Java as an Intermediate Language

Jonathan C. Hardwick[1] and Jay Sipelstein[2]

August 12, 1996

CMU-CS-96-161

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present our experiences in using Java as an intermediate language for the high-level programming language NESL. First, we describe the design and implementation of a system for translating VCODE—the current intermediate language used by NESL—into Java. Second, we evaluate this translation by comparing the performance of the original VCODE implementation with several variants of the Java implementation. The translator was easy to build, and the generated Java code achieves reasonable performance when using a just-in-time compiler. We conclude that Java is attractive both as a compilation target for rapid prototyping of new programming languages and as a means of improving the portability of existing programming languages.

[1]jch@cs.cmu.edu

[2]sipelstein@cs.cmu.edu

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Keywords: Java, NESL, VCODE, compiler, intermediate language, performance, prototyping.

1 Introduction

Intermediate languages are used by many modern compilers. Typically they are produced by a compiler's front end, which handles parsing and error checking for a particular high-level language, and are consumed by the back end, which handles code generation for a particular machine architecture. Intermediate languages simplify the inevitable process of porting a compiler to a new architecture by enabling the developer to re-use the front end of the compiler. Creating a new high-level language is also made easier if an existing intermediate language is used, because back ends can then be reused.

Choosing the right intermediate language is an important decision for language developers and compiler writers. Ideally, an intermediate language should be simple enough to serve as a good compiler target, while at the same time allowing for efficient execution on a range of platforms. Moreover, it should isolate low-level issues such as error checking and memory management. This paper was inspired by the observation that the recently developed Java[1] programming language [14] appears to possess all of these characteristics. We wanted to know whether Java would make a good intermediate language for current and future compilers.

Java has several attractions as an intermediate language. The first is the design of the language itself. By being strongly-typed, Java makes the compiler easier to debug, because many code generator bugs will trigger type errors during compilation of the resulting Java code. In contrast, if a weakly-typed intermediate language were used, these bugs might not be found until runtime. Java also provides garbage collection, thereby removing a large source of potential memory-management bugs in the generated code. In addition, if the language being implemented must itself provide garbage collection, the developer can simply push the responsibility down to Java.

Another attraction of Java is that it is a portable, network-aware language. The details of machine architecture, operating system, and display environment are all handled transparently by the Java *virtual machine* [15]. The same Java program can run on a Unix workstation, a PC, and a Macintosh, while retaining the same "look and feel" on each platform. Using Java as an intermediate language also allows programs to be distributed in an executable form (Java bytecode) over the Internet.

Finally, Java is a highly successful commercial product. This fact has huge advantages for a language developer. In particular, there is a whole industry devoted to porting Java to new platforms, improving Java compilers and run-time systems, writing libraries of Java objects, and fixing bugs. The developer would normally have to do all of these chores if a special-purpose intermediate language were used.

Given all these advantages, what might stop us from using Java as an intermediate language? There are three main questions to consider:

- Is Java easy to use in a new or existing system?
- Does Java provide sufficient functionality to model the features of the source language?
- Can the resulting programs be efficiently executed by a Java virtual machine?

To try to answer these questions, we built a system that translates VCODE [5] (a specialized intermediate language for the high-level parallel language NESL [4]) into Java, and performed a series of benchmarks to compare this new implementation with the original.

[1] Java is a trademark of Sun Microsystems, Inc. All other trademarks in this paper are the property of their respective owners.

The rest of this paper is organized as follows. Section 2 gives an overview of NESL, VCODE, and the current NESL system. Section 3 describes the translation of VCODE, its run-time system, and its libraries into Java. Section 4 discusses our experiences in building the system and outlines additional optimizations that we incorporated into the final version. Section 5 presents benchmark results, and Section 6 describes related projects. Finally, Section 7 summarizes the work and our conclusions.

2 The NESL System

Java [14] has been defined as "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic, buzzword-compliant, general-purpose programming language".[2]

In the same spirit of buzzword-compliance, NESL [3] is an interactive, high-level, strongly-typed, applicative, sequence-based, portable, nested data-parallel language. The primary data structure in NESL is the sequence, each element of which can itself be a sequence. Parallelism is expressed in NESL through an apply-to-each form over elements of sequences and through parallel operations on sequences.

The current NESL system consists of three layers, as shown in Figure 1 (see [7] for full details of the system). The front end of the system is an interactive compiler that lets users enter NESL expressions and programs. Every NESLexpression is first compiled into an intermediate language called VCODE [5]. The compiler then invokes a VCODE interpreter (either locally or on a remote machine), passes it the VCODE via `rcp` or a distributed filesystem, and reads back the results. The VCODE interpreter is the back end of the system; using VCODE as a portable intermediate language allows the user to execute the same code transparently on different machines, ranging from a Unix workstation to a parallel supercomputer.

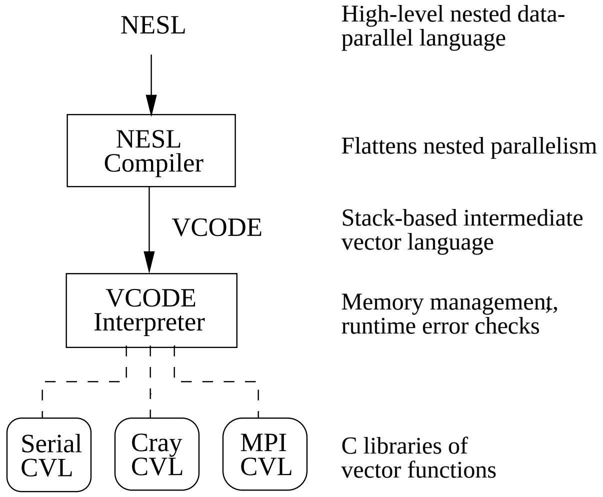

Figure 1: Components (boxes) and languages (lines) of the current NESL system. Solid lines represent translation, and dotted lines represent linkage to C libraries (rounded boxes).

The primary duties of the NESL compiler are implementing high-level aspects of the NESL language (such as type checking and the removal of higher-order code) and converting operations on

[2] Sadly, "buzzword-compliant" is now missing from `http://java.sun.com/`, although the full definition remains intact at several mirror sites.

arbitrarily nested sequences into operations on segmented vectors [2]. Although NESL was designed primarily to support efficient data-parallel programming, the high-level algorithmic nature of the language also makes it ideal for teaching and prototyping algorithms [4].

The middle layer of the system consists of the intermediate language VCODE and its interpreter. A VCODE program manipulates a stack of strongly-typed vectors. Each vector contains an arbitrary number of atomic values of a single type; VCODE vectors cannot be nested, unlike the NESL sequences they represent. The language provides a set of vector operations, stack manipulation instructions, and associated control and memory management instructions. The main tasks of the VCODE interpreter are managing the stack and vector memory efficiently, and implementing the vector operations via calls to CVL. The extra overhead of interpreting VCODE instructions, rather than executing a compiled version of them, is amortized over the length of the vectors on which they operate. Note that VCODE shares several properties with Java bytecode [10]: portability, strong typing, a stack-based execution model, and a design allowing for easy interpretation.

At the bottom of the system is CVL (C Vector Library), a machine-specific library that implements an abstract vector machine [6]. An example of a CVL function is `add_wuz`, which adds the corresponding elements of two integer vectors together and returns the results in a third vector. CVL is the only part of the system that must be rewritten for a new architecture [11].

3 Implementing VCODE in Java

To use Java as an intermediate language in an existing compiler, the current intermediate language (assuming that one exists) either can be totally replaced by Java or it can be translated into Java by an additional stage of the compilation process. The first approach entails rewriting the front end of the compiler to generate Java. The second approach requires no changes to existing parts of the compiler. Moreover, the additional translation stage should be simple to implement because the current intermediate language was probably designed for easy compilation. Finally, the front end has already performed most of the error checking, and has reduced complex high-level constructs of the source language into more primitive operations that are easier to map into Java.

However, the second approach will probably generate less efficient Java code than the first, because semantic information is lost. In particular, the code produced will be Java "in the style of" the existing intermediate language, and probably will not take full advantage of Java language features (in the same way that Fortran written in the style of Lisp is unlikely to be efficient). These observations are not specific to Java; they apply to any intermediate language.

We chose the second approach, favoring ease of implementation over the efficiency of the generated code (we investigate the final impact on performance in Section 5). The design of our system is shown in Figure 2. The NESL compiler is unchanged. Below it, the `vcodetojava` phase converts VCODE into Java. A standard Java compiler is then used to compile the Java, which consists mainly of calls to vector methods, into portable Java bytecode. This, together with a `VcodeEmulation` library that implements the vector methods and associated vector stack, can be executed by any Java virtual machine. Note that we have effectively replaced the VCODE interpreter with a Java virtual machine.

In the rest of this section we discuss the design and implementation of each of the major components of this system: the vector stack, the vector operations, and the `vcodetojava` program.

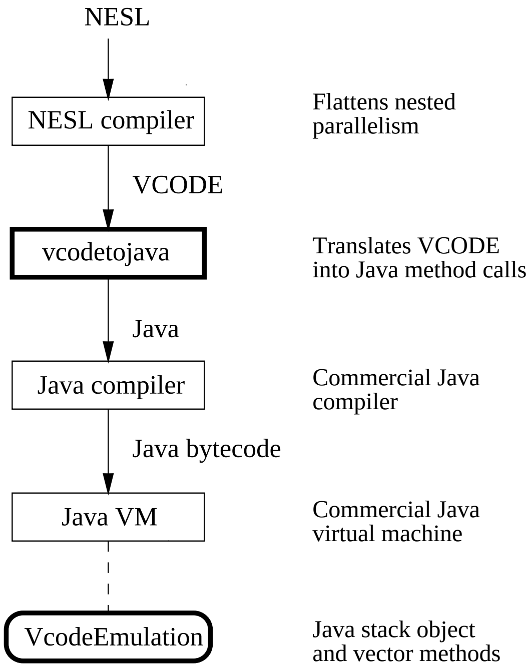

Figure 2: Components (boxes) and languages (lines) of the new NESL-to-Java system. Solid arrows represent translation, and dotted lines represent linkage to Java libraries. New components are shown with heavy borders.

3.1 Emulating the VCODE vector stack

The basic VCODE data structure is a vector of a primitive type. We represent this by a Java array: a first-class object with a length attribute. Java also supplies a vector class, `java.util.Vector`, which is essentially a dynamically-sized array capable of storing heterogeneous types. However, because VCODE vectors are both homogeneous and of a fixed length once created, there is no need for the extra generality and overhead of the `Vector` class. The VCODE types are mapped to Java types as shown in Table 1.

VCODE	Java	Bits
INT	int	32
FLOAT	double	64
BOOL	boolean	1
CHAR	char	16

Table 1: Mapping of VCODE types to their Java equivalents.

The VCODE interpreter allocates space for vectors when they are created, and uses reference counting to reclaim the space when they are no longer used. In Java, we create arrays corresponding to VCODE vectors with the `new()` method, and rely on them being reclaimed by the Java garbage collector when they are no longer used.

VCODE operations receive all their arguments, and return all their results, via the vector stack. In a typical implementation, the stack contains pointers to vectors rather than the vectors themselves. This approach permits fast stack operations (especially when popping, copying, or moving more than one element) and enables multiple copies of the same vector to be represented by identical pointers to the same piece of data. In Java, we achieve the same effect by storing references to arrays in an instance of the standard Java stack class, `java.util.Stack`.

3.2 Implementing VCODE vector operations in Java

VCODE provides over 130 vector operations. These operations typically have a direct mapping to functions provided by CVL. The VCODE interpreter runs a function-dispatch loop to execute programs: fetch the next VCODE operation, decode it, pop the appropriate number of arguments from the vector stack, call the matching CVL function, and push the result(s) back onto the stack.

For portability reasons we cannot rely on a machine-specific library such as CVL; all of the vector operations must be implemented as Java methods. However, the task of writing the methods is simplified because most of them fall into one of three major groups, with the code for operations in each group being very similar.

The vector methods are contained in the `VcodeEmulation` class; this also holds the stack where array references are stored. Intuitively, this class implements an abstract vector stack object and its associated vector operations. Just like their VCODE equivalents, the Java vector methods operate on the stack itself, popping their arguments and pushing their results. Figure 3 shows a Java method for the VCODE operation `+ INT`, which adds together two integer vectors.

```
void AddI () {
  int[] a = (int []) pop();                  // pop the argument array
  int[] b = (int []) pop();
  int[] dst = new int[a.length];             // create a result array
  for (int i = 0; i < a.length; i++) {       // loop over the elements...
    dst[i] = a[i] + b[i];                    // ...adding them together
  }
  push(dst);                                 // push the result onto the stack
}
```

Figure 3: Java method to implement the VCODE operation `+ INT`, which adds two integer vectors.

Note that this code assumes that the two argument arrays `a` and `b` have the same length. The `+ INT` operation makes the same assumption, but the VCODE interpreter can also check for vector length mismatches at runtime. In the VCODE-to-Java system, Java throws an exception if a runtime length mismatch causes the shorter array bound to be over-stepped. For full protection, we could extend the method to throw an exception immediately if the two lengths are not equal.

VCODE implements NESL's nesting of data structures efficiently by using *segmented vectors* [2]. Segmented vectors use two kinds of vectors to represent arbitrary sequence nesting: a normal non-nested vector to hold the data, and a series of specialized vectors (called *segment descriptors*) to describe how the data is subdivided. Many VCODE operations are defined only for segmented vectors, and require their arguments to have segment descriptors. We chose to represent a segment descriptor in Java as an array of integers holding the individual segment lengths. As a consequence, the Java implementation of a segmented operation is only slightly more complex than that of its unsegmented counterpart, with two nested loops iterating over the segments and the elements within each segment. Figure 4 shows a Java method for the VCODE operation `+_REDUCE FLOAT`, a segmented add-reduce (sum) that takes as arguments a segment descriptor and a floating-point data vector. The result is a vector of the sums of each of the segments.

```
final void AddReduceF () {
  int[] segd = (int []) pop ();              // pop the segment descriptor
  double[] src = (double []) pop ();         // and the source array
  double[] dst = new double[segd.length];    // create a result array
  int k = 0;
  for (int i = 0; i < segd.length; i++) {    // loop over the segments...
    double sum = 0.0;                        // ...initializing a sum of
    for (int j = 0; j < segd[i]; j++) {      // ...all values in a segment
      sum += src[k++];
    }
    dst[i] = sum;                            // ...and storing the sum
  }
  push (dst);                                // push the result
}
```

Figure 4: Java method to implement the segmented VCODE operation `+_REDUCE FLOAT`, which sums the individual segments within a floating-point vector.

3.3 Translating VCODE into Java

Rather than translating VCODE into Java and then compiling and running the resulting Java code, we could have implemented a VCODE interpreter in Java. There were two main reasons for rejecting this approach: simplicity and efficiency. Performing the translation is very easy, whereas writing a new interpreter would have been a much more ambitious undertaking, and adding an additional level of interpretation would inevitably slow down the resulting code.

The Java program generated by the translation process defines a single Java class of the same name as the original VCODE program. This class contains a Java method for each of the user-defined VCODE functions. A Java compiler is then used to produce Java bytecode from this source code. The bytecode is a portable executable representation of the program and can be run by any Java virtual machine, just as the original VCODE program can be run by any VCODE interpreter.

A simple Perl script called `vcodetojava` performs the translation, using an associative array to map most VCODE operations into the matching Java method calls. However, a few operations require extra processing. In particular, the function-defining operation `FUNC` is mapped to the opening of a new user-defined Java method, and the function-calling operation `CALL` is mapped to a Java call of the corresponding user-defined method. Both of these operations also require some name-demangling to ensure that VCODE function names remain legal in Java. The only other VCODE control flow operation is the `IF...ELSE` construct. This construct is mapped into a Java `if...else` block, where the `if` condition contains a method call that returns the boolean value on top of the vector stack.

Figure 5 shows an example of the translation process for a dot-product function, which multiplies the elements of two equal-length sequences together and returns the sum of the products. The NESL definition is a single line of code, and is compiled into a seven-line VCODE function. The VCODE function takes two pairs of segment descriptors and data vectors as input, corresponding to the `X` and `Y` sequences. The initial `POP` operation throws away a segment descriptor that is not needed by the following unsegmented elementwise multiplication `* FLOAT` (in general, `POP` n m means "pop n elements from a depth of m"). The `COPY` operation copies a segment descriptor to the top of the stack, where it is used by the plus-reduce (sum) operation. Finally, another `POP` discards the segment descriptor produced by the plus-reduce operation, and the function returns.

The Java method at the bottom of Figure 5 is generated by `vcodetojava`. As can be seen, the

NESL:	`function dotproduct (X, Y) = sum ({x * y: x in X; y in Y})`

VCODE:
```
FUNC DOTPRODUCT_47
POP 1 1
* FLOAT
COPY 1 1
+_REDUCE FLOAT
POP 1 1
RET
```

Java:
```
private static void DOTPRODUCT_47 () {
   s.Pop (1,1);
   s.MultF ();
   s.Copy (1,1);
   s.AddReduceF ();
   s.Pop (1,1);
}
```

Figure 5: NESL, VCODE, and Java representations of a dot-product function.

translation is very simple and can be applied on a line-by-line basis. Note that the Java method calls are being applied to the object `s`, which is an instance of the `VcodeEmulation` class.

The `vcodetojava` script consists of about 210 lines of Perl, of which 80 constitute the actual algorithm while the rest initialize the associative array. On a Sun SPARCstation 5/85 workstation, the script translates the 13,500 lines of VCODE corresponding to the NESL test suite in about 3.5 seconds. This time compares to about 19 seconds for the NESL compiler to generate the VCODE, and 71 seconds for Sun's portable compiler `javac` to compile the Java into bytecode. The times on a low-end PC are roughly comparable, although a faster native Java compiler can be used in place of Sun's portable compiler. For example, Microsoft's Visual J++ development environment takes just over 3 seconds to compile the same Java file on a DX4-120 system.

4 Pros and Cons of Java

It took one of the authors just over two days to complete a working prototype of the system, with his time divided roughly equally between implementing the Java stack model and translation script, and writing the vector methods. From this standpoint, the experiment was clearly a success: we quickly had a working system that enabled us to execute NESL code on any Java platform.

In terms of language features, Java's stack and array classes were a great help in rapid prototyping, and the language's built-in garbage collection meant that we did not have to adapt the reference-counting code used by the VCODE interpreter. We did not exploit Java's object-oriented features; there is no inheritance, and very few little composition of objects. In essence, we used Java as a portable dialect of C with garbage collection and a good collection of preexisting data structures.

Some aspects of Java slowed down both the development process and the generated code. Without templates, parametric polymorphism, or a built-in preprocessor, it is impossible to generate efficient type-specialized versions of the same basic method from within Java itself. Although useful for prototyping, the standard Java stack class is limiting in that it allows manipulation only of the element at the top of the stack, whereas VCODE requires the ability to operate on multiple elements at arbitrary positions in the stack. In terms of runtime performance, creating Java arrays is relatively expensive, because they are defined to be filled with the null value appropriate for their type. This requirement causes an implicit loop over the array, even though the initialization is unnecessary for arrays that will be written before being read. Finally, because Java is a young language, there is little performance data available for use in making informed design and optimization decisions.

However, it is easy to solve or work around most of these problems. We could have used an external preprocessor such as the Unix `m4` tool to generate multiple type-specialized versions of each vector method.[3] Microbenchmarks were used to establish the comparative cost of various Java operations; the results can be found at `http://www.cs.cmu.edu/~jch/java/optimization.html`. Profiling revealed that the awkward coding required to use the standard Java stack class was also a significant performance bottleneck. We therefore created a new version of the `VcodeEmulation` class that represents the stack as a directly-accessible array of objects. Each element of the stack holds an array corresponding to a VCODE vector, and the stack is grown as necessary; this is essentially a specialized version of the standard Java vector class. This modification improved the performance of the stack-intensive selection benchmark (see Section 5) by approximately 30%, and was adopted for the final version of the code.

5 Benchmarking the System

There are three performance characteristics of Java that we wish to understand: the effectiveness of just-in-time compilation, the cost of portability, and the overall system speed. We therefore benchmarked four different implementations of VCODE:

1. *JDK:* The Java interpreter from Sun's Java Development Kit (JDK), using the `VcodeEmulation` class described in Section 3.

2. *JIT:* A just-in-time (JIT) Java compiler, also using the `VcodeEmulation` class. The just-in-time compiler compiles Java bytecodes into machine code as it interprets them for the first time, and then stores the machine code for future reuse.

3. *Native:* The JDK interpreter, using the `VcodeNative` class. This class is a replacement for the `VcodeEmulation` class, and uses native C functions similar to those in CVL to implement the vector methods. However, it still uses Java for the vector stack code and for array allocation, because we want these objects to be accessible to Java's garbage collector.

4. *Vinterp:* The existing VCODE interpreter `vinterp`, written in C and linked against a serial version of the CVL library. This combination has been tuned for asymptotic performance on large vectors, with hand-unrolled loops and a memory management mechanism designed specifically for VCODE.

Comparing the performance of the JDK interpreter and JIT compiler implementations helps us understand the effectiveness of JIT compiler technology. Comparing the JDK and JIT implementations with the native-methods implementation, which uses compiled C code, lets us study the

[3] In the same way, many CVL implementations use function-defining macros to generate typed versions of an untyped function body.

price of portability. Finally, benchmarking the existing VCODE interpreter enables full end-to-end performance comparisons.

We used three different NESL benchmarks to compare the different implementations of VCODE:

- Least-squares line-fit. This function finds the best fit to a sequence of points. It is simple straight-line code with no conditionals or loops.
- Selection (generalized median-finding). This function uses a recursive randomized algorithm to find the element in a vector that would be at a specified position if the vector were sorted.
- Sparse matrix-vector multiplication. This function multiplies a sparse matrix stored in compressed row format by a dense vector, using a nested data-parallel algorithm.

We give the source code and test data for the benchmarks in Appendix A. Timings for supercomputer platforms have previously been reported [7, 11]. All three benchmarks have asymptotic running times that are linear in the size of the problem.

5.1 Methodology

To try to expose any performance effects that could be due to machine architecture rather than to the code being tested, we used two different machines for benchmarking: a Sun SPARCstation 5/85 with an 85 MHz MicroSPARC2 processor and 32 MB of RAM, running Solaris 2.4; and a PC with a 120 MHz AMD 486 processor and 24 MB of RAM, running Windows 95. For compilation we normally used the GNU C Compiler 2.7.0 (`gcc -O2`) and the Java compiler from Sun's Java Development Kit (JDK) 1.0.2 (`javac -O`). However, for the PC native code benchmarks we used a third-party port of JDK 1.0.1, running on Linux 1.2.13. For the just-in-time compiler on the PC we used the JIT in Microsoft Internet Explorer 3.0 Beta 2. We also tried Netscape 3.0b5, but its JIT proved to have a much higher overhead and a slightly higher per-element cost on all of the benchmarks; Internet Explorer was 18–38% faster for the largest benchmark sizes. No JIT compiler was available for the SPARCstation.

We compiled the NESL source code into VCODE using version 3.1 of the NESL compiler [3], combined with an additional optimization phase that inlines VCODE functions and removes unnecessary stack operations. All benchmarks were performed on idle machines to minimize outside effects. This was particularly important for the Java benchmarks, because Java provides only a time-of-day clock (`java.lang.System.currentTimeMillis()`), rather than a per-process timer. The poor resolution of the PC clock also created problems. To obtain accurate timings of the benchmarks at small problem sizes, we timed multiple iterations of each benchmark, adjusting iteration counts so that each run took at least a second.

We ran the Java virtual machines with their default heap sizes, which resulted in some garbage collection taking place for all but the smallest of runs. To reduce these nondeterministic memory effects, we forced a Java garbage collection before the beginning of each timing run. This reduced the variance but did not eliminate it.

5.2 Results

We timed each of the benchmarks at problem sizes ranging from 2^1 to 2^{17} (131072). Table 2 gives timings for selected problem sizes, averaged over five runs and rounded to two significant figures. We'll analyze the results for the line-fit benchmark in depth, and then briefly discuss the results for the selection and sparse matrix-vector multiplication benchmarks.

Problem size	120 MHz 486 JDK	JIT	Native	Vinterp	SPARCstation 5/85 JDK	Native	Vinterp
				Line-Fit			
16	9.1	2.6	5.8	0.73	9.5	7.0	0.62
128	17	4.3	7.1	1.3	16	8.0	0.85
1024	84	14	16	6.6	70	15	5.4
8192	580	99	90	50	500	71	40
65536	4800	800	570	460	4000	480	310
				Selection			
16	14	5.4	8.8	1.3	15	11	1.3
128	25	8.2	14	2.1	25	16	2.0
1024	74	18	25	4.7	70	29	4.3
8192	350	52	48	19	320	52	21
65536	2600	280	200	130	2300	190	160
				Sparse matrix-vector multiplication			
16	2.0	0.62	1.2	0.17	2.0	1.4	0.14
128	3.9	1.1	1.4	0.28	4.1	1.8	0.20
1024	18	3.9	2.9	1.4	17	2.9	0.95
8192	135	25	15	12	120	13	8.2
65536	1100	200	120	110	940	97	68

Table 2: Running times in milliseconds for three NESL benchmarks using different intermediate language implementations on a 120 MHz 486 PC and a SPARCstation 5/85.

5.2.1 Line-fit Benchmark

Figure 6 shows the performance (in elements per second) achieved by the four implementations on the line-fit benchmark, with the minimum, average, and maximum times plotted for each point. The error bars are only noticeable for the JIT and native code implementations at large problem sizes, where there appeared to be significant variations in the time taken by garbage collection during a run. We treat the current VCODE interpreter (`vinterp`) implementation as the base case, and discuss the results for each of the Java implementations (native, JIT, and JDK) in turn, concentrating on the results for the PC platform.

The VCODE interpreter is clearly the fastest of the four implementations of VCODE tested here, as we would expect for a special-purpose implementation. Its relative performance compared to the other implementations is best at small vector sizes, and its absolute performance falls off after about 16,000 elements, when the PC's 256 kB L2 cache can no longer hold two double-precision floating-point vectors.

The native methods implementation approaches the performance of the VCODE interpreter for large problem sizes, but does not quite reach it. There are two probable causes for this performance difference. First, the VCODE interpreter is linked to a CVL library whose loops have been unrolled by hand, reducing the loop overhead. Second, the Java requirement that every element of an array be initialized when the array is allocated causes an extra loop over the data that CVL does not need to perform. Note that there is no clearly observable cache effect for the native methods implementation, possibly because it is masked by the additional memory activity of the Java interpreter.

The JIT compiler achieves approximately half the performance of the VCODE interpreter for problems bigger than about 1,000 elements. The additional slowdown compared to the native methods implementation is probably due to the requirement that every array operation in Java

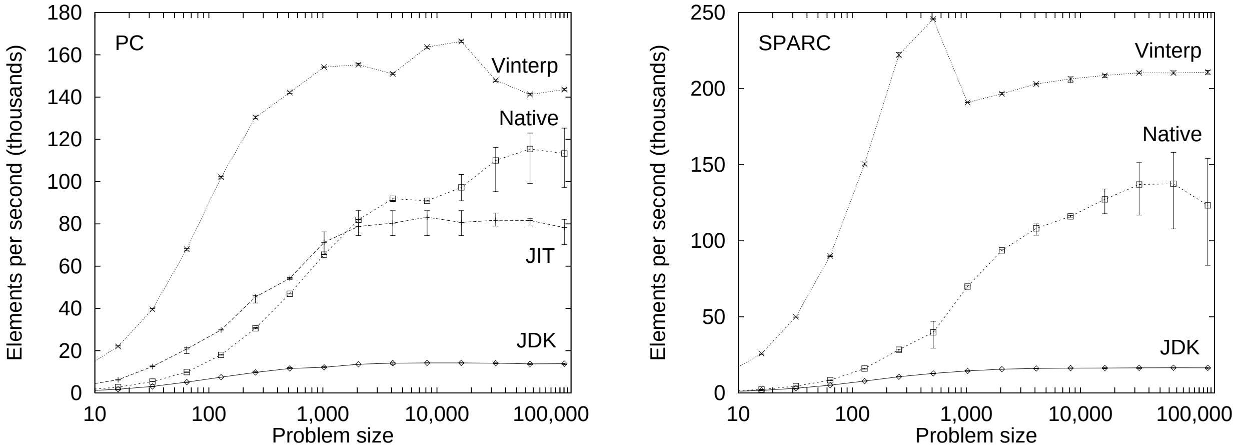

Figure 6: Performance of Nesl line-fit benchmark using different intermediate language implementations on a 120 MHz 486 PC (left) and a SPARCstation 5/85 (right).

must check for valid indices. The JIT compiler must therefore generate extra conditionals in the inner loop of vector code. There are techniques for guaranteeing valid indices without requiring these extra conditionals, such as performing loop-bounds analysis or exploiting virtual memory mechanisms for protection purposes [1], but to our knowledge these optimizations are not performed by any current JIT compiler. Note that we were unable to measure any extra compilation overhead incurred by the JIT compiler; this null result can probably be attributed entirely to the poor resolution of the PC clock.

Finally, the JDK interpreter achieves about one tenth the performance of the Vcode interpreter, and is four to six times slower than the JIT compiler.

As well as graphing the results of the line-fit benchmark, we can also use them to calculate the constant overhead and the asymptotic time per element of each implementation. This is only possible because the benchmark executes a fixed number of Vcode operations—and hence should have a fixed interpretive overhead—for all problem sizes. The resulting figures for the constant overhead and the time per element are shown in Table 3.

	120 MHz 486 PC		SPARCstation 5/85	
Implementation	Overhead	Per-elt.	Overhead	Per-elt.
JDK	8200	72	8500	60
JIT	2200	12	*N/A*	*N/A*
Native	5700	8.6	6900	7.1
Vinterp	690	7.1	610	4.7

Table 3: Constant overhead and asymptotic time per element (in microseconds) for Nesl line-fit benchmark using different intermediate language implementations on a PC and a SPARCstation.

Using the overhead and the asymptotic time per element we can now understand why the performance curves for the JIT compiler and the native methods implementation cross. The JDK interpreter used by the native methods implementation has a higher constant overhead than the JIT compiler, and this overhead dominates performance for small problem sizes. However, for large problem sizes the per-element speed advantage of the native vector methods outweighs the higher overhead of the JDK interpreter, and the native methods implementation is faster.

We can also calculate the percentage of the total running time due to the constant overhead, as shown in Figure 7. The disparity between the speed of native methods and the speed of the Java interpreter is reflected in the top curve; because the native methods are much faster than the interpreter, the impact of the fixed interpretive overhead is bigger. The JDK and JIT compiler implementations have similar results to that of the VCODE interpreter, indicating that the ratios of their interpretation and execution costs are comparable, and the problem size at which they achieve half of their peak performance ($n_{1/2}$) is similar. Put another way, the performance difference between the specialized VCODE interpreter and the general-purpose Java interpreter and JIT compiler is roughly the same as the performance difference between the machine-specific C code in CVL and the Java vector methods in `VcodeEmulation` being executed by the Java interpreter and JIT compiler.

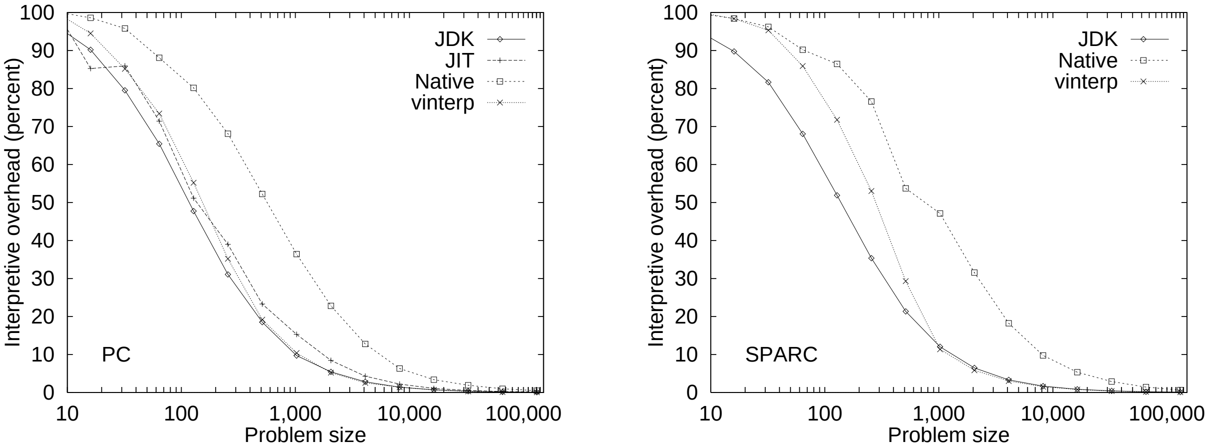

Figure 7: Percentage of interpretive overhead in NESL line-fit benchmark for different intermediate language implementations on a 120 MHz 486 PC (left) and a SPARCstation 5/85 (right).

The shapes of the performance curves on the SPARCstation are generally similar to those on the PC, although the cache effect for the VCODE interpreter is much more pronounced and happens at around 500 elements, due to the SPARCstation's much smaller cache. This general similarity of the results for the two platforms is true for all three benchmarks, suggesting that there are no architecture-dependent effects skewing the results. The two platforms are also comparable in terms of absolute speed.

5.2.2 Selection Benchmark

Figure 8 shows the performance achieved for the selection benchmark. The ordering of the results is the same as for the line-fit benchmark: for small problem sizes, the VCODE interpreter is fastest, followed by the JIT compiler, native methods, and finally the JDK; for large problem sizes, the ordering of the JIT compiler and the native methods implementation is reversed. However, the shapes of the curves are different than for the line-fit benchmark, reflecting the fact that the selection benchmark spends less time in straight-line code than line-fit, and places more emphasis on recursion and dynamic memory use. In particular, the performance curves of the JIT compiler and the native methods implementation cross at a larger problem size, because there is more work going on in the Java virtual machine (where the JIT compiler is faster) and less in the vector methods (where the native methods are faster).

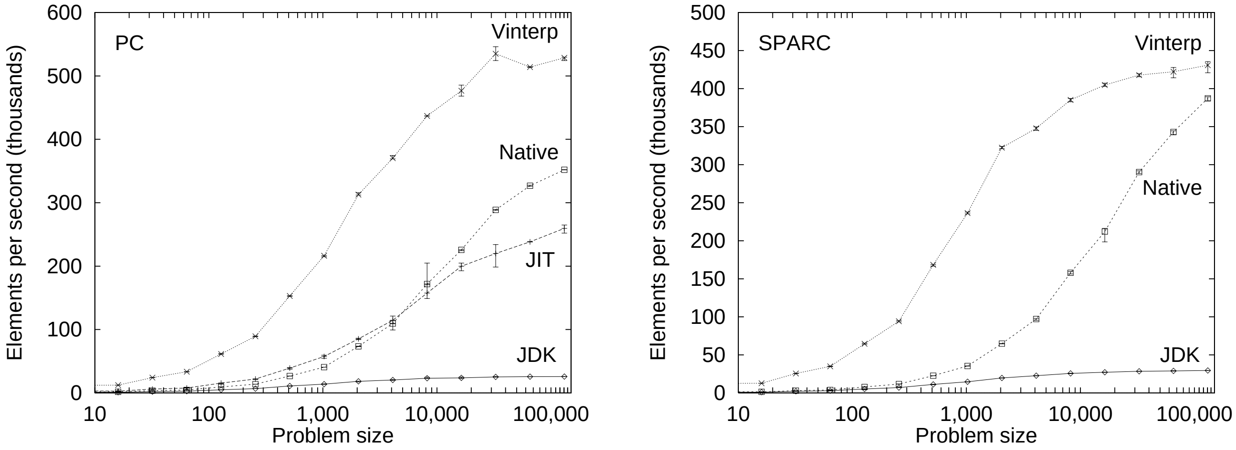

Figure 8: Performance of Nesl selection benchmark using different intermediate language implementations on a 120 MHz 486 PC (left) and a SPARCstation 5/85 (right).

5.2.3 Sparse matrix-vector multiplication benchmark

Figure 9 shows the performance achieved on the final benchmark, sparse matrix-vector multiplication. The ordering of the results is the same as for the previous two benchmarks. Even though this is a nested data-parallel algorithm that uses segmented Vcode operations, the shapes of the graphs and the performance ratios are similar to those for the non-nested line-fit benchmark, which uses mostly unsegmented operations. Note that there is less variance in the results than for line-fit because sparse-matrix vector multiplication uses fewer temporary vectors, and hence less garbage collection occurs.

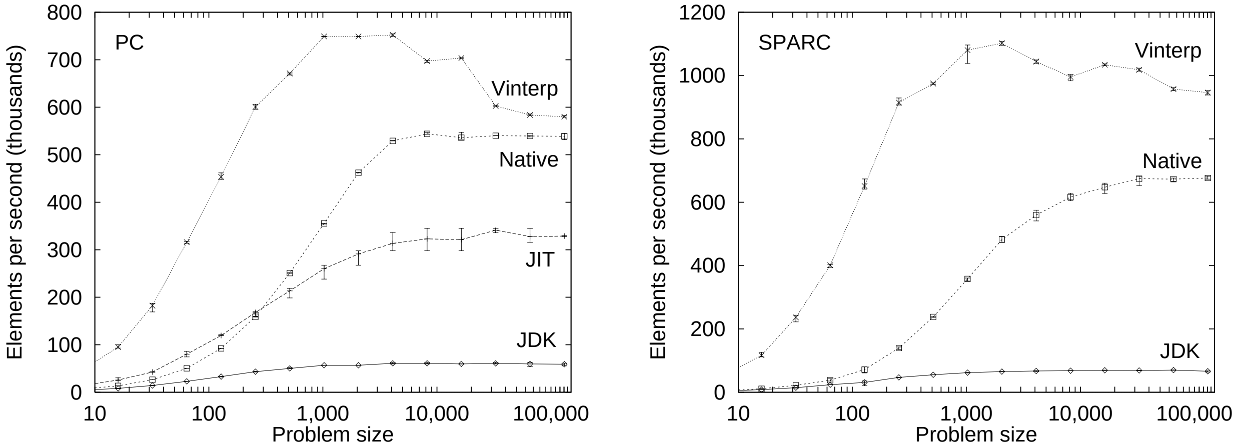

Figure 9: Performance of Nesl sparse matrix-vector multiplication benchmark using different intermediate language implementations on a 120 MHz 486 (left) and a SPARCstation 5/85 (right).

5.3 Memory Usage

The space efficiency of an intermediate language is often equally as important as its time efficiency. The Java `VcodeEmulation` class and the existing Cvl implementation use essentially the same data types, and so their memory usage per vector is similar. For example, a Java integer array of length n occupies $4n + 16$ bytes in the Sun JDK, compared to $4n$ bytes in a typical C implementation. However, the dynamic memory usage of the VCODE and Java interpreters differ. The VCODE interpreter is optimized for the case of a few big objects (vectors), whereas Java's general-purpose memory allocation mechanism is optimized for many small objects. In particular, the VCODE interpreter uses reference counting to determine when a vector is no longer used, and hence can reclaim its space immediately. The interpreter has to halt and compress vector memory only when it can no longer find a free fragment large enough to satisfy a request. By comparison, current Java virtual machines typically perform garbage collection only when the system is idle, when there is no longer enough free memory, or on demand.

As an example, the VCODE interpreter requires just over 1.75 MB of heap to run the line-fit benchmark on input vectors of length 2^{16} (65536) without performing memory compaction. A double-precision floating-point vector of this length requires 0.5 MB of memory, so the VCODE interpreter is storing at most three of these vectors at any one time. The JDK Java interpreter using the original `VcodeEmulation` class requires 8.5 MB of heap to run the same benchmark without triggering a garbage collection, because it cannot implicitly reclaim the space used by the temporary vectors that the benchmark generates. We therefore extended `VcodeEmulation` to reuse the last vector popped from the top of the stack whenever possible (i.e., whenever the vector is of the right length and type to be used for a result). This modification reduces the minimum Java heap size required to run the benchmark without garbage collection to 3.5 MB, and all benchmark results are for this modified version of `VcodeEmulation`. A full reference-counting algorithm similar to that employed by the VCODE interpreter would probably reduce the memory usage still further. This would effectively be implementing a second level of garbage collection specialized for our particular language, which has rather unusual memory usage characteristics.

6 Related Work

Several other projects are also using Java to implement high-level languages, encompassing a wide range of design approaches. Perhaps the highest level is Libero [12], which compiles a program expressed in the form of a finite state machine into one of a variety of target languages, including Java. NetRexx [9] is a dialect of the Rexx language that also compiles to Java. Both of these projects take an approach similar to that described in this paper, in that Java source code is generated. More directly, Intermetrics Inc. have adapted their Ada 95 compiler [16] to generate Java bytecode, dispensing with the intermediate step of using a Java compiler such as Sun's `javac`. The Kawa Scheme-in-Java compiler [8] also generates Java bytecode, enabling the compiler to perform tail-recursion elimination using the GOTO bytecode instruction (there is no corresponding goto statement in the Java language). Finally, HotTEA [13] implements a simple Basic interpreter on top of the Java interpreter, without requiring any compilation at all.

7 Conclusions

Ideally, an intermediate language should be simple, portable, efficient, and (when possible) maintained by somebody else. In this paper we have investigated whether Java makes a good intermediate language. Specifically, we have described the design, implementation, and benchmarking of a system that uses Java as an intermediate language for the high-level parallel language NESL. Java proved to be very easy to use, as demonstrated by the completion of the prototype in a weekend, and had enough functionality to allow a clean implementation of the system. After additional tuning to improve the speed and space efficiency of the generated code, a just-in-time Java compiler achieved a performance between two and four times slower than that of the existing implementation of NESL (which uses hand-tuned C code) on a set of vector algorithm benchmarks. This performance gap is likely to narrow as just-in-time compilation technology improves. We conclude that Java is a strong candidate for use as an intermediate language for rapid prototyping of new high-level languages, and for increasing the portability of existing languages.

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. Technical Report CMU-CS-95-204, School of Computer Science, Carnegie Mellon University, October 1995.

[2] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[3] Guy E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, July 1995.

[4] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[5] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings Frontiers of Massively Parallel Computation*, pages 471–480, October 1990.

[6] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[7] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, San Diego, May 1993.

[8] Per Bothner and R. Alexander Milowsk. The Kawa Scheme interpreter project. `http://www.winternet.com/~sgml/kawa/`.

[9] Mike Cowlishaw. NetRexx. `http://www.ibm.com/Technology/NetRexx/`.

[10] James Gosling. Java intermediate bytecodes. *SIGPLAN Notices*, March 1995. Originally appeared at ACM SIGPLAN Workshop on Intermediate Representations (IR'95).

[11] Jonathan C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 68–77, October 1994.

[12] iMatix. Libero—the thinking programmer's tool. `http://www.imatix.com/html/libero/`.

[13] Mike Lehman. HotTEA. `http://www.mbay.net/~cereus7/HotTEA.html`.

[14] Sun Microsystems. *The Java Language: An Overview*, 1995. `http://java.sun.com/doc/Overviews/java/`.

[15] Sun Microsystems. *The Java Virtual Machine Specification*, August 1995. http://java.sun.com/doc/language_vm_specification.html.

[16] S. Tucker Taft. Programming the Internet in Ada 95. Submitted to Ada Europe '96. Also available at http://www.inmet.com/~stt/adajava_paper/.

Copies of [3], [4], [5], [6], [7], and [11] are also available from the Scandal project web site at http://www.cs.cmu.edu/~scandal/papers.html.

A Benchmark Code

This section contains the NESL source code for the line-fit, selection and sparse matrix-vector multiplication routines, and also describes the test data used for the benchmarks.

A.1 Line-fit

```
function linefit(x, y) =
let
  n    = float(#x);
  xa   = sum(x)/n;
  ya   = sum(y)/n;
  Stt  = sum({(x - xa)^2: x});
  b    = sum({(x - xa) * y: x; y}) / Stt;
  a    = ya - xa*b;
  chi2 = sum({(y - a - b * x)^2: x; y});
  siga = sqrt((1.0 / n + xa^2 / Stt)* chi2 / n);
  sigb = sqrt((1.0 / Stt) * chi2 / n)
in
  (a, b, siga, sigb);
```

For the line-fit benchmarks, x and y were both copies of the floating-point index vector $[0.0, 1.0, 2.0, \ldots]$.

A.2 Selection

```
function select_kth(s, k) =
let pivot = s[#s/2];
    les = {e in s | e < pivot}
in
  if (k < #les) then
    select_kth(les, k)
  else
    let grt = {e in s | e > pivot}
    in if (k >= #s - #grt) then
         select_kth(grt, k - (#s - #grt))
       else pivot;
```

For the selection benchmarks, s was the integer index vector $[0, 1, 2, \ldots]$, and k was one third of the length of s.

A.3 Sparse matrix-vector multiplication

```
function nest(p, mlen) =
  let vector(seg,vals) = p;
      (seg1,seg2) = mlen
  in vector(seg1,vector(seg2,vals));

function MxV(Mval, Midx, Mlen, Vect) =
let v = Vect -> Midx;
    p = {a * b: a in Mval; b in v}
in
  {sum(row) : row in nest(p, Mlen)};
```

For the sparse matrix-vector multiplication benchmarks, every row in the matrix had a length of 5 and the matrix values were random floating-point data.