Annotating the Java Bytecodes in Support of Optimization

Joseph Hummel[*]
University of Illinois, Chicago

Ana Azevedo[†], David Kolson, Alexandru Nicolau
University of California, Irvine

Abstract

The efficient execution of Java programs presents a challenge to hardware and software designers alike. The difficulty however lies with the Java bytecodes. Their model of a simplistic, platform-independent stack machine is well-suited for portability, though at the expense of execution speed. Various approaches are being proposed to increase the speed of Java bytecode programs, including: (1) on-the-fly compilation to native code (also known as JIT or "just-in-time" compilation); (2) traditional ("ahead-of-time") compilation of bytecodes to some higher-level intermediate form and then to native code; and (3) translation of bytecodes to a higher-level language and then use of an existing compiler to produce native code. Speedups on the order of 50 over standard bytecode interpretation have been claimed.

All of these approaches rely upon bytecode analysis (of varying sophistication) to extract information about the program, which is then used to optimize the native code during the translation process. However, extracting information from a lower-level representation such as the Java bytecodes can be very expensive. And given the fact that most approaches for executing Java bytecodes cannot spend a great deal of time recovering high-level information, the solutions adopted during the translation process must use faster and less accurate analysis techniques, thus penalizing the quality of the native code.

In this paper we propose an optimization approach based on bytecode annotations. The bytecodes are annotated during the original source code to bytecode translation, allowing both traditional interpretation by a JVM and aggressive optimization by an annotation-aware bytecode compiler. Annotations do not hinder portability nor compatibility, while preserving optimization information that is expensive to recompute. Preliminary results yield bytecode with C-like performance using JIT technology.

1 Introduction and Motivation

On modern CPUs, direct interpretation of all but the simplest Java bytecode programs is horribly inefficient. Speedups of 5 to 10 over interpretation are commonplace [DS96, HGH96], while speedups of 50 have been claimed [Inca]. The inefficiencies lie not with Java itself, but with the definition of the Java bytecodes.

[*]This work supported in part by ONR grant N00014-93-1-1348.
[†]This work supported in part by CAPES. Email aazevedo@ics.uci.edu

Their model of a simplistic, platform-independent stack machine is well-suited for portability, though at the expense of execution speed.

One bytecode feature that accounts for the inefficiency of the Java portable representation is the stack machine model. This model does not map directly onto today's CPUs, which rely heavily upon registers, caches, and sophisticated instruction scheduling to achieve high performance. Firstly, the Java bytecode virtual machine (JVM) provides no operand registers and instead requires the use of a stack. In addition, the JVM stack model sequentializes computation and prevents the reuse of values, since operands must always be pushed (copied) onto the top of the stack. In fact, Sun Microsystems conservatively estimates that 40% of all instructions executed are loads and stores to and from the stack [Way96]. Secondly, the Java bytecodes provide no means for expressing the result of many common and important optimizations. For example, register allocation is a critical optimization that is expressed in the native code produced by most compilers. However, bytecodes provide no means for referencing registers, thus preventing the expression of register allocation. Other important optimizations which cannot be expressed include:

1. instruction scheduling: access to only top of stack prevents reordering;
2. elimination of run-time checks: all checks are implicit in the bytecode;
3. automatic reclamation of memory: bytecodes provide no explicit free.

For example, consider the following single line of Java code inside a method `foo` (Figure 1):

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

Figure 1: Example `a[i] = (2 * a[i]) + b[i]`.

In essence, this code loads two array elements, performs a multiply and an add, and stores the result. Now consider the corresponding bytecode stream shown in Figure 2, modified slightly to increase readability (variable references are written using actual variable names, e.g. "`aload_0`" appears as "`aload a`"). Of particular interest are the columns denoting the number of loads, stores, and run-time checks performed by each bytecode instruction. Firstly, observe that **every** instruction writes to memory, and all but one instruction reads one or more values from memory. Secondly, note that every array access (`iaload` and `iastore`) conceptually requires three run-time checks as per the JVM specification: (a) the array reference is not Null, (b) the array index is greater than or equal to 0, and (c) the array index is less than the length of the array. Since these run-time checks are implicit in the bytecode, the JVM must perform these checks unless it can prove they are superfluous (which requires run-time overhead to track the necessary information). Since JVMs currently do not perform this optimization, each array access adds a hidden cost of two conditionals and one load[1].

Bytecode	Comments	LOADS	STORES	R-T Checks
`aload a`	`push ref to a`	1	1	
`iload i`	`push i`	1	1	
`iconst 2`	`push const 2`		1	
`aload a`	`push ref to a`	1	1	
`iload i`	`push i`	1	1	
	`run-time checks performed by iaload:`			
	`if (a==NULL)`			
	`throw NullPointerException;`			
	`if (unsigned(i) >= a.length)`			
	`throw ArrayIndexOutOfBoundsException;`			
`iaload`	`pop, pop, push a[i]`	3	1	2
`imul`	`pop, pop, push 2*a[i]`	2	1	
`aload b`	`push ref to b`	1	1	
`iload i`	`push i`	1	1	
	`run-time checks performed by iaload:`			
	`if (b==NULL)`			
	`throw NullPointerException;`			
	`if (unsigned(i) >= b.length)`			
	`throw ArrayIndexOutOfBoundsException;`			
`iaload`	`pop, pop, push b[i]`	3	1	2
`iadd`	`pop, pop, push (2*a[i]) + b[i]`	2	1	
	`run-time checks performed by iastore:`			
	`if (a==NULL)`			
	`throw NullPointerException;`			
	`if (unsigned(i) >= a.length)`			
	`throw ArrayIndexOutOfBoundsException`			
`iastore`	`pop, pop, pop, store into a[i]`	3	1	2

Figure 2: Java bytecodes for `a[i] = (2 * a[i]) + b[i]`.

As a result, our single line of Java code produces a bytecode stream with a sum total of 22 memory loads, 12 memory stores, 6 conditionals, and a few miscellaneous instructions. In all fairness, note that this is not the best possible bytecode stream—it is possible to reorder the code and then replace two consecutive instructions by a single `dup2` instruction. This "improved" bytecode sequence however still requires the same number of loads, stores, and conditionals.

The goal of our work is to achieve C-like performance, while retaining bytecode's advantage of instant portability and remaining backward compatible with existing JVMs. Our approach is based on bytecode annotation. During the *initial* compilation from source program to bytecode[2], the compiler behaves like a traditional optimizing compiler: it builds an intermediate form, performs various analyses, applies numerous optimizations, and then generates bytecodes. However, in our approach, for each emitted bytecode instruc-

[1] Even though there are conceptually three run-time checks that must be performed, note that the two index checks can be performed using a single unsigned compare. Thus, a total of only two conditionals are needed. The load is necessary to retrieve the array's length.

[2] Currently Java and Ada95 compilers exist for generating bytecode.

tion the compiler also emits a corresponding annotation. This annotation contains information concerning such things as register allocation, memory disambiguation, memory reclamation, and run-time checking—a concise summary of the same information that if lost, would have to be recomputed (at compile or even run-time). The annotation stream thus parallels the bytecode stream, though it is stored separately to enable existing JVMs to process the bytecodes without incident. However, an annotation-aware compiler, armed with this precomputed stream of annotation information, can now generate high-performance native code. Our preliminary results demonstrate this claim, achieving C-like performance from traditional bytecodes with on-the-fly technology. Furthermore, we expect that existing approaches will also find the annotations useful, since they offer information that is both precomputed and potentially unavailable otherwise.

The format of this paper is as follows. In the next section we discuss existing and related work, followed by a detailed discussion of our approach in Section 3. In Section 4 we present some preliminary results, followed by our conclusions in Section 5.

2 Existing and Related Work

To counter Java's bytecode inefficiencies, various approaches are being proposed. Existing approaches all perform some degree of bytecode analysis, followed by a translation to native code. The essential difference between these approaches is when, where, and how much time is spent in performing these steps.

The most popular approach is on-the-fly compilation, also known as "just-in-time" (or JIT) compilation [Incc, Incd, Incb, Wil, USCH92, GR83, Fra94]. Analysis and translation are done online, typically during run-time as each method (subroutine) is first called. However, analysis and translation can also be done at load-time (e.g. during the bytecode verification process), or lazily on a per-bytecode basis [Jon]. The native code is cached for reuse during the *same* execution run, but is generally not saved for reuse across execution runs. Though most JIT technology is proprietary and the resulting native code cannot be captured, commonly discussed approaches [Jon] include:

1. using machine's native stack in place of JVM stack,
2. using registers to hold top-most n stack values.

The advantage of a JIT approach is improved performance while retaining "instant" portability, the ability to download and run immediately. Though not important in traditional execution environments, this feature is critical in the world of networks and ever-changing bytecode streams. The performance improvement is also significant; a recent study of a number of JIT systems found speedups of 2-15 over direct interpretation; e.g. Microsoft's JIT compiler achieved a 15-fold improvement on an FFT benchmark [DS96]. The disadvantage of JIT compilers is that they cannot spend nearly enough time on analysis and optimization, otherwise translation time may well dominate execution time. JIT compilers generally perform only local or basic-block optimizations, and thus cannot perform global (method-level) analyses and optimizations (e.g. data and control flow, register allocation, instruction scheduling). This severely limits the impact of optimization.

The second approach follows that of a more traditional or ahead-of-time (AOT) compiler [HGH96, Inca, PHT+, vV]. These systems view the bytecodes as a source language, and produce native code as their output. The source is translated into some intermediate form, after which various analyses and optimizations are performed. All this is done offline, before the bytecode program is loaded for execution. The obvious advantage is that the compiler now has time to more fully analyze and optimize the bytecode stream. This has the potential to yield native code with better performance than code produced by a JIT compiler. Speedups of 5 [PHT+], 20 [HGH96], and even 50 [Inca] over direct interpretation have been reported; speedups on the order of 6 have been claimed over current JIT systems [Inca]. The disadvantage of ahead-of-time compilation is three-fold: (1) instant portability is now lost, (2) analysis information must be recomputed, and (3) recomputing from a lower-level representation is more difficult as compared to computing the same information at the source-level. Note that in some situations it is even impossible to recompute all information originally available at the source-level; for example, one cannot recompute algorithmic-level assertions inserted by a programmer and then lost in the translation.

Bytecode	**Native Code**
`aload a`	`load a, R0`
`iload i`	`load i, R1`
`iconst 2`	`move #2, R2`
`aload a`	`load a, R3`
`iload i`	`load i, R4`
	`// run-time checks performed by iaload`
	`if (R3 == NULL) throw NullPointerException;`
	`if (unsigned(R4) >= length(R3)) throw ArrayIndexOutOfBoundsException;`
	`lshift #2, R4, R4 // index*4 to access element`
`iaload`	`load (R3)R4, R3`
`imul`	`mult R2, R3, R2`
`aload b`	`load b, R3`
`iload i`	`load i, R4`
	`// run-time checks performed by iaload`
	`if (R3 == NULL) throw NullPointerException;`
	`if (unsigned(R4) >= length(R3)) throw ArrayIndexOutOfBoundsException;`
	`lshift #2, R4, R4 // index*4 to access element`
`iaload`	`load (R3)R4, R3`
`iadd`	`add R2, R3, R2`
	`// run-time checks performed by iastore`
	`if (R0 == NULL) throw NullPointerException;`
	`if (unsigned(R1) >= length(R0)) throw ArrayIndexOutOfBoundsException;`
	`lshift #2, R1, R1 // index*4 to access element`
`iastore`	`store R2, (R0)R1`

Figure 3: Postfix-based native code for 3-address, load/store RISC machine.

One ahead-of-time strategy is to map each JVM stack location to a machine register, and then generate native code using these registers instead of the stack [HGH96]; a similar approach maps each JVM stack location to a C variable, and then generates equivalent C code [PHT+]. For discussion purposes, we call this strategy a *postfix*-based approach to generating native code. For example, tracing the bytecodes shown in Figure 2, the reader will discover that at most 5 stack locations are needed. Assigning these locations to machine registers R0-R4, it is straightforward to generate the native code shown in Figure 3. The benefit is obvious: though the code contains the same number of conditionals as before (6), it requires only 11 memory loads (versus 22) and 1 store (versus 12). Note that it is possible to further improve upon this native code, e.g. by performing copy elimination and register renaming as discussed in [HGH96]. However, this approach is handicapped by its poor initial register allocation, which for example commonly assigns the same value to multiple registers (`i` is loaded into registers `R1` and `R4`) and places constants in registers (which hinders strength reduction such as replacing `mult` by `lshift`). In general the compiler must work significantly harder to undo the effects of this initial register allocation; in the presence of complicated control flow, this will likely lead to less effective optimization.

Bytecode	**Native Code**
`aload a`	`load a, R0`
`iload i`	`load i, R1`
`iconst 2`	
`aload a`	
`iload i`	
	`// run-time checks performed by iaload`
	`if (R0 == NULL) throw NullPointerException;`
	`if (unsigned(R1) >= length(R0)) throw ArrayIndexOutOfBoundsException;`
	`lshift #2, R1, R2 // index*4 to access element`
`iaload`	`load (R0)R2, R3`
`imul`	`lshift #1, R3, R3`
`aload b`	`load b, R4`
`iload i`	
	`// run-time checks performed by iaload`
	`if (R4 == NULL) throw NullPointerException;`
	`if (unsigned(R1) >= length(R4)) throw ArrayIndexOutOfBoundsException;`
`iaload`	`load (R4)R2, R4`
`iadd`	`add R3, R4, R3`
	`// No run-time checks needed`
`iastore`	`store R3, (R0)R2`

Figure 4: Infix-based native code for 3-address, load/store RISC machine.

A more general strategy adopted by other ahead-of-time compilers is to recover as much of the original source program as possible—its "infix" style and control flow—and then optimize [Inca]. The approach then reduces to one of optimizing a more traditional intermediate form, a problem that has been well-studied and with known techniques. For example, starting once again with the bytecodes of Figure 2, the native code shown in Figure 4 can be produced using this approach. The resulting code is what one normally expects from the compilation of our original line of Java source: a few memory loads and array calculations, some run-time checks, some arithmetic, and a store. Altogether, this final (optimal) code sequence requires 7 loads (2 due to run-time checks which cannot be safely eliminated in this example), 1 store, 4 conditionals, and no multiplies (replace with a shift). Note that 2 run-time checks can, and were, safely eliminated.

Yet another approach to cope with Java's lackluster speed performance is to use an entirely different intermediate representation. Various proposals include UNCOL [Con58], ANDF [BCD+89], and slim binaries [Fra94]. Though these approaches may yield better performance than plain bytecodes, none attempt to include high-level analysis information. Furthermore, each represents a radical change from the existing bytecode format, and is thus incompatible with existing JVMs.

3 Our Approach: Bytecode Annotation

Our approach is based on the general observation that compilers typically discard high-level information during the translation from source code to intermediate code [NHN96]. As a result, during the generation and optimization of the final machine code, the compiler has to not only (a) recompute information that was already available at the higher-level, but also (b) accept a potential loss of higher-level information due to time constraints (e.g. in JIT compilers) This leads to missed opportunities for optimization, as well as longer compilation times than is strictly necessary—a particularly critical problem in the context of Java as mentioned earlier.

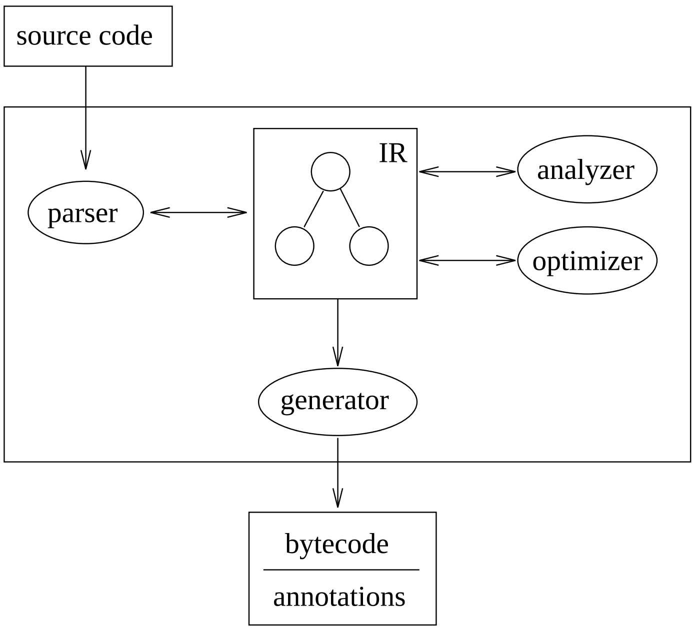

Figure 5: An annotation-generating compiler.

We propose a new approach for optimizing the Java bytecodes, based on the notion of annotations. This system is summarized by Figures 5 and 7. The information collected by the compiler during the original source to bytecode translation is encoded in the bytecode stream (Figure 5). This information may come from sophisticated analysis, as well as from programmer directives. For each method `foo` in the source program, the *attribute* $\$\$annotations$ is added to its list of attributes. Each bytecode of `foo` has a corresponding entry in the data portion of $\$\$annotations$; these entries denote our annotations. When the bytecode stream is eventually loaded for execution, the underlying JVM (which may or may not include a JIT compiler) can intelligently use the annotations in the translation process or silently ignore them if it does not recognize this new code attribute (e.g. Figure 7). The annotations attribute is input into the class file in the same way as the predefined Sun's JVM code attributes for debugging information are incorported [LY96].

Thus, our approach remains backward compatible with existing JVMs, while enabling annotation-aware JIT compilers to perform more sophisticated optimizations without expensive analysis. The result is very good performance without a loss of instant portability. The annotations are also useful in more traditional, ahead-of-time compilation systems, since their availability can reduce the need for expensive analysis while potentially providing information that cannot be recomputed. The obvious disadvantage is that the presence of annotations will lengthen the bytecode. Table 1 compares the annotated class file size against the original class file size, for the kernels of the benchmarks evaluated in Section 4. Under the current encoding scheme, annotations can result in a class file size increase varying from 33% to close 100%. It should be noted that our initial encoding scheme is very simplistic and easily optimized to reduce space.

Benchmarks	**Class File Size (bytes)**	**Annotations Overhead (bytes)**	**Annotated Class File Size (bytes)**	**Percentual Increase (%)**
`Neighbor`	368	267	635	73
`EM3D`	481	158	639	33
`Huffman`	706	550	1256	78
`Bitonic Sort`	970	945	1915	97

Table 1: Annotations overhead.

Another potential disadvantage concerns security, since incorrect or malicious annotations can cause erroneous native code to be generated. However, the *Digital Signature Initiative* recently proposed by the World Wide Web Consortium (along with similar schemes such as Microsoft's Digital Certificates) can be used to digitally sign and authenticate bytecodes and their annotations. This will help eliminate security problems resulting from malicious or untrusted code. One cannot of course eliminate all chance of error, since even existing JIT compilers can potentially generate incorrect code. In our future work we are also considering the study of formal methods for validating the annotations.

3.1 A Simple Example

Each annotation contains a concise summary of information necessary for generating optimized native code. Currently, this includes information about register allocation, run-time checking, and memory disambiguation. With a single pass and minimal bookkeeping (tracking physical register usage and assignments), an annotation-aware JIT compiler will be able to process the annotations and produce quality native code.

Bytecode	Src	Inter	Dest	Last Use	R T Check	Memory Ref Tag
aload a			v0			/stack/objref/a
iload i			v1			/stack/int/i
iconst 2						
aload a	v0		v0			/stack/objref/a
iload i	v1		v1			/stack/int/i
iaload	v0, v1	v2	v3		111	/heap/array/int/*
imul	2, v3		v3			
aload b			v4			/stack/objref/b
iload i	v1		v1			/stack/int/i
iaload	v4, v1	v2	v4		101	/heap/array/int/*
iadd	v3, v4		v3	v4		
iastore	v0, v1, v3	v2	v3	v3	000	/heap/array/int/*

Figure 6: Annotated Java bytecodes for `a[i]=(2*a[i])+b[i]`.

Consider once again our running example from Figure 2. In Figure 6 we show this same code, along with our annotations. The `src`, `inter`, `dest`, and `last use` columns denote a virtual register allocation performed during the original source to bytecode translation. Virtual register `v0` should be mapped to physical register `R0`, `v1` to `R1`, and so on until the number of physical registers is exhausted. The remaining virtual registers are then mapped to memory locations. The `inter` column informs the JIT compiler to save intermediate values in the specified register(s) if possible; in this example, `inter` is used to retain information from array index calculations. The `last use` column denotes when a register becomes dead, though it is also overloaded to denote when a store back to memory is necessary—e.g. `iastore` in Figure 6. In particular, if the `src` register that contains the value to be stored also appears in the `last use` column, then the store must be performed. Note that the virtual registers in the `src` column appear in the order in which they are pushed onto the stack in the bytecode; thus, given any bytecode instruction it is always possible to determine which `src` register contains which operand.

The `r-t check` column specifies which run-time checks should be performed. For array accesses, at most three possible checks are required:

1. is the array reference = Null?
2. is the array index < 0?
3. is the array index $>=$ the length of the array?

Each check is assigned a bit in `r-t check`; if the bit is 1, then code must be generated to do the check. Note once again that if both (2) and (3) are required, only a single unsigned compare is actually needed.

The last column, `memory ref tag`, provides memory reference information suitable for performing disambiguation. This in turn enables other forms of optimization such as instruction scheduling. Memory reference tags also carry type information for Java local variables and temporaries on the stack, which frees the underlying JVM from performing dataflow analysis to recover type information. Each bytecode that may yield native code containing a memory reference is annotated with a "memory reference tag." This tag denotes an imaginary path through the memory hierarchy of a machine, where provably different memory locations are assigned different paths. For example, in Figure 6 the array reference (i.e. pointer) `a` is given the tag `/stack/objref/a` since it is an object reference variable allocated on the stack, and is known to represent the unique memory location named "a". On the other hand, note that references to `a[i]` are tagged with `/heap/array/int/*`, since `a[i]` denotes any element in an unnamed heap-allocated integer array. The array is unnamed because heap-allocated objects can possibly alias one another in this example; recall the bytecodes are derived from the `foo` method of Section 1, which takes two array arguments that could in fact refer to the same array. Thus, references to `b[i]` are likewise tagged with `/heap/array/int/*`. Using these tags, it becomes trivial to disambiguate memory references; e.g. `a` and `i` clearly denote different memory locations, as do `a` and `b`. However, `a[i]` and `b[i]` may refer to the same memory location.

From these annotations, an annotation-aware JIT compiler can produce the same optimal native code shown earlier in Figure 4.

3.2 Compile-time Production and Run-time Usage

During the initial compilation from source language to bytecode stream, our compiler (see Figure 5) behaves like a traditional optimizing compiler: it builds an intermediate form, executes various analyses, applies numerous optimizations (such as common subexpression elimination and loop invariant code removal), performs a virtual register allocation, and then generates the bytecodes. However, instead of discarding analysis information once the bytecodes are generated, the compiler instead emits this information in the form of annotations. Each instruction thus has an associated annotation, stored in the method's *$$annotations* attribute.

At run-time, an annotation-aware JIT compiler can safely ignore the annotations, or process them in a single pass as it performs code generation. Enough information is supplied to assign physical registers and track their lifetimes; this is the only bookkeeping required of the run-time system. Virtual registers are mapped directly to physical registers, until the number of physical registers is exhausted; any remaining virtual registers are then mapped to memory locations. The only exception is that depending on the flexibility of the target machine's instruction set, it may be necessary for the the JIT compiler to reserve a physical register for scratch purposes (e.g. reloading spilled virtual register values).

4 Results

Our results revolve around four benchmarks: `Neighbor`, which performs a nearest-neighbor averaging across all elements of a two-dimensional array; `EM3D`, a code that creates a graph and then performs a 3D electromagnetic simulation [CDG+93]; `Huffman`, a character string compression and decompression application; and `Bitonic Sort`, which builds a binary tree and then performs bitonic sorting (recursively) [BN86].

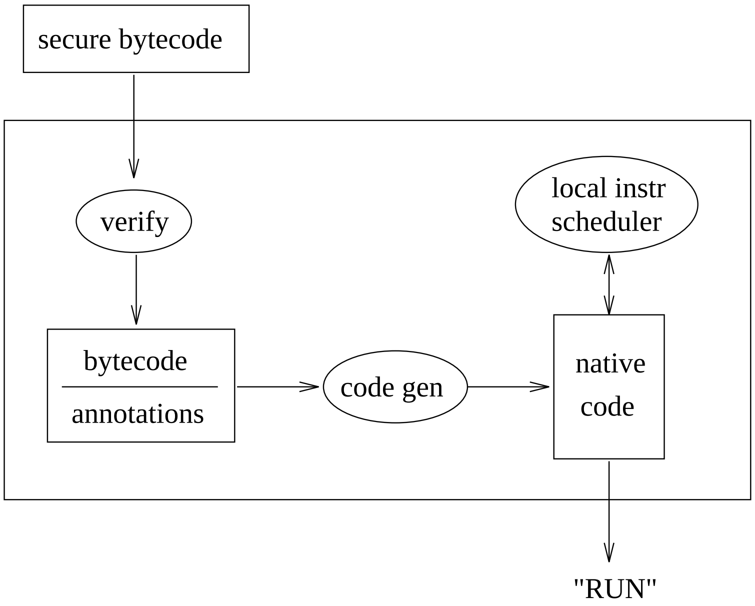

Figure 7: The AJIT (Annotation-aware JIT) system.

To measure the impact of our annotations, we collected results using a variety of tools from Sun, Microsoft, and Asymetrix. We compared these against a hand-simulated **AJIT** (Annotation-aware JIT) system, an overview of which is shown in Figure 7. All codes were run on a Pentium 200MHz 32MB PC (Windows 95).

4.1 Benchmark Timings

Table 2 presents the results for running each benchmark in a number of different environments. Note that the times do not include translation nor compile time; timings thus represent the quality of the generated code only. The codes were compiled using Sun's `javac` Java compiler, and then executed in a number of ways: Sun's `java` interpreter, Microsoft's `jview` JIT compiler, Asymetrix's SuperCede `sc10java` AOT compiler, our AJIT system, and finally the native code generated by compiling an unsafe C version using

Microsoft Visual C++. The latter is "unsafe" in the sense that C does not perform run-time checking, a feature required as part of the JVM specification. However, the C version does serve as a rough measure of the best possible case.

Benchmarks	**Interpreter**	**JIT**	**AOT**	**AJIT**	**C**
`Neighbor` `Array Size` = 256x256 `Iterations` = 1500	554	47	104	19	18
`EM3D` `Tree Size` = 1300 nodes `Iterations` = 200	468	60	32	32	28
`Huffman` `Array Size` = 30000 `Iterations` = 288	506	28	49	22	20
`Bitonic Sort` `Tree Size` = 4096 nodes `Iterations` = 512	500	23	21	20	20

Table 2: Benchmarks results, in seconds.

A summary of Table 2 is presented in Tables 3 and 4. JIT and AOT offer speedups ranging from 5.3 to 23.8 over direct interpretation, but on average are still a factor of 2.0 (JIT) and 2.9 (AOT) slower in comparison to C. Our annotation-based approach however offers speedups the same or better than both JIT and AOT, and even more importantly runs on average only 1.1 times slower than C. Note that these results are achieved without any loss in safety; all necessary run-time checks are still being performed.

A few notes. The execution times shown for `Bitonic sort` in Table 2 are roughly all the same; this is due to the heavily recursive nature of the code, in which subroutine calling overhead overwhelmed all other optimizations. The JIT system did quite well in all cases except `EM3D`; the difference is that `EM3D` is a pointer-based code, while `Neighbor` and `Huffman` are array-based. The AOT system performed just the opposite; it did well for `EM3D`, and worse than JIT for `Neighbor` and `Huffman`. Since the AOT system is running offline, it has the potential to do much better in all cases. Its uneven results are most likely due to an immature product.

4.2 The `Neighbor` Benchmark

To gain some insight into the power of the annotations, we will consider the `Neighbor` benchmark in more detail. The corresponding Java code is shown in Figure 8.

In the case of `Neighbor`, the annotations specified 10 virtual registers, 8 integer and 2 floating-point. Even though not all the virtual registers were physically available (only 5 of the Pentium's 8 integer registers

Benchmarks	Interpreter	JIT	AOT	AJIT	C
`Neighbor`	1	11.78	5.32	27.70	36.93
`EM3D`	1	7.80	14.63	14.63	16.71
`Huffman`	1	18.07	10.33	23	25.30
`Bitonic Sort`	1	21.74	23.81	25	25

Table 3: Speed-up over direct interpretation.

Benchmarks	Interpreter	JIT	AOT	AJIT	C
`Neighbor`	36.93	3.13	6.93	1.33	1
`EM3D`	16.72	2.14	1.15	1.15	1
`Huffman`	25.3	1.40	2.45	1.10	1
`Bitonic Sort`	25	1.15	1.05	1	1

Table 4: Slow-down in comparison to C.

```
public void neighbor (double a[][])
{
   int i, j;

   for (i = 1; i < (a.length - 1); i++)
      for (j = 1; j < (a[i].length - 1); j++)
         a[i][j] = ( a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1] ) /4;
}
```

Figure 8: The `Neighbor` benchmark.

Benchmarks	Interpreter	**AJIT** Reg. Alloc. All RT Checks w/o Instr. Sched.	**AJIT** Reg. Alloc. RT Checks w/o Instr. Sched.	**AJIT** Reg. Alloc. RT Checks with Instr. Sched.
`Neighbor`	554	26	20	19
`EM3D`	468	41	36	32
`Huffman`	506	24	22	22
`Bitonic Sort`	500	24	20	20

Table 5: Breakdown of each annotation's contribution; times in seconds.

are available for use within the JVM), the AJIT system was able to eliminate nearly half the bytecodes from the final native code. For example, all bytecode load instructions with identical `src` and `dest` columns denote register reuse, and thus can be skipped (as discussed in Section 3.1). Likewise, all writes to memory can be skipped if they are annotated such that the `src` register being stored does not appear in the `last use` column; e.g. in `Neighbor`, the index variables `i` and `j` never really need to be written to memory. Finally, the annotations also allowed the number of static run-time checks in the code to be greatly (and safely) reduced from 35 to 6, with none appearing in the inner loop. Dynamically, the total number of checks thus dropped from $O(N^2)$ to $O(N)$ for an N-by-N array. Note that it is not possible to eliminate all the run-time checks, since 2D arrays are implemented as an array of (references to) arrays, and the sub-arrays are not necessarily rectangular. Thus, each sub-array reference must be checked against `Null`, while the lengths of the `i-1` and `i+1` sub-arrays must also be compared to the length of sub-array `i`. Since Sun's `javac` compiler currently does very little source-level optimization, in order to be fair to the other tools we hand-optimized the `Neighbor` source code to cache references to sub-arrays.

An interesting aspect of the annotations is that they allow the compiler to generate explicit bytecodes to perform run-time checking, and then disable the implicit checks that otherwise form a part of every array access. For example, in `Neighbor` 34 of the 35 run-time checks are by default performed in the innermost loop. However, all these checks can in fact be hoisted out of the inner loop, which is profitable if there also exists a mechanism for disabling the run-time checks performed inside the loop; our annotations represent such a mechanism, one that can be efficiently applied by any JIT or AOT compiler. The analysis otherwise required to disable these checks can be prohibitively expensive in the case of JIT systems.

4.3 The Impact of Different Types of Annotations

Table 5 summarizes the contribution of each annotation on the overall efficiency of the native code produced by our AJIT system. The biggest impact by far is found in AJIT column 1, which shows the effect of using native code with good register allocation. The second AJIT column shows the impact of also eliminating unnecessary run-time checks; the speedups are less dramatic, but still significant (a reduction of 2 to 6 seconds in execution time). The final column summarizes the impact of our memory reference tags, an annotation that is used primarily to aid in aggressive instruction scheduling. This annotation adds little benefit, reducing execution time in only two cases (and even then by at most only 4 seconds). The reason is that the instruction scheduling being performed is only a local, basic-block form of scheduling. This is the best one can reasonably expect from a JIT system, which does not have the resources (time nor space) to devote to more aggressive techniques such as software pipelining. However, we know the memory reference tags are quite useful; for a simulated VLIW CPU, we achieved near 5-fold speedup on the same EM3D benchmark, due solely to aggressive instruction scheduling enabled by better memory reference information [NHN96].

5 Conclusions

The principal argument against the use of portable bytecodes is their lack of efficient execution. However, we have shown that efficiency does not need to suffer in relation to portability. Our annotation-based approach is the first to achieve C-like performance while requiring only JIT technology. Using our approach, one retains the advantage of instant portability of bytecode streams, remains backward compatible with existing JVMs, and enables highly-efficient bytecode execution on modern CPUs. Our approach can also benefit a more traditional AOT compilation system, since the annotations provide information that is expensive to recompute.

References

[BCD+89] M.E. Benitez, P. Chan, J.W. Davidson, A.M. Holler, S. Meloy, and V. Santhanam. ANDF: Finally an UNCOL after 30 years. Technical Report TR-91-05, University of Virginia, Department of Computer Science, Charlottesville, VA, March 1989.

[BN86] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical Report TR86-769, Cornell University, 1986.

[CDG+93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing 1993*, pages 262–273, November 1993.

[Con58] M. Conway. Proposal for an uncol (universal computer object language). *Communications of the ACM*, 1(10):5–8, October 1958.

[DS96] R. Dragan and L. Seltzer. Java speed trials. *PC Magazine*, October 22, 1996.

[Fra94] M. Franz. *Code-Generation On-The-Fly: A Key to Portable Software*. PhD thesis, ETH Zurich, February 1994. See http://www.ics.uci.edu/ franz.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.

[HGH96] C. Hsieh, J. Gyllenhaal, and W. Hwu. Java bytecode to native code translation: The caffeine prototype and preliminary results. *Proceedings of the 29th Annual Workshop on Microprogramming*, December 1996.

[Inca] Asymetrix Inc. The asymetrix supercede virtual machine. See http://www.asymetrix.com/nettools.

[Incb] Borland Inc. Jbuilder. See www.borland.com/jbuilder.

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

[Incd] Symantec Inc. Just in time compiler for windows 95/NT. See http://www.symantec.com/jit/index.html.

[Jon] J. Jones. Summary of java just-in-time compilers. Compiled from various postings on the web, see http://www.cen.uiuc.edu/ jjones/jit.html.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[NHN96] 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.

[PHT$^+$] T. Proebsting, J. Hartman, G. Townsend, P. Bridges, T. Newsham, and S. Watterson. Toba: A java-to-c translator. Part of the U. of Arizona's Sumatra project; see http://www.cs.arizona.edu/sumatra/toba.

[USCH92] D. Ungar, R. Smith, C. Chambers, and U. Holzle. Object, message, and performance: how they coexist in Self. *IEEE Computer*, 25(10):53–64, October 1992.

[vV] Hanpeter van Vliet. Mocha: The java decompiler. See http://web.inter.nl.net/users/H.P.van.Vliet/mocha.htm.

[Way96] P. Wayner. Sun gambles on Java chips. *BYTE*, November 1996.

[Wil] Tim Wilkinson. Kaffe: A free JIT virtual machine to run java code. See web.soi.city.ac.uk/homes/tim/kaffe/kaffe.htm.