind of annotations is moderate as method calls are frequent in Java and the number of method parameters is small in general [5].

2.1.3 Parameter Passing Virtual Register Allocation Annotations (PVRA)

Annotations Semantics

This is a machine independent annotation that allows the JVM engine to take advantage of calling conventions on certain machines. In SPARC, for example, arguments to functions are passed through special registers `o1-07`. If the values to be passed are not in the these machine registers, copy operations have to be inserted to move the values to respect the calling conventions. The idea of having PVRA annotations is to avoid such copy operations. If a value defined at a certain program point is later used as an input parameter to some method call in some path reached by this definition of the value then PVRA annotations would mark this value definition as a method argument giving hint to the run-time register allocator to do the correct mapping from virtual to physical register.

Annotations Encoding Format

Each bytecode that defines a variable is annotated with a bit `PARAM` indicating whether in the list of uses of this definition it is passed as an argument to a method. If positive other pieces of information that may be useful for the run-time register allocator are (1)the order in which the parameter is passed; (2)the types of the preceding parameters; (3) the target method call site; (4) whether the mapping from virtual register to the specific machine register should be fixed after the call. The first three information types can be encoded in two ways as illustrated in Figure 10. The first alternative encodes the bytecode sequence number that corresponds to the method call site (2 bytes at minimum and 4 bytes at most are needed to represent a bytecode address). With this information, the run-time system can load the corresponding class and retrieve the method signature. The second encoding scheme provides the same information by annotating the bytecode with the argument passing order and the Java types of all preceding arguments. The last annotation information is a bit-long field `FIXED-MAPPING` in the PVRA annotations that instructs the JVM engine to keep the mapping fixed and survive the method call.

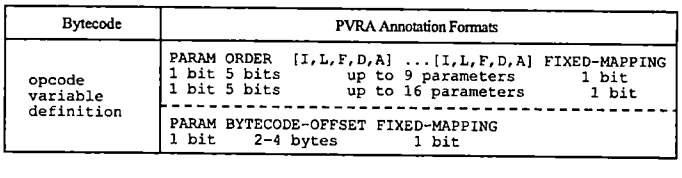

Figure 10: Example of PVRA annotations

Generating Annotations in a Java Bytecode Compiler

PVRA annotations can be generated using traditional data flow analysis and profiling information or static counts to estimate method calls priorities. As in Java method calls are very frequent, the potential cost of spilling a virual register assigned to a calling convention machine register to free the latter for other method calls may offset the cost of producing the copy operations we want to avoid. We initially mark only (1) definitions which last use is a method call, and all operations in the code between the definition and the method call are not method calls or are method calls that require no parameter passing or require parameter passing in non-conflicting order; (2) or definitions for which all their uses following the method call are not other method calls requiring parameter passing. If there is more than one method call reached by the value definition, the method call with higher priority is the one annotated.

Employing Annotations in a JVM engine

A JVM engine uses PVRA annotations information when mapping a virtual register to a physical register. It first checks whether the annotation bit indicates the value is a method argument. If yes, it assigns a physical register respecting calling convention. For doing so, the allocator checks the rest of the PVRA annotation bits for the order of arguments and their types in the target method call. This assignment is fixed up to the method call. At this point the machine register may be freed, according to the information in the `FIXED-MAPPING` annotation bit.

Annotations Benefits and Cost

These annotations are useful on machines with special calling conventions. They have less performance impact as compared to the two previous annotations. The cost is low to moderate as method calls are frequent in Java programming but our conservative code analysis reduces the number of method call candidates.

2.1.4 Virtual Register Allocation Annotations Improvement and Extensions

A first improvement in our VRA scheme is to allow global virtual register allocation. In this way class variables are directly mapped into virtual registers much like the way local variables and compiler generated temporaries are treated. Class member variables are prioritized together with all other variable types and get a chance to be fixedly mapped to a machine register at run-time. This optimization is possible if we have whole program information, i.e., all class files of an application are available at bytecode generation time. If in the application there are calls to methods we do not have the code or the code is available but may have some side effect on these variables (e.g., an exception may be thrown), at the entry point of these unsafe program points a spill annotation is generated. When implementing global virtual register allocation we make use of SVRA annotations and extend these annotations not only for method calls but for all bytecode operation types. An extra annotation bit indicates whether SVRA annotations are present or not at a bytecode. Traditional interprocedural analysis together with profiling information or static frequency of methods and call graph analysis can be done at compile time to extract information for global register allocation annotations.

Related to our global virtual register allocation is Wall's approach for interprocedural register allocation at link-time introduced in [11]. In that case, all object files, i.e., the whole program, is available for the link-time register allocator. His approach allows object files to be compiled separately and implements local register allocation. While doing this register allocation annotations are generated containing actions to be taken in case at link-time global register allocation is attempted. At link-time his approach "fixes" the register allocation of individual modules to decrease the interference between the independent allocation decisions.

Though we can do global virtual register allocation to guide our mapping-based dynamic global register allocation much like in the way Wall's approach does link-time global register allocation, the Java context we work on is sometimes different as classes and methods are loaded on demand and whole program information is not available not even at run-time.

Our annotations scheme provides a way to do efficient local register allocation at run-time and we can think of further improvement that allows incremental cross module register allocation even if not all classes and methods are available at run-time. In this new scheme the annotations for register allocation described above do not change. What changes is the way our dynamic register allocation works. In this new approach, a method gets loaded and the dynamic register allocation is invoked at each bytecode. For method call instructions for which profiling information or static frequency for the method being called indicate it is more frequently executed than the one been compiled, and if it is a leaf method, the register allocation for the current method is interrupted, and register allocation of the called method is done first. Combining the information provided by the SVRA annotations together with new annotations for static frequency or profiling information on methods, locals and global variables accessses per method it is possible to identify the virtual register to physical register mappings that should be maintained fixed in the calling method and which physical registers are not available for allocation in the called method. After identifying such machine registers, local register allocation in the called method proceeds as normal, using the VRA annotations and only then the calling method register allocation is subsummed. If it happens that the called method is not a leaf method or profiling information indicates it has no priority over the calling method, there is no interruption in the allocation of the latter. This kind of cross module register allocation helps optimizing call costs at some program points.

2.2 Annotations for Code Improving and Instruction Scheduling Optimizations

Our VRA scheme serves as a mechanism for identifying redundant bytecode operations and also redundant bytecode sub-operations and eliminating them, conveying information to the JVM engine to do local common subexpression elimination and copy propagation with no run-time cost (except for the annotations overhead). In this section we identify other common compiler optimizations that would be interesting for the JVM engine to apply to generate high performance code at low cost. We designed annotations that carry

information on code replacement and alternative schedules, annotations for run-time check elimination, annotations for memory disambiguation and annotations for simple post-scheduling optimizations (peephole optimizations). These annotations convey information of different levels of complexity targetting a wide range of optimizations, some more applicable to interpreters others more to optimizing JIT compilation systems.

2.2.1 Code Replacement and Alternative Schedules Annotations

Annotations Semantics

In Code Replacement annotations each bytecode is annotated with a future bytecode operation that can be moved up and code for it can be generated at the time of producing code for the current bytecode, or each bytecode can be annotated with a future bytecode it can be moved down with, delaying its translation. In case the bytecode operation candidate for replacement is composed of implicit sub-operations, each sub-operation will have an independent annotation of how far it can be moved. This kind of annotation is useful for simple optimizations such as expressing loop invariant removal or instruction scheduling optimizations such as branch scheduling. At run-time the JVM engine can ignore this extra annotation or use it if it results in some benefit in the target architecture.

Extending on the idea of Code Replacement Annotations, we designed the Alternative Scheduling annotations that improve on the former by supporting fancier instruction scheduling optimizations such as, list scheduling, trace scheduling and software pipelining. These traditional compile-time optimizations depend on machine specific information but the annotations can carry the best machine independent scheduling alternatives that are evaluated only at run-time for selection of the best choice for the target architecture the code is being compiled to. When encoding such annotations we indicate the corresponding virtual register allocation annotation that should be used in case the optimization is carried out.

Annotations Encoding Format

Our current annotations for code replacement supports loop invariant removal and is illustrated in Figure 11. It is encoded as a list of pairs `(bytecode sequence number, (bytecode sequence number, sub-operation sequence number)...)`. The first bytecode number represents the point where translation process should consider the translation of other bytecode sub-operations suggested in the subsequent pairs `(bytecode sequence number, sub-operation sequence number)` in the alternative schedule annotation.

Generating Annotations in a Java Bytecode Compiler

We have experimented with generating annotations for loop invariant removal. Traditional compilation analysis techniques could be applied for generating the Code Replacement annotation. All that was necessary was to keep track of which operations in our Java IR corresponded to which bytecode operation. If after applying the optimization the bytecode sub-operations did not get split, or a valid bytecode sequence can be generated from the optimized Java IR, no annotation is generated. Otherwise, an annotation is generated indicating how the bytecode sub-operations should be split.

Bytecode	Code Replacement Table Annotation Format
generalopcode BYTECODE-OFFSET 2-4 bytes	BYTECODE-OFFSET SUB-OPERATION-OFFSET 2-4 bytes 3 bits

Figure 11: Example of Code Replacement Annotation Format

Other optimizations for instruction scheduling may require a change in the traditional compilation techniques to adapt them back to more machine independent optimizations or their respective heuristics adjusted to other cost functions that take into account the fact that we are trying to generate the best machine independent optimization we can and the encoding of it as annotations has an associated cost, such as the size of the annotated class file, the complexity for interpreting the annotations in the JVM engine. As we try with other optimizations the format of alternative schedule annotations will be redefined and improved.

Employing Annotations in a JVM engine

When doing our virtual register allocation all basic compiler optimizations such as common sub-expression elimination, copy propagation and loop invariant removal have already been applied to the code. Therefore, the VRA annotations generated take into account the effect of all these optimizations and the annotation for code replacement is not optional and has to be encoded in the class file and interpreted by the JVM engine as they carry extra information that could not be conveyed in the Java stack language program representation. The first action of the JVM engine is to check the presence of Code Replacement annotations in the class file and as it translates the bytecode stream it checks whether a bytecode sequence number matches the pairs listed in the annotations bytes. If so, translation of the named referenced bytecode sub-operations is carried out. This ordering is a consequence of our implementation. It does not reflect a restriction on the interaction among different annotations types.

Annotations Benefits and Cost

The benefits and costs of such annotations depend on the optimizations we want the annotations to support. For loop invariant removal optimization the cost of the annotations is low in space consumption and the benefit is high as we allowed the JVM engine perform a very basic and traditional optimization with no run-time cost and we completely overcame the innefficiency of Java stack language in expressing this optimization.

2.2.2 Run-Time Checks Annotations (RTC)

Annotations Semantics

Our run-time check elimination annotations have been explained in [8]. They serve as a mechanism for selectively disabling implicit Java run-time array bounds checks.

Annotations Encoding Format

As specified in [5], for array references, three kinds of run-time checks need to be performed: whether the array object reference is not NULL, the array index is greater than zero and less than the array size. This information can be encoded using 3 bits as shown in Figure 12. Another possible encoding combines the array index range checks annotation bits into 1 bit, assuming the JVM engine, via an unsigned comparison can check both range limits.

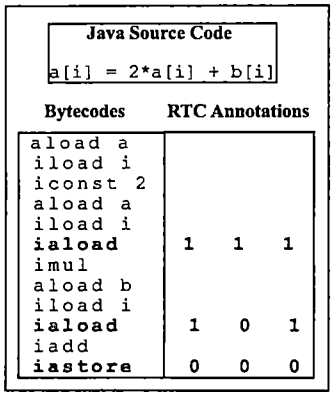

Figure 12: Example of Run-Time Checks Annotations

Generating Annotations in a Java Bytecode Compiler

For generating RTC annotations we reused array bound checks analyses as decribed in [6, 9].

Employing Annotations in a JVM engine

The JVM engine uses this annotation in a very simple way. Depending on the value of the annotation bits, extra code for performing the checks is generated when translating the bytecode.

Annotations Benefits and Cost

The cost of RTC elimination annotations is low and the benefit is high. It eliminates redundant checks that involve branches that are always expensive operations even in current CPUs.

2.2.3 Memory References Tags Annotations

Annotations Semantics

Memory References Tags annotations were introduced in [8] and they were designed based on our past experience with instruction scheduling optimizations [10] where we noticed the need for high level source code information to produce better memory disambiguation analyses. Pointer analyses are an important problem in optimizing C, C++ code. Though Java disallows explicit pointer arithmetic it also suffers from ambiguity in memory references which prevent code optimizations. Figure 13 shows such problem, where array references `a` and `b` may be alias to the same array object.

This kind of annotation could be useful in choosing among alternative schedulings for a method code.

```
public void foo2(int a[], in b[], int i){
  a[i] = 2*a[i] + b[i];
}
```

Figure 13: Example of memory ambiguation problem

For example, observe the code in Figure 14. Suppose when compiling class Class1, the code for method `Class2.foo2()` was not available and therefore in the compilation of classes Class1 and Class2 we could not use the information that the parameters to `Class2.foo2())` reference the same objects. When compiling Class2, suppose method

ttClass2.foo2 was identified as having high execution frequency. In this case two alernative schedules were generated for method `Class2.foo2()`. At run-time, having the memory reference tag annotation helps the JVM engine to choose the correct and most optimized code for method `Class2.foo2()`. If the alternative schedule is not generated at compile time, we can use the memory reference annotations to help the JVM engine implement such optimization at run-time.

```
class Class1{

  public static void foo1(){
     a =...
     b =...
     Class2.foo2(a, a);
  }
}
```

```
class Class2{
  public static void foo2(int a[], int b[], int i){
     ...
     a[i] = 2*a[i] + b[i];
     ...
  }
}

optimized code: a[i] = 3*a[i]
non-optimized code: a[i] = 2*a[i] + b[i]
```

Figure 14: Example of Memory Reference Tag Annotations use in selecting alternative instruction schedules

Annotations Encoding Format

Different encoding formats can be designed depending on the target optimization. We can propose an encoding scheme where each bytecode that manipulates a variable is annotated with an encoded form of the name of the variable and of its memory allocation strategy as shown in Figure 15. This information can be used as memory disambiguation information in an optimizing JVM engine. If the JVM is not an optimizing engine, still better code quality can be achieve by having annotations for alternative method scheduling and opting for one of them depending on the information provided by memory reference tags annotations. In this second case we could think of an encoding scheme that for each method call bytecode annotates information on the variables passed as argument. At translation time, the JVM engine checks the presence of memory reference tags annotations in the calling method and alternative scheduling annotations in the called method and a decision is made to pick the most appropriate one.

Generating Annotations in a Java Bytecode Compiler

Traditional memory disambiguation analyses can be used to generate memory reference tags annotations.

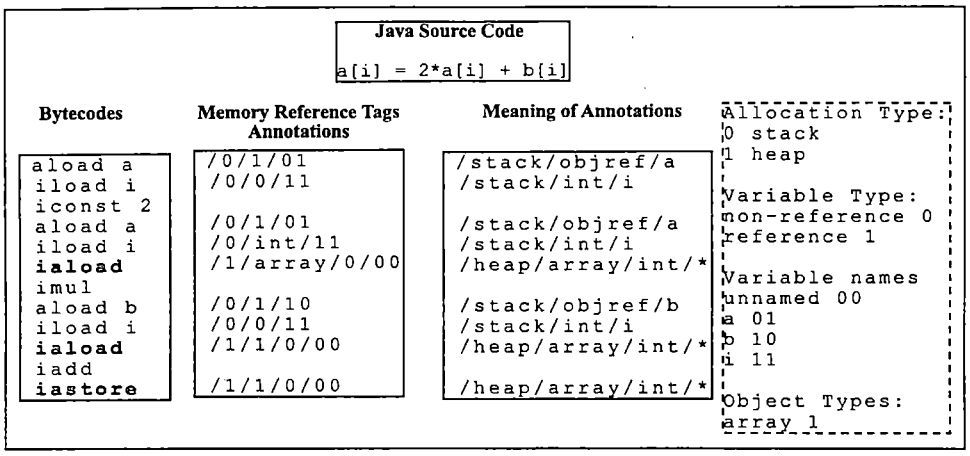

Figure 15: Example of Memory Reference Tag Annotations Encoding

Employing Annotations in a JVM engine

The Memory Reference Checks annotations can be used in guiding run-time code optimizations, eliminating the need for memory disambiguation analyses at run-time or can be used by a non-optimizing JVM to decide on alternative scheduling annotations, and therefore producing high quality code without no run-time overhead.

Annotations Benefits and Cost

The cost of Memory Reference Checks annotations can be high due to the potential need for encoding all method variables and their memory hierarchy paths and the frequency of method calls in Java codes. The benefits, bearing on past experience, are moderate.

2.2.4 Simple Post-Scheduling Annotations

Simple Post-Scheduling annotations help the JVM engine to produce code that better uses the target machine idioms. Although these annotations are still machine independent, the idea is to identify and mark blocks of bytecode instructions that can be performed by common special machine instructions. Special machine instructions can be seen as super-operators that combine two or more basic machine instructions. This kind of instructions are very common in current architectures such as Sun SPARC, IBM PowerPC, DEC Alpha, Intel i386, HP PA-RISC, Motorola 68000, MIPS.

Annotations Semantics

Some machines combine simple operations into more powerful ones. For example, in SPARC, a simple translation of the machine code sequence `sub i,10, result; subcc result, 0, 0; be` can be simplified by `subcc b,c, 0; be;`. In PA-RISC, a loop condition test and the increment of the condition variable can be transformed into `addBT, <= i limit Label`. Using annotations we can mark the bytecodes that form a common instructions combinations listed in a predefined set of super-operators. At run-time it is up to

the run-time engine to check the applicability of this instruction combination for the target machine it is generating code for and, if super-operators overlap, the order in which the instruction combination is applied depending on the benefit it results.

Annotations Encoding Format

Figure 16 shows an example of how Post-scheduling annotations can be encoded. When generating code for SPARC machines, bytecodes 3 and 4 can be combined and the translation produces one less machine instruction. When generating code for PA-RISC, bytecodes 3, 4 and 10, representing a loop construct can be combined and only one machine instruction is generated. One possible annotation encoding scheme would create a table of pointers to bytecodes that should be combined forming a certain pattern and which bytecodes should be skipped if this particular combination is valid on the target machine.

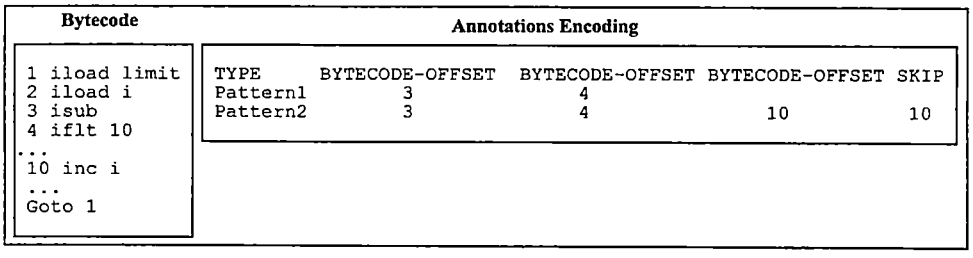

Figure 16: Example of Post-Scheduling Annotations Encoding

Generating Annotations in a Java Bytecode Compiler

By studying the instruction set of common architectures we can identify common combinations of basic machine instructions and use this information for producing the annotations. The analysis required in the front-end is basically pattern matching. For some fancier instruction combinations, control flow information (e.g., identification of loop constructs, for statements, if-then-else statements) may be necessary.

Employing Annotations in a JVM engine

These annotations can be used in a JVM engine as the last phase of the code optimizing process or after all scheduling annotations and register allocation annotations have been processed. The run-time effort it demands is minimum as the JVM only has to check whether a certain marked bytecode block corresponds to a supported machine idiom. If yes, the substitution is carried out. An improvement when no other alternative scheduling annotation is present and that avoids any code rewriting is to process post-scheduling annotations as the bytecodes get translated.

Annotations Benefits and Cost

The benefit of this annotation type remains to be anlyzed. Most instruction combinations are very simple and should not be time consuming to be implemented in a JVM engine, either an interpreter or JIT compiler. However, if present, these annotations do save time as it frees the run-time engine from searching for bytecodes pattern matching, rewriting of code to correct instructions addresses, and in the case of more

complicate instructions combination, frees the JVM engine from producing control flow analysis to find out high level constructs, such as loops. The space cost is low to moderate.

2.3 Annotations for Memory System Optimizations

Another area we believe annotations information can speed up or improve on the work done in a Java run-time system is to provide information for efficient memory management. We have thought about kinds of information that can improve the efficiency of garbage collection and also that can reduce the frequency garbage collection is invoked. Precise information on variables types and objects liveness are compiler analyses information useful for the run-time system to accomplish such goals.

Related work in this area has shown how such analyses information can be generated at compile-time and at run-time and how a compiled code can carry such information. In [4] Diwan shows several things: how tidy and untidy pointers (those derived from expressions that use tidy pointers, resultant from program computation or compiler optimizations) can be tracked; their liveness information collected; at which program points garbage collection should be invoked and liveness information is needed (basically method calls and loops); when code has ambiguity on the values of pointers, how it can be modified to remove the ambiguity; and how liveness information can be efficiently encoded as sets of tables per garbage collection points and input in the compiled object code making it available to the run-time memory management system. In [2] the authors report an extension of the former work that brings it to the Java language context. They show how type information and liveness information can be combined to implement a more precise garbage collection system. Even in the strongly typed Java language type information can be ambiguous at certain program points and the authors describe a compiler analysis that can be done at load time to solve this problem. They show how liveness analysis can be collected at load-time and how it is useful in controlling heap size. In their findings they concluded that augmenting a type-precise garbage collector with live variable analysis reduces heap size by an average 11% and the cost of generating this information at load-time is 50% greater than the cost of generating only type information. The authors consider the possibility of pre-analyzing the code and inserting information into class files as additional attributes. They claim that this would be a solution for trusted class files only because for classes obtained over a network the verification process is more expensive than the re-computation of the information.

We can contribute in this area by proposing a good annotation encoding of the liveness information and encode a solution for ambiguity type information without code rewriting as the work listed above does. This guarantees a speed up for trusted classes. The paper does not comment on the complication of rewriting instructions at load-time (to remove code ambiguities that complicate garbage collection) but acknowledges the complexity of generating type and liveness information as classes are dynamically loaded. Having annotations can speedup part of the process and may be helpful as a starting information for tracking liveness of untidy pointers that can appear as a result of translating bytecodes into machine codes.

A more original work direction is to use livessness information to identify short-lived objects and for them allocation is done on the stack instead of on the heap. These objects get allocated and freed by the JVM engine without calling garbage collection memory management functions. Short-lived objects do not have much impact on heap size but they reduce the amount of garbage collection points decreasing the frequency garbage collection is invoked.

2.3.1 Object Liveness and Heap/Stack Allocation Annotation

Annotations Semantics

Object liveness annotations, indicate, which variables are object references and are live at each garbage collection point, providing information for more precise garbage collection. Heap and Stack Allocation annotations use this liveness information and the object structure and size to suggest an allocation strategy for the object. Object allocation bytecodes get annotated with this information.

Annotations Encoding Format

What needs to be encoded by object liveness information is not program variables (Java method local variables) but stack slot positions these program variables are mapped to. This happens because in Java, variables of different types may end up been allocated in the same stack slot in the stack frame (depending on how the compiler generates code, as shown in the code example in Figure 17 extracted from [1]. As seen in this example, the stack layout is not simple to figure out and in some cases may be control flow dependent. The difficulty is to calculate precise stack layout for each garbage collection point at run-time. If we can have this information already computed, time is saved. Schemes for encoding this information already exist and our work would be to adapt them to our annotations scheme with low decoding overhead. Most stack maps are encoded as tables. As tables tend to repeat from one GC point to another, only the differences from a common main base table are annotated. In Java, GC points would be object allocation bytecodes; method invokations we do not know the code at compilation-time or that include instructions for allocating objects; and loops (to avoid storage retention if the loop operations do not include allocation instructions).

Our Heap/Stack allocation is much less expensive to encode. For each object allocation bytecode a bit `STACK-ALLOCATION-SUGGESTED` is set according to the result of the compiler analysis.

```
...
if (b){
   int i;
   ....
}
else{
   object o;
   ....
}
...
```

Figure 17: Example of Java code that can lead to ambiguous stack slot mapping

Generating Annotations in a Java Bytecode Compiler

We would reuse the compiler analyses from [2, 4] and we would create new analysis for choosing objects for stack allocation. The size of an object, the types of the object's fields (whether the fields are reference or non-reference fields) are the information that will guide our analysis.

Employing Annotations in a JVM engine

Annotations for liveness information are used by garbage collection functions as described in the literature. Annotations for stack/heap allocation guide the invokation of memory allocation functions that reserved memory space that may be garbage collected or not.

Annotations Benefits and Cost

Previous work has analyzed the benefits of liveness information for precise garbage collection in the context of non-generational collector. Remains to be seen how it impacts other types of collectors. We are not aware of any JVM implementation that has tried different strategies for Java objects allocation. The benefits of our stack/heap allocation annotations remains to be checked. Previous work report that the cost of encoding liveness information is as high as 16% of the code size. The cost of annotations for allocation strategy is very low, one bit per allocation bytecode.

3 Conclusions

In this report we listed an initial set of annotations that we believe are useful in speeding up the work and the quality of code generated by JVM engines. This is not yet a complete set of Java Bytecode annotations but they represent attempts to optimize three important aspects of any high performance Java application code: the quality and run-time applicability of register allocation and instruction scheduling optimizations and the reduction of the impact of the Java memory management scheme on the application running time. The annotations set presented here reflects what we have implemented and what is under implementation in our annotation-generating compilation system and in our Annotation-aware Java Virtual Machine (AJVM) engines.

References

[1] Ole Agesen and David Detlefs. Finding References in Java Stacks. *OOPSLA Workshop on Garbage Collection and Memory Management*, 1997.

[2] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. *Proceedings of ACM Programming Languages Design and Implementation*, June 1998.

[3] A. Azevedo, J. Hummel, and A. Nicolau. Java Annotation-aware Just In Time (AJIT) Compilation System. *To appear in ACM Java Grande Conference*, June 1999.

[4] Amer Diiwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. *Proceedings of ACM Programming Languages Design and Implementation*, June 1992.

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

[6] Rajiv Gupta. Optimizing Array Bound Checks Using Flow Analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, March 1993.

[7] 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.

[8] J. Hummel, A. Azevedo, and A. Nicolau. Annotating and Optimizing the Java Bytecodes. *Workshop on security and efficiency Aspects of Java, Eilat, Israel (part of MASCOTS-97)*, January 1997.

[9] P. Kolte and M. Wolfe. Elimination of Redundant Array Subscript Range Checks. Technical report, Oregon Graduate Institute of Science and Technology Technical Report, 1996.

[10] S. Novack, J. Hummel, and A. Nicolau. A simple mechanism for improving the accuracy and efficiency of instruction-level disambiguation. In C. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Eighth International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 289–303. Springer-Verlag, 1996.

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