Interactive Simulations on the Web: Compiling Nesl into Java

Jonathan C. Hardwick[*] jch@cs.cmu.edu

Girija Narlikar[†] girija@cs.cmu.edu

Jay Sipelstein[‡] sipelste@susq.com

Abstract

We motivate and describe the design and implementation of a system for compiling the high-level programming language Nesl into Java. As well as increasing the portability of Nesl, this system has enabled us to make existing simulations and algorithm animations available in applet form on the web. We present performance results showing that current Java virtual machines running the generated code achieve about half the performance of a native implementation of Nesl. We conclude that the use of Java as an intermediate language is a viable way to improve the portability of existing high-level programming languages for scientific simulation and computation.

1 Introduction

Java [1] has several attractions as a language for scientific computation and simulation. Its implementation as a secure virtual machine with a portable windowing model enables graphical simulations to run safely on client computers. Java's object orientation, strong typing, garbage collection, and predefined libraries all ease the implementation of new applications. Finally, its rapid growth in popularity has resulted in an explosion of APIs, libraries, and development tools.

However, there are many legacy applications that would be tedious to convert by hand. Moreover, certain application areas (for example, rapid prototyping of parallel algorithms) are better suited to special-purpose high-level languages than to Java's more general and low-level model. In this paper we describe a system to translate programs written in Nesl [7], a high-level parallel programming language, into Java. Using this system we have translated several algorithm simulations—which previously ran only on X11-based Unix systems—into applets that are now available to anyone with a Java-capable browser. Note that we do not currently exploit Nesl's parallelism in the generated Java code, although it should be straightforward to extend our system to use Java threads on an SMP [2], or to use one of the parallel Java approaches discussed in [9]. In this paper, we discuss the design decisions and tradeoffs made in the development of the system, and we present a series of benchmarks to evaluate its performance. We also discuss two qualitative aspects of using Java as an intermediate language in this way: is it easy to incorporate into new or existing systems, and does it provide sufficient functionality to model the features of the source language?

The rest of this paper is organized as follows. Section 2 gives an overview of the Nesl language and system. Section 3 describes the translation of Nesl into Java. Section 4 contains examples of Nesl simulations that have been compiled into Java applets. Section 5 presents benchmark results. Section 6 discusses our

[*]CMU School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15232.

[†]CMU School of Computer Science, 5000 Forbes Avenue, Pittsburgh, PA 15232.

[‡]Susquehanna Investment Group, 401 City Ave, Suite 220, Bala Cynwyd PA 19004.

experiences in building the system. Finally, Section 7 describes related projects, and Section 8 summarizes the work and our conclusions.

2 The NESL System

NESL [4] is a portable, high-level, functional, nested data-parallel language. The primary data structure in NESL is the sequence, each element of which can itself be a sequence. NESL code closely resembles high-level pseudocode, and places more responsibility on the compiler and runtime system for achieving good efficiency [7]. Therefore, the language is well-suited for the concise and simple expression of parallel algorithms. For example, quicksort can be written in 10 lines of NESL, compared to 1700 lines of C with MPI (which includes code for explicit load-balancing). Although NESL was designed to explore support for nested data-parallelism, those issues are orthogonal to the point of this paper and are not discussed here.

The standard NESL system consists of three layers, as shown in Figure 1 (see [7] for full details). The interactive front end of the system compiles NESL programs into a machine-independent intermediate language called VCODE [5]. The front end then invokes a VCODE interpreter, which in turn calls the low-level CVL vector library [6]. The primary advantage of using an interpreted intermediate language for NESL is that it allows users to switch transparently between running their programs on workstations (for development) and supercomputers (for performance).

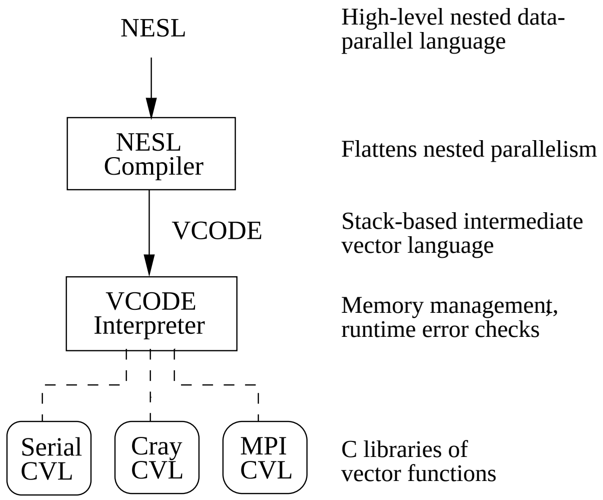


Figure 1: Components (boxes) and languages (solid lines) of the current NESL system. Arrows represent translation, and dashed lines represent linkage to C libraries (rounded boxes).

A VCODE program manipulates a stack of strongly typed vectors. The VCODE language provides a large number of stack-based vector operations, as well as instructions for stack manipulation, program control, memory management, and input/output. Note that VCODE shares several properties with Java bytecode [10], including portability, strong typing, a stack-based execution model, and a design that allows easy interpretation. The interpreter's main tasks are to manage the stack and vector memory, and to implement the vector operations via calls to CVL (C Vector Library), a machine-specific library that implements an abstract vector machine [6].

3 Compiling NESL into Java

We considered three different ways to improve NESL's portability using Java: write a VCODE interpreter in Java, rewrite the NESL compiler so that it generates Java, or write a translator from VCODE into Java. The first approach would impose an additional layer of interpretive overhead. The second would provide the greatest opportunity for optimizations (for example, by replacing VCODE's stack model with a vector register model), but would also require considerable effort. We took the third approach, choosing ease of implementation over efficiency of the generated code.

The design of our system is shown in Figure 2. The NESL compiler is unchanged. Below it, a new `vcodetojava` phase converts VCODE into Java source code plus calls to a VCODE emulation library written in Java. A standard Java compiler compiles this into bytecode, which can then be executed by any Java virtual machine. Note that we have effectively replaced the VCODE interpreter with a Java virtual machine.

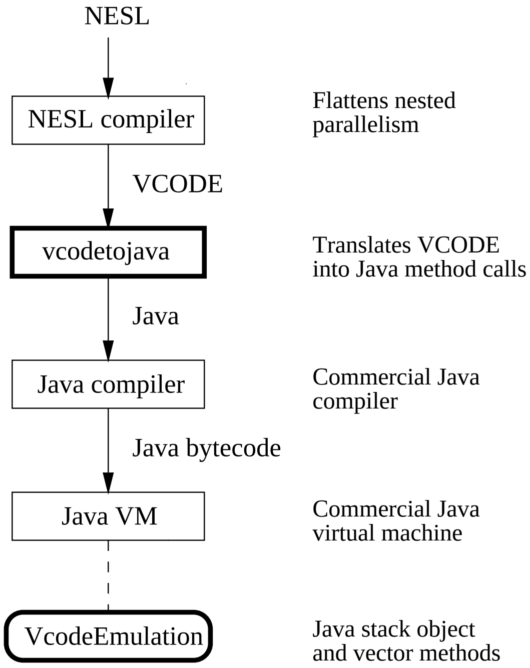


Figure 2: Components (boxes) and languages (solid lines) of the NESL-to-Java system. Solid arrows represent translation, and dashed lines represent linkage to Java libraries. New components are shown in bold.

3.1 Implementing VCODE Data Structures and Operations

The only VCODE data structure is a vector of primitive type. We represent vectors as Java arrays, which are first-class objects with a length attribute. The VCODE types (integers, double-precision floating-point numbers, booleans and characters) are mapped to their obvious Java equivalents. The VCODE interpreter allocates space for vectors from a heap and uses reference counting for garbage collection. In Java, we create arrays using `new()` and rely on the Java garbage collector to reclaim them 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.

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. In Java, the VCODE vector operations are implemented as methods of a `VcodeEmulation` class. This class implements an abstract vector-stack object with a stack of references to vectors (Java arrays). Just like their VCODE equivalents, the Java vector methods pop arguments from the stack and push results onto the stack. Figure 3 shows a Java method to implement a VCODE floating-point vector multiplication operation.

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

Figure 3: Java implementation of the VCODE operation `* FLOAT`, which performs an elementwise multiplication of two floating point vectors.

VCODE implements NESL's nesting of data structures efficiently by using *segmented vectors* [3]. 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.

3.2 Translating VCODE into Java

The `vcodetojava` phase uses a Perl script to generate the Java code. An associative array maps most VCODE instructions to calls to methods in `VcodeEmulation`, while conditionals and function calls are translated directly to Java conditionals and methods. Figure 5 shows an example of the translation process for a dot-product function in VCODE, which was generated from a single line of NESL code. As can be seen, the translation into an equivalent Java method is straightforward and can be applied on a line-by-line basis.

The VCODE program is implemented by the `run` method of a thread spawned by a Java applet. VCODE graphics operations are mapped to calls from this thread to Java's standard AWT library, while asynchronous

```
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 implementation of the segmented VCODE operation `+_REDUCE FLOAT`, which sums the individual segments within a floating-point vector.

tasks such as refreshing and event handling are handled by the parent applet thread. This corresponds to the use by the standard NESL system of a subprocess, connected by Unix pipes to the VCODE interpreter, to interact with the X window system.

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
POP 1 1
RET
```

Java:

```
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. The extra Pop and Copy stack operations handle segment descriptors. `s` is an instance of the `VcodeEmulation` vector-stack class.

4 Examples of NESL Applets

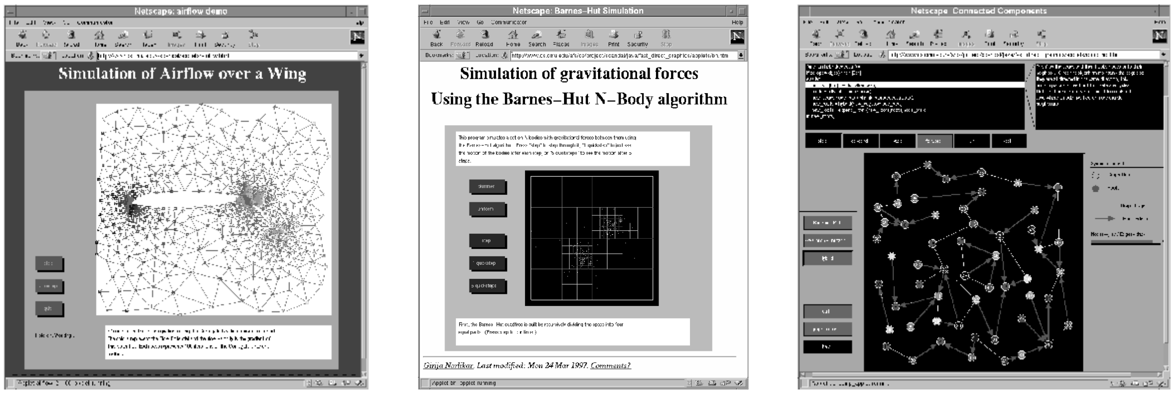
In this section we describe three existing NESL applications that are now available as Java applets on the web. Figure 6 shows screen shots of the applets running in a web browser; other applications that we have converted include a geometric graph separator and a two-dimensional convex hull [15].

1. **Airflow simulation.** This is a simulation of airflow over an airfoil using a parallel implementation of the finite volume method. Steps in the simulation include: finding a Delaunay triangulation of the set of points; generating the dual Voronoi diagram from the triangulation; writing a linear equation for each Voronoi cell such that the net flow into each cell is zero; setting the boundary conditions (Neumann conditions); solving the set of linear equations using the conjugate gradient method; and generating

the flow velocities on each triangle from the flow potentials at the three corners. The generated mesh can be zoomed in or out, and each step is animated to aid in its explanation.

2. **N-body simulation.** This is a simulation of gravitational forces between N bodies in two dimensions, using the Barnes-Hut $O(N \log N)$ algorithm. In each timestep, the algorithm builds a quadtree based on the bodies, calculates the total force on each body by traversing the quadtree, and updates the positions and velocities of the bodies. The simulation allows both uniform and non-uniform distributions of bodies.

3. **Connected components tutorial.** This tutorial demonstrates three algorithms for finding the connected components in a graph: random mate, Awerbuch-Shiloach, and a hybrid of the two. The tutorial highlights each line of code as it is executed, explaining its effect on the displayed graph. It also displays graph statistics as it runs, and allows the user to edit the graph.



(a) (b) (c)

Figure 6: Screen shots of algorithm animations written in NESL, compiled into Java, and running as applets: (a) airflow simulation, (b) N-body simulation, and (c) connected components tutorial. From `http://www.cs.cmu.edu/~scandal/applets/`.

5 Benchmarking the System

It would be impractical to use Java as a new intermediate language if its performance was not comparable to that of the existing system. In this section we compare the performance of the generated Java code running on different Java virtual machines to that of the original VCODE running on a native VCODE interpreter. Specifically, we test:

- *JDK:* A Java interpreter. Java interpreters are available for most current platforms.

- *JIT:* A just-in-time (JIT) Java compiler. JIT compilers use run-time code generation to reduce interpretive overhead in iterative constructs such as loops; however, they are more difficult to implement (for example, at the time of writing, no web browser for a Unix platform has a JIT Java compiler).

- *Native:* The native VCODE interpreter, written in C and linked against a serial version of the CVL library. The code has been tuned for asymptotic performance on large vectors, with hand-unrolled loops and a memory-management mechanism designed specifically for VCODE.

We use three standard NESL benchmarks:

- *Least-squares line-fitting*: Finds the best fit to a sequence of points using the least-squares method. It is simple straight-line code with no conditionals or loops, and hence can be used to measure interpretive overhead [13].

- *Selection (generalized median-finding)*: Finds the element in a vector that would be at a specified position if the vector were sorted. It picks the middle element of the vector as a pivot to split it into two parts, and recursively finds the required element in the appropriate part.

- *Sparse matrix-vector multiplication*: Multiplies a sparse matrix stored in compressed row format by a dense vector. It uses a nested data-parallel algorithm.

We give the source code and test data for the benchmarks in Appendix A. All three benchmarks have asymptotic running times that are linear in the size of the problem. Timings for the benchmarks have previously been reported in [7, 12].

5.1 Methodology

To minimize performance effects due to machine architecture, we used two different machines for benchmarking: a Sun SPARCstation 5/85 running Solaris 2.5, and a DX4-120 PC running Windows 95. Compilation was done using NESL 3.1, gcc v2.7.0, and JDK 1.1.1, with full optimization. We used Sun's latest reference JDK 1.1.1 as the Java interpreter on both platforms (although the code generated by our system is backwards-compatible with JDK 1.0.2). In addition, on the PC we used Microsoft's JIT compiler from Internet Explorer v3.02, and on the Sparc we used Sun's JIT compiler for JDK 1.0.2.

All benchmarks were performed on idle machines to minimize outside effects. This is particularly important for the Java benchmarks, because Java provides only a time-of-day clock rather than a per-process timer. The poor resolution of the standard 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 total run took at least one 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 garbage collection before the beginning of a benchmark. Each of the benchmarks was run five times at problem sizes ranging from 2^1 to 2^{17} (131072). Table 1 contains results for selected problem sizes, while Figure 7 shows performance graphs. We'll analyze the results for the line-fit benchmark in depth (concentrating on the PC platform), and then briefly discuss the results for the selection and sparse matrix-vector multiplication benchmarks.

5.2 Line-fit Benchmark

On the line-fit benchmark, the native VCODE interpreter is fastest, as we would expect, and its relative advantage is greatest at small vector sizes. Absolute performance falls off after approximately 16,000 elements, when the problem no longer fits into the PC's 256 KB L2 cache.

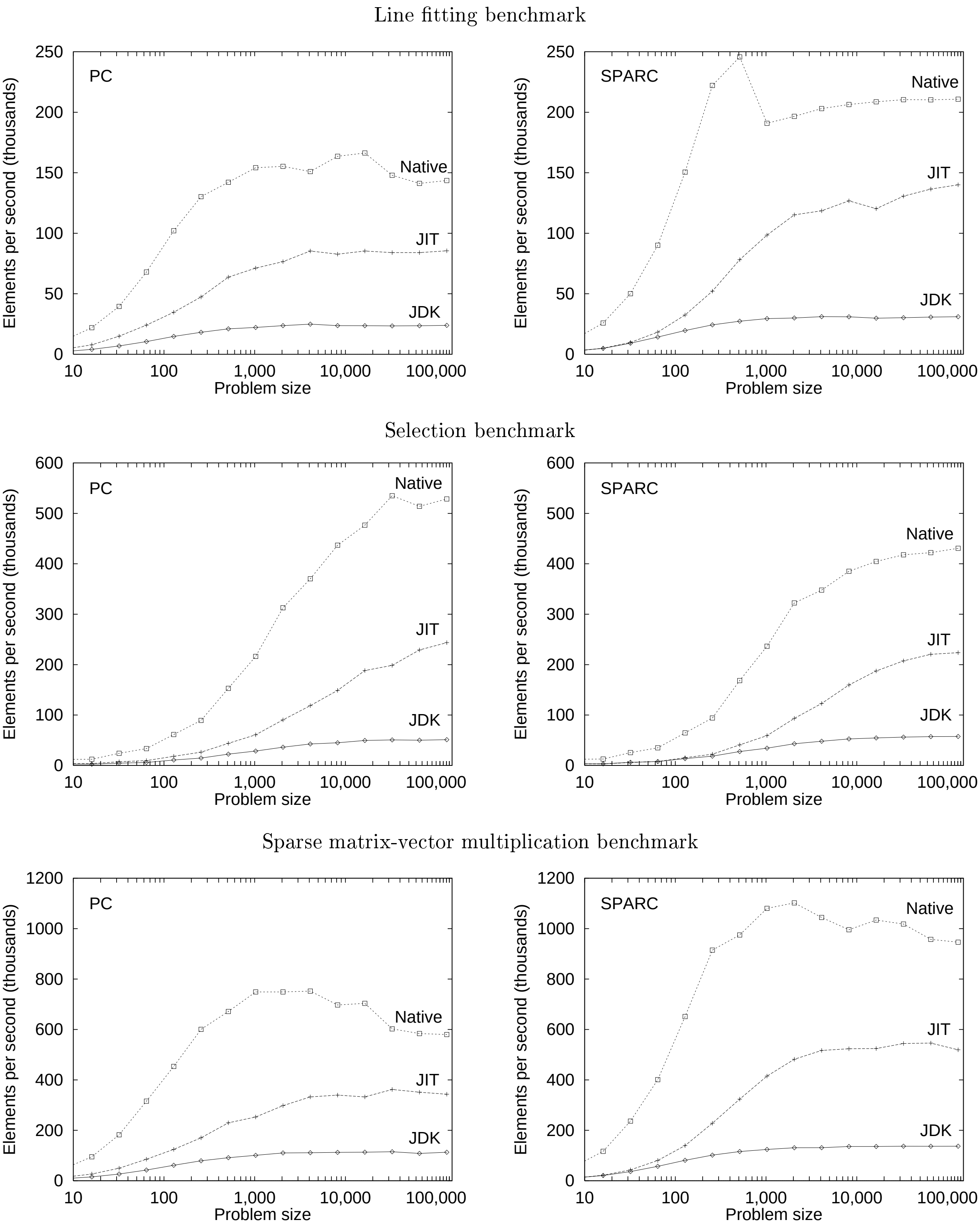


Figure 7: Performance of different intermediate language implementations for three Nesl benchmarks using a DX4-120 PC (left) and a SPARCstation 5/85 (right). Note that the x axis has a log scale.

Problem size	DX4-120 PC			SPARCstation 5/85		
	JDK	JIT	Native	JDK	JIT	Native
	Line-fit					
16	3.9	2.0	0.73	3.3	3.2	0.62
128	8.7	3.7	1.3	6.5	3.9	0.85
1024	46	14	6.6	35	10	5.4
8192	346	99	50	265	65	40
65536	2780	780	464	2132	480	312
	Selection					
16	6.5	4.1	1.3	4.8	5.2	1.3
128	12	7.0	2.1	9.4	8.3	2.0
1024	36	16	4.7	30	17	4.3
8192	182	55	19	156	51	21
65536	1308	286	127	1145	297	155
	Sparse matrix-vector multiplication					
16	1.0	0.60	0.17	0.7	0.73	0.14
128	2.1	1.0	0.28	1.6	0.92	0.20
1024	10	4.0	1.4	8.2	2.5	0.95
8192	73	24	12	60	16	8.2
65536	604	187	112	479	120	68

Table 1: Average running times in milliseconds for three NESL benchmarks using different intermediate language implementations on a DX4-120 PC and a SPARCstation 5/85.

The JIT compiler achieves about half the performance of the VCODE interpreter for problems bigger than approximately 1,000 elements. The line-fit benchmark is dominated by looping over arrays; we suspect that some of the performance loss is to due to the requirement that a Java VM must initialize every element of an array before use, and must perform bounds checks on every array access. 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; to our knowledge, however, these optimizations are not performed by any current JIT compiler.

Finally, the JDK interpreter achieves about one-sixth the performance of the native VCODE interpreter, since it has the additional overhead of interpreting every bytecode instruction in the loops of the code. This is too slow for computationally-intensive simulations; JIT compilers are required for good performance.

We can also use the results of the line-fit benchmark to calculate the fixed overhead and asymptotic time per element of each implementation. This is possible because the benchmark executes a fixed number of VCODE operations—and hence should has a fixed VCODE interpretive overhead—for all problem sizes. The resulting figures for the constant overhead and the time per element are shown in Table 2.

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.

	120 MHz 486 PC		SPARCstation 5/85	
Implementation	Overhead	Per-elt.	Overhead	Per-elt.
JDK	3220	42	2420	32
JIT	1870	12	3060	7.3
Native	460	6.8	410	4.7

Table 2: Constant overhead and asymptotic time per element (in microseconds) for the NESL line-fit benchmark using different intermediate language implementations on a PC and a SPARCstation.

5.3 Selection Benchmark

The ordering of results for the selection benchmark is the same as for the line-fit benchmark: the VCODE interpreter is fastest, followed by the JIT compiler and finally the JDK. However, the shapes of the curves are different, 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.

5.4 Sparse Matrix-Vector Multiplication Benchmark

The ordering of results for the sparse-matrix vector multiplication benchmark 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.

5.5 Memory Usage

The space efficiency of an intermediate language is often 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 does not immediately 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 Pros and Cons of Java

A working prototype of the system (without graphics support) was written in a weekend, with the time divided roughly equally between the translation script and the vector methods. Java's standard collection of predefined classes proved very helpful in this rapid-prototyping phase, as did its built-in garbage collection. Object-oriented features were not heavily used—in essence, we treated Java as a secure, portable dialect of C with garbage collection, polymorphism, and a good collection of pre-existing libraries.

However, two aspects of Java proved troublesome during development. First, Java lacks templates, parametric polymorphism, a built-in preprocessor, or any other way to generate type-specialized versions of the same basic method. Thus, each VCODE vector operation has to be implemented as a separate method, although many of them perform essentially the same loop, changing only the operator or type. We used the `m4` preprocessor to generate multiple specialized methods from a single macro body. An alternative would be to add parametric polymorphism to Java using a system such as Pizza [16].

Second, Java compilers and virtual machines are still in their infancy. The Sun JDK compiler does not perform standard optimizations such as common subexpression elimination or loop-invariant code motion (for example, the reference to `a.length` in the loop in Figure 3 will be evaluated on each loop iteration). Currently, these optimizations must be performed by the programmer. Additionally, there is a lack of performance data available for use in making informed design and optimization decisions. We performed a series of micro-benchmarks to establish the comparative cost of various Java operations [11]. These benchmarks show that creating objects is a dominant cost for small problem sizes and becomes relatively more expensive in just-in-time compilers. We were therefore careful in our design to avoid creating unnecessary objects (for example, we never create objects in an inner loop).

7 Related Work

There are currently more than 25 language implementations that use Java in some way [19]. They include both academic and commercial projects, and range from visual programming systems, through interpreters for Prolog and Basic, to compilers for Ada, Scheme, Lisp, and Rexx. The interpreters are designed mainly for demonstration purposes rather than computationally-intensive simulations, because the extra level of interpretation significantly reduces their performance. Of the compilers, some generate Java bytecode directly for reasons of functionality (for example, the Kawa Scheme compiler [8] uses the GOTO bytecode instruction to perform tail-recursion elimination). However, by compiling into Java source code instead of bytecode, we can take advantage of source-level tools such as the JAVAR restructuring compiler for automatic parallelization [2], and optimizing Java-to-C compilers for generating stand-alone executables [14, 17].

A recent study of interpreters (including one for Java) concluded that performance is related to the expressiveness of the language, the use of native libraries, and the way memory is accessed [18]. These effects can be seen in the standard NESL system, which achieves good performance by using an expressive intermediate language (in particular, the interpretive overhead of each VCODE vector instruction is amortized over the length of the argument vectors), and a native CVL library that performs efficient unit-stride vector memory accesses. For the system described in this paper, the expressiveness and advantages of VCODE remain the same, while native libraries can be supplied either dynamically (compiled from `VcodeEmulation` by a JIT compiler) or statically (by linkage to native code [13]). However, as noted in Section 5, memory access is hindered by the requirement to initialize arrays and perform bounds checks, and this limits the performance of the generated code on current Java virtual machines.

8 Conclusions

This paper has described the design, implementation, and evaluation of a system to compile the parallel language NESL into serial Java code. This system has enabled us to convert existing simulations and algorithm animations written in NESL into Java applets [15]. By combining the portability of Java with the power of a high-level programming language we can quickly make available demonstrations of new parallel algorithms developed by our project.

Java was easy to use, and has enough functionality to allow a clean implementation of our system. The translation was made significantly easier by NESL's use of a vector-based intermediate language, whose operations can be implemented by a Java class; a similar approach could be used for other data-parallel languages such as HPF and APL. Just-in-time Java compilers achieve about half the performance of the native NESL implementation on a set of algorithm benchmarks, and this performance gap is likely to narrow as Java technology improves. We conclude that Java is a viable choice as an intermediate language, and that it can be used to increase the portability of existing languages for scientific simulation and computation. Source code is available from the authors.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] Aart J.C. Bik and Dennis Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, June 1997. See also `http://www.extreme.indiana.edu/~ajcbik/JAVAR/`.

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

[4] 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.

[5] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Symposium on The 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. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

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

[9] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with HPJava. In *Concurrency: Practice and Experience*, June 1997. See also http://www.npac.syr.edu/users/dbc/HPJava/experiments/.

[10] James Gosling. Java intermediate bytecodes. *Proceedings of Workshop on Intermediate Representations, ACM SIGPLAN Notices*, 30(3), March 1995.

[11] Jonathan C. Hardwick. Java optimization. http://www.cs.cmu.edu/~jch/java/optimization.html.

[12] 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.

[13] Jonathan C. Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, August 1996.

[14] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Usenix Conference on Object-Oriented Technologies and Systems*, June 1997. See also http://www.irisa.fr/compose/harissa/harissa.html.

[15] Girija Narlikar. Scientific simulations and algorithm visualizations using NESL and Java. http://www.cs.cmu.edu/~scandal/applets/.

[16] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997. See also http://www.dcs.gla.ac.uk/~wadler/topics/pizza.html.

[17] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications. A *way ahead of time* (WAT) compiler. Technical Report TR97-01, Department of Computer Science, University of Arizona, 1997. See also http://www.cs.arizona.edu/sumatra/toba/.

[18] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996. See also http://cag-www.lcs.mit.edu/asplos7/program/.

[19] Robert Tolksdorf. Programming languages for the Java virtual machine. http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html.

All CMU papers can also be found 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 double-precision 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 double-precision data.