The Cost of Being Object-Oriented: A Preliminary Study

Zoran Budimlic

Ken Kennedy

Jeff Piper

Center for Research on Parallel Computation

Rice University

Abstract

Since the introduction of the Java programming language, there has been widespread interest in the use Java for the high performance scientific computing. One major impediment to such use is the performance penalty paid relative to Fortran. To support our research on overcoming this penalty through compiler technology, we have developed a benchmark suite, called OwlPack, which is based on the popular LINPACK library. Although there are existing implementations of LINPACK in Java, most of these are produced by direct translation from Fortran. As such they do not reflect the style of programming that a good object-oriented programmer would use in Java. Our goal is to investigate how to make object-oriented scientific programming practical. Therefore we developed two object-oriented versions of LINPACK in Java, a true polymorphic version and a "Lite" version designed for higher performance. We used these libraries to perform a detailed performance analysis using several leading Java compilers and virtual machines, comparing the performance of the object-oriented versions of the benchmark with a version produced by direct translation from Fortran. Although Java implementations have been made great strides, they still fall short on programs that use the full power of Java's object-oriented features. Our ultimate goal is to drive research on compiler technology that will reward, rather than penalize good object-oriented programming practice.

1.0 Introduction

Since the introduction of the Java programming language, there has been widespread interest in the use of Java for high performance scientific computing. The principal impediment to such use is poor performance, relative to the similar programs written in Fortran and C. There are three main reasons why Java programs do not achieve high performance by comparison with Fortran and C:

- Java compilers and execution environments are not yet on par with the traditional optimizing compilers. Although there has been a significant advancement in this area lately [2, 7, 9], especially with the run-time compilation and optimization techniques, Java systems still have to improve to be able to compete with the traditional languages.
- The non object-oriented features of Java add significant overhead. Bytecode portability requires that major optimizations be delayed until run time. Garbage collection, synchronization, and the exception mechanism all require additional overhead for their implementation. Java security measures force the virtual machine implementation to examine the code for security holes before execution. All these requirements, important as they are, reduce the performance of Java programs at run time.
- Java is an object-oriented language, and as such it encourages programers to use object-oriented style when writing scientific programs. It is far more natural for the programers to thinks of matrices, vectors and complex numbers as objects and pass them around and use encapsulation and code reuse when performing operations on them, than to perform all the operations directly on the Fortran-style arrays. Later in the paper, we show that Java compilers (both static and JIT) are not yet up to the task of effectively optimizing away the overhead resulting from using the object-oriented style.

In spite of the cost, much of the value of using Java is lost if the programmer does not freely use the advanced features of the language, particularly the support for object-oriented program development. The right solution is to build compiler systems that minimize the penalties for fully utilizing the features of the language. However, effective research on Java compiler systems must be driven by experimental methods. Without good benchmarks on which to conduct these experiments, it will be difficult to validate the compiler strategies proposed by researchers.

The majority of benchmarks available for evaluation of the cost of using Java in high performance scientific computing are either microbenchmarks or benchmarks obtained by direct translation from Fortran (either automatic [4], semi-automatic or manual [11]). Neither of these two closely resemble the programs that Java programers would prefer to write. The need for a benchmark that would closely reflect the ‘real world’ scientific computation in Java is clear. Unfortunately, although there have been some reports of scientific applications implemented in Java that could be easily converted to serve as benchmarks, we suspect that many of these have been translated to Java without a corresponding conversion to true object-oriented programming style. A good example is the Java version of the LINPACK Benchmark, which strongly resembles the Fortran version.

To address this issue and to help foster more research on Java compilation, we have designed and implemented in Java an object-oriented version of the LINPACK linear algebra library. We call this library OwlPack (Objects Within Linear algebra PACKage). We used OwlPack to perform a detailed analysis of the performance of Java programs written in different programming styles. Specifically, we compared the performance of the object-oriented version of the benchmark with a version written in a style closer to Fortran. We then analyzed the cost of the additional overhead incurred when object-oriented design is used in high-performance computing.

The principal contributions of this research are as follows:

- We have constructed an object-oriented Java implementation of the LINPACK library that is available to the public (http://www.cs.rice.edu/~zoran/OwlPack.zip) for evaluating the performance of Java compilers and run-time systems.
- We have used this package to evaluate the cost that is associated with object-oriented design of scientific programs using the best Java compilers and VMs available today.
- We have analyzed the results of these experiments to illuminate important issues that must be addressed by Java compiler and run-time systems research if we are to improve the performance of scientific programs written in Java.

The rest of this paper is organized in the following way: in Section 2.0 we describe the design of our implementation of the OwlPack, present the class hierarchy and explain some design decisions. In Section 3.0 we present the experimental results we obtained by executing several of our routines on various VMs and using different static compilers, along with the same experiments performed on a Fortran-style version of LINPACK library. In Section 4.0 we provide some insights and guidance for compiler designers, based on the results reported in Section 3.0. In Section 5.0 we describe some of the work related to the topic of this paper and future research plans.

2.0 OwlPack Design

The creation of a general framework for classes in Owlpack is motivated by the primary goal of employing the most natural hierarchy possible while achieving the highest level of abstraction. The central design decision was how to store the elements of a matrix. Using two dimensional array of primitive types, like double and float, permits direct manipulation of the data and does not force translation during parameter passing to the library, since the users would be likely to have their data in this form. However, this approach suffers from the disadvantage that the class hierarchy would have to be replicated for each primitive type.

On the other hand, a polymorphic solution could be achieved by creating a new abstract class of numbers to which any primitive type can be translated. This approach allows for a single implementation for all primitive types and can be extended to any type for which the required operations are defined. Therefore, it is easier to program, easier to maintain, and more general than the specializing the code to each primitive type. In that sense, it uses the full power of the object-oriented features of the language.

As an experiment, we decided to explore both styles. We created two different matrix abstractions, Matrix and NMatrix, that represent the different storage strategies while performing the same numeric computations. The Matrix class is the superclass of all classes that are specialized based on primitive types and gives rise to a replicated class hierarchy parametrized by primitive element type. We call the resulting style of programming “lite” object-oriented style (Lite OO) style.

The classes that derive from NMatrix all use the abstract number class, LNumber. Thus NMatrix is fully polymorphic and can be extended to new number types easily. We will refer to this style of programming as object-oriented (OO) style.

FMatrix, DMatrix, and CMatrix all extend Matrix and contain the BLAS (Basic Linear Algebra Subroutines) for the corresponding data type (F = float, D=double, C=complex). The CMatrix class is further specialized for double and single precision complex numbers. In our current implementation, the complex numbers used in CMatrix are objects that are an extension of our LNumber class, although consistency with the efficiency goals of the Lite OO style would dictate that CMatrix use pairs of two-dimensional arrays of primitive types—one array for the real part of the complex numbers and one array for the imaginary parts. We plan to make this change in a future version. All classes whose names begin with an N are extensions of NMatrix and use LNumbers.

FIGURE 1. LNumber Class Hierarchy

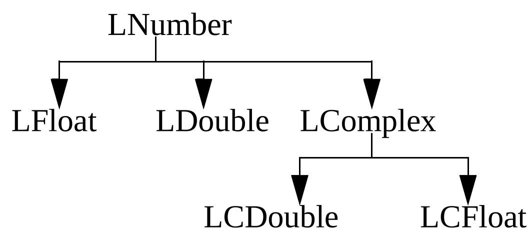
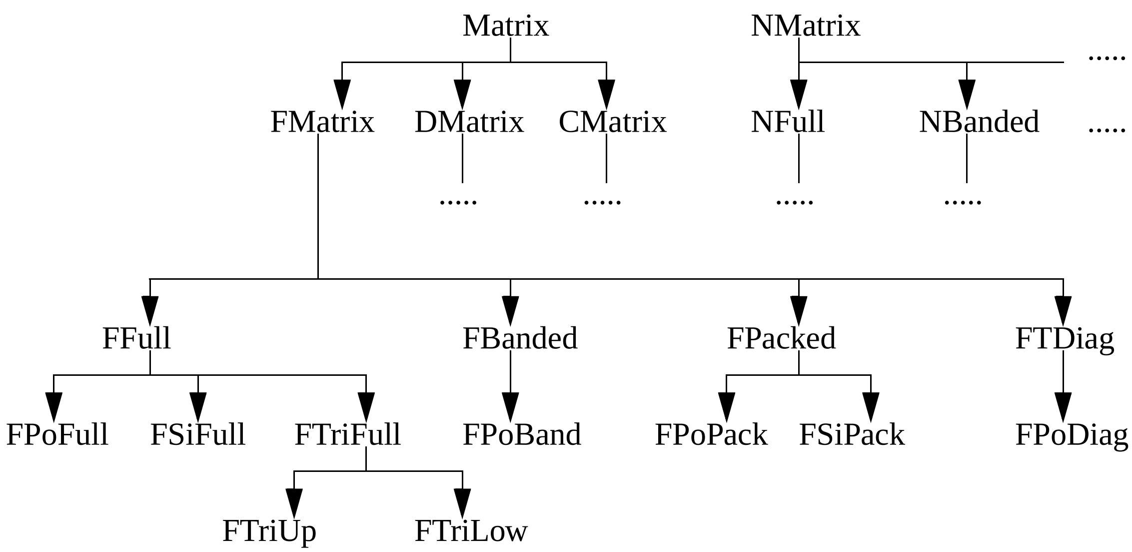


FIGURE 2. OwlPack Class Hierarchy



For all classes, the next differentiation is based on the storage type, so DMatrix is extended by DBanded, DPacked, DTDiag, and DFull (and similarly for FMatrix, CMatrix and NMatrix). By separating the matrices by storage form, the classes further down the hierarchy can reuse code because the internal variables are accessed the same way.

To attain a high degree of generality, Matrix and NMatrix contain the size information, an internal pivot array and the base interface for all of the methods available for matrices. These method are, in general, overridden by the extended classes. However, the base functions throw an exception whenever the base method is not overridden because it does not always make sense to provide every method for every extension. For example, with banded matrices it does not make sense to implement the inverse operation, because the inverse would require more storage than the allocated for the original Matrix. In the interest of abstracting these functions, our solution is to throw an exception whenever an invalid function call is made. This brings most of the interesting methods to the Matrix level where storage form and data type are not important.

In order to raise solve() and determ() to the highest level, we created a Vector class that is barely more than an array of the primitive type being used in the corresponding Matrix. Since Matrix must account for single, double, and complex precision, but cannot see the types of the Matrix elements inside it, solve() and determ() could not return an array of the same type. Instead, they return a subclass of Vector that contains an array of the needed primitive type.

FIGURE 3. OO style determinant computation

```
for(int i=0;i<cols;i++) {
      if(pivot[i] != i+1) Det[0].negate();
      Det[0].multTo(Mat[i][i]);
      if(Det[0].equals(0)){
        return Det;
       }else{
        while(!(Det[0].abs()).greaterOrEqual(1)) {
          Det[0].multTo(10);
          Det[1].subTo(1);
        }
        while((Det[0].abs()).greaterOrEqual(10)) {
          Det[0].divTo(10);
          Det[1].pplus();
        }
       }
}
return Det;
```

The next level of the structure was determined so that the subclasses of this level could have the most in common. Because floats, doubles, and any new number type cannot interact with each other easily, the storage type was deemed to be the most useful factor of separation between the Matrices. By having FMatrix, DMatrix, CMatrix, and NMatrix, we obtained a set of abstract classes that contain the BLAS for each type, which is the natural basis for such a set of routines.

The doubly subscripted array that each of these classes also holds was chosen over a faster design consisting of a single array with computed subscripts because the form (arguably) better represents the way people think about matrices. An example of the differences among the OO style, the Lite OO style, and the code resulting from direct Fortran translation code is provided in Figures 3, 4, and 5 by the code for computing and normalizing the determinant.

The OO style code works for all LNumbers, invoking methods for comparison and numeric operations.

The Lite OO style allows for the direct manipulation of the information within the array, so that primitive operations are performed on the elements directly. Before returning with the determinant, though, a new Vector must be formed.

FIGURE 4. “Lite” OO style determinant computation

```
for(int i=0;i<cols;i++) {
   if(pivot[i] != i+1) Det[0] = -Det[0];
   Det[0] *= this.Mat[i][i];
   if(Det[0] == 0){
      return (new DVector(Det));
   }else{
      while(Math.abs(Det[0]) < 1) {
        Det[0] *= 10;
        Det[1] -= 1;
      }
      while(Math.abs(Det[0]) >= 10) {
        Det[0] /= 10;
        Det[1]++;
      }
   }
}
return (new DVector(Det));
```

The Fortran-style code, while similar in appearance to the second style above, follows Fortran conventions of passing in the Matrix and breaking out of loops. This function also gets a reference to `det` as an argument and hence there is no need to return it.

FIGURE 5. Fortran style determinant computation

```
for (i = 0; i < n; i++) {
   if (ipvt[i] != i) det[0] = -det[0];
   det[0] *= a[i][i];
   if (det[0] == 0.0) break;
   while (Math.abs(det[0]) < 1.0) {
      det[0] *= ten;
      det[1]--;
   }
   while (Math.abs(det[0]) >= ten) {
      det[0] /= ten;
      det[1]++;
   }
}
```

DBanded, DFull, and DTDiag all extend DMatrix and are at the highest level of non-abstract Matrices. These classes contain the Gaussian (LU) decomposition routines since they are the most general routines that can be used by any Matrix with the corresponding storage form. If a Matrix is declared as a sub-class of one of these types, it can be factored and solved with Gaussian elimination by calling its super class's factor and solve routines. DPacked is an abstract class which does not contain the Gaussian routines.

DFull also contains the singular value decomposition, qr-decomposition, and Cholesky's updating routines. The code for Cholesky decomposition is also in DFull so that both symmetric subclasses can use it, but it is impossible to call it from DFull.

In designing NMatrix, we encountered some difficulties when dealing with LNumbers, because these two classes are completely separate in their internal representation and LNumber is a fully self-supporting class. There were several instances when a new LNumber was needed inside an NMatrix, but because LNumber is an abstract class, it was impossible to create a new instance. To solve this problem, we made LNumber implement Clonable() and added the methods setZero() and setOne() which would set the value of the LNumber to zero or one, respectively. An example of the intended implementation can be found in the method NFull.determ():

```
Det[0] = Mat[0][0].Clone();
Det[1] = Mat[0][0].Clone();
Det[0].setOne();
Det[1].setZero();
```

Another consideration when implementing LNumber was the way in which methods should be called. For example, LNumber could have `add` defined to take LFloat, LDouble, LCFloat, and LCDouble and throw an exception if the method is called and not overloaded for the type, much the way `Number.getValue()` is implemented in the standard Java library. Every class would overload the method that took its own type and perform the operation quickly because it knows precisely what the passed object is. While slightly faster, this method was not considered appropriate for the full object oriented version, since it would introduce into LNumber a knowledge of all of the classes that extend it. Instead, by having the method take an LNumber and cast it, only one version of the method is written, and the exception is still thrown if an incompatible LNumber is passed.

3.0 Experimental Results

We compared our implementation of the LINPACK library with the partial Java version from FPL Statistics Group [11], obtained by the straight-line transformation from the Fortran source code. We will refer to this version of the code as “Fortran style”. We have only included the timings for the routines that have been implemented in the FPL version—factorization and solving the positive definite matrix, LU and QR decomposition and solving of the full matrix, inverse and determinant computation and singular value decomposition. These routines were only available for double precision floating point numbers, so we compared the running times of their equivalents in our implementation: DPoFull and DFull classes handle these functions in our “Lite” OO version, while NPoFull and NFull classes instantiated with LDouble numbers handle them in our OO version. A short description of these routines is given below:

- `dpofa` factors a 300x300 random generated positive definite matrix
- `dposl` solves the equation A*x = B, where A is the matrix factored by dpofa and B is random generated vector of 300 numbers
- `dpodi` computes the determinant and the inverse of the 300x300 positive definite matrix
- `dgefa` performs an LU factorization of a 200x200 random generated full matrix
- `dgesl` solves A*x = B, where A is factored full matrix, B is a vector size 200
- `dgedi` computes the determinant and the inverse of a full 200x200 matrix
- `dqrdc` performs QR decomposition with pivoting on a 300x300 random full matrix
- `dqrsl` solves A*x = B, where A is QR decomposed matrix, B is a vector
- `dsvdc` performs the singular value decomposition on a random 100x100 matrix

The tests on Solaris were performed on the Sun Ultra 5, with 64MB of memory, running under Solaris 2.6, with the jdk 1.1.5 from JavaSoft for the interpreter tests, and the jdk 1.2 Production Release for Solaris for the JIT tests.

TABLE 1. Experimental results for Sun Ultra 5

	jdk 1.1.5 interpreter			jdk 1.2 JIT		
	Fortran style	**“Lite” OO**	**OO style**	**Fortran style**	**“Lite” OO**	**OO style**
dpofa	4.719	5.675	40.404	1.334	1.607	7.073
dposl	8.866	10.323	81.779	2.646	2.730	11.017
dpodi	10.664	12.343	146.283	2.729	3.547	16.439
dgefa	3.113	3.540	41.863	1.122	0.950	5.478
dgesl	4.107	6.978	33.553	1.176	1.023	5.086
dgedi	6.213	7.020	83.148	2.317	2.022	10.519
dqrdc	21.278	25.106	218.473	7.020	7.821	30.875
dqrsl	14.232	16.344	109.558	4.015	4.149	16.277
dsvdc	9.017	14.949	126.192	1.396	2.717	24.664

TABLE 2. Experimental results for Pentium Pro

	jdk 1.1.6 interpreter			Symantec JIT 3.00.029			Microsoft VM 4.79.2405		
	Fortran style	**“Lite” OO**	**OO style**	**Fortran style**	**“Lite” OO**	**OO style**	**Fortran style**	**“Lite” OO**	**OO style**
dpofa	5.598	6.470	78.854	0.621	0.822	40.899	0.631	0.711	15.041
dposl	16.994	18.677	110.489	8.342	7.641	77.682	6.580	6.860	27.579
dpodi	12.989	14.340	198.956	1.262	1.952	135.415	1.763	1.882	51.384
dgefa	4.056	4.316	69.550	0.411	0.541	20.399	0.521	0.580	14.490
dgesl	5.327	6.048	45.556	0.571	0.671	16.474	0.511	0.571	12.168
dgedi	7.100	8.512	127.153	0.831	1.082	44.774	1.021	1.072	27.239
dqrdc	24.285	29.312	306.231	4.065	4.296	201.74	3.605	4.477	74.807
dqrsl	18.686	20.410	180.409	2.033	2.183	99.083	1.883	2.083	46.086
dsvdc	11.627	20.46	265.202	1.703	2.253	93.455	1.032	2.985	59.876

The PC tests were performed on a 200 MHz Pentium Pro with 64MB of memory, running under Windows NT Workstation 4.0, using the jdk 1.1.6 from JavaSoft for the interpreter tests, and the Symantec JIT 3.00.029 that comes with Symantec Visual Cafe for the JIT tests. We also measured the execution times for the Microsoft VM 4.79.2405 that comes with Microsoft Java SDK 3.0.

All tests are run as single processes on empty machines with no other processes running. All times are an average of three runs. All the classes were compiled using javac from jdk 1.1.6 with the -O option.

We encountered a surprising result with the Symantec JIT on Windows NT: the 3.00.053 version that comes with jdk 1.1.6 had virtually identical results on Fortran style and on the “Lite” OO benchmarks as the 3.00.029 version of the Symantec JIT that comes with Symantec Visual Cafe, but on the OO benchmarks the 3.00.053 JIT was 50-100% slower than the 3.00.029 version.

For reference, we performed the same measurements of the Fortran version of the LINPACK, using the Fortran 90 native compiler for Solaris. In general, the Fortran style Java LINPACK on the jdk 1.2 Production Release was only 70% - 130% slower than the native Fortran compiled version on our Sparc Ultra 5, which suggests that the today’s JIT Java compilation technologies have improved performance to within the factor of two of native Fortran code.

4.0 Analysis of the Results

The data presented in the previous section provides some interesting and somewhat surprising results. The substantial performance hit associated with the object-oriented design was expected, although the magnitude of this cost exceeded our expectations.

In general, the Lite OO version of OwlPack performed slightly slower than the Fortran-style version, with differences ranging from 12% faster for `dgedi` on Solaris JIT, to around 3 times slower for `dsvdc` on Microsoft VM. The main reason for this performance degradation is extra indirection in the innermost loops of our routines. The Fortran-style library has all the matrices passed by reference to the targeted routines, where the argument of the routine contained the direct reference to the matrices involved in the computation. The “Lite” OO library treats matrices as objects, with the reference to the actual array that contains the matrix data stored as an instance variable. The most natural way to reference the matrix data is through the instance variable, i.e. `Mat[i][j]`, but this generates an extra field access to obtain `Mat` from `this` reference, followed by the usual array access instructions. In most, but not in all cases, the JIT compilers were able to optimize this extra reference away. This problem could be easily eliminated even in a static compiler with some form of loop invariant code motion.

The most interesting timings were the ones recorded for the OO version of the OwlPack. The object oriented version was substantially slower in all of our tests, ranging from 4.4 times slower than the Fortran-style version for `dqrdc` on jdk 1.2 on Solaris, to more than 100 times slower for `dpodi` on the Symantec JIT. The average slowdown was about a factor of 10.

Although the Lite OO version of OwlPack has an enormous performance advantage over the OO version on current Java implementations, the polymorphic object-oriented style results in a code that is much smaller in size—a factor of 3 for OwlPack, expected to grow to factor of 4 when the intended implementation of CFMatrix and CDMatrix is complete. In addition it is easier to maintain because algorithmic changes need only be made once. Ideally, future compiler technologies will be able to automatically transform the OO style OwlPack code into something that approaches the Lite OO version in performance.

The main reasons for the poor performance of the OO version are fairly obvious:

- Every number that is a part of a computation is allocated on the heap as a separate object, requiring additional overhead for instantiation and garbage collection.
- Numbers that are elements of a matrix are scattered over the heap, effectively eliminating the cache performance benefits of spatial locality in standard matrices used in the Fortran and OO Lite versions.
- All operations on numbers are done through method calls to the corresponding objects, incurring additional overhead for the method invocation and the dynamic dispatch required to determine which method is being invoked (since all numbers are abstracted in the `LNumber` class).
- There is a much greater memory requirement for the OO style, as every number takes up memory associated with object representation in addition to memory for the data itself.
- The presence of objects and method calls prevents some forms of local compiler optimization, such as common subexpression elimination, that are possible in the Fortran and Lite OO versions.

It should be possible to eliminate most of these performance penalties through advanced compiler optimizations. A form of *specialization* or *procedure cloning* [8, 5, 6] could convert the NMatrix class hierarchy into four specialized hierarchies based on the four types of the numbers that could be used as elements of the matrix (LDouble, LFloat, LCDouble, LCFloat). This optimization alone would not bring significant performance gains, but coupled with *unboxing* or *object inlining* [3, 9, 14] that would convert the internal representation from arrays of LDouble for example, to arrays of `double` and inline all of the operations on numbers, it would generate a class structure very similar to the Matrix hierarchy in OwlPack. The real challenge is to improve on these techniques so they can be applied in Java environment without sacrificing the portability and security of the Java VM.

5.0 Related and Future Work

Dongarra et al. [2, 3, 10] are currently working on automatic translation from Fortran to Java and applying it to LAPACK library. While this is an attractive way to quickly transform existing high performance Fortran codes into Java and obtain reasonable efficiency (as demonstrated in this paper), the resulting code would not reflect the best object-oriented programming style. If the goal is to conduct research on how to ameliorate the cost of using the powerful object-oriented features of Java, code produced by this process would not make an ideal benchmark.

Pendragon Software Corporation [16] provides one of the most frequently cited benchmarks for Java. However, the benchmark is not designed to reflect the high performance scientific computation. Furthermore, the product is a microbenchmark, which has led to some controversy about the practice of matching patterns in the code.

This paper demonstrates that there is a need for further research on compiler technology for object-oriented languages like Java. The JIT compilers of today are not sophisticated enough to eliminate the overhead incurred by polymorphic object-oriented implementations. Static compilers must overcome special problems even before they can apply traditional optimizations because the exception mechanism interferes with control flow analysis and makes it difficult to use the straightforward implementation of some code motion optimizations [2].

Whole [7] and almost whole [3] program optimization techniques have to be further developed for Java environment. Because these strategies sacrifice some of the portability and security of the language, they might not be acceptable to some users.

Dean et al. [7, 8], Agesen [1] and others [2, 3, 5, 9] have been working on compiler technologies that would utilize whole program optimization and type analysis to deliver procedural code from object-oriented programs and eliminate the overhead imposed by the object-oriented design. It has been shown that the use of *object inlining* [2, 3, 9] and *specialization* and *customization* techniques [5, 8] in compilation of scientific Java programs can lead to major improvements in running time. However, these techniques have yet to be validated on a large scale, real world applications and in a more restrictive Java bytecode environment. One goal of the benchmark suite we have produced is to make such evaluations possible.

6.0 Conclusions

We have designed and implemented object-oriented versions of the BLAS and LINPACK linear algebra library in Java. The implementation has been done in two flavors: a fully polymorphic, object-oriented version and a more performance-oriented “Lite” version. Using several leading Java implementations, we compared the performance of the object-oriented versions of the benchmark with a version using Fortran style Java generated by a translator and analyzed the cost of additional overhead that object-oriented design brings to the high-performance computing. We observe that there is a substantial cost associated with a full object-oriented design in scientific programs, and that the current compilers and VMs have a room for substantial improvements in this area.

We should emphasize that the intention of this project was not to design a linear algebra library that scientific programmers should use in their Java programs—if that were the goal we would probably use the translation from Fortran or the Lite style. Instead, our goal was to expose the performance penalties that are associated with object-oriented design and programming style, and to provide compiler researchers with benchmarks that more closely reflect how numerical applications would be implemented using the full power of an object-oriented language. Our long-term goal is to drive our own research (and that of others) on the Java compiler technology needed to eliminate the overhead of the object-oriented approach. If this research is successful, it will reward, rather than penalize, good programming practice.

Ideally, scientific programmers should be able to design their systems in an object-oriented style, without attention to the performance issues. One goal of Java compiler research should be to help achieve that ideal.

7.0 References

[1] O. Agesen. *Concrete type inference: delivering object-oriented applications.* Ph.D. thesis, Stanford University, 1995.

[2] Z. Budimlic and K. Kennedy. *Optimizing Java: theory and practice*. Concurrency: Practice and Experience 9(6), 445-463, 1997.

[3] Z. Budimlic and K. Kennedy. *Static Interprocedural Optimizations in Java*. Rice University technical report CRPC-TR98746, 1998.

[4] H. Casanova, J. Dongarra and D. M. Doolin. *Java Access to Numerical Libraries.* http://www.cs.utk.edu/f2j/hpjhtml/

[5] C. Chambers and D. Ungar. *Customization: Optimizing Compiler Technology for Self, a Dynamically-Typed Object-Oriented programming language*. In Proceedings of the ACM SIGPLAN '89 Conference on PLDI, 24(7):146-160, July 1989.

[6] K. Cooper, M. Hall and K. Kennedy. *Procedure cloning*. Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, 96-105, April 1992.

[7] J. A. Dean. *Whole program optimization of object-oriented languages.* Ph.D. thesis, University of Washington, 1996.

[8] J. Dean, C. Chambers, and D. Grove. *Selective specialization for object-oriented languages*. In Proceedings of the ACM SIGPLAN '95 Conference on PLDI, pages 93-102, June 1995.

[9] J. Dolby. *Automatic Inline Allocation of Objects*. In Proceedings of ACM SIGPLAN conference on POPL, Las Vegas, Nevada, June 1997.

[10] J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart. *LINPACK Users' Guide.* SIAM, Philadelphia, 1979.

[11] FPL Statistics Group. *Linear Algebra for Statistics Java Package.* http://www1.fpl.fs.fed.us/linear_algebra.html

[12] P. Havlak. *Interprocedural symbolic analysis*. Ph.D. thesis, Rice University, Dept. of Computer Science, May 1994.

[13] M. T. Heath. *Scientific Computing: An Introductory Survey.* WCB/McGraw-Hill, 1996.

[14] X. Leroy. *Unboxed Objects and Polymorphic Typing*. In Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on POPL, 177-188, Albequerque, New Mexico, January 1992.

[15] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification.* Reading, Mass., Addison-Wesley, 1996.

[16] Pendragon Software Corporation. *Caffeine Mark™ Java Benchmark*. Available on the Web: http://www.webfayre.com/cm.html