Prospects for Scientific Computing in Polymorphic, Object-Oriented Style *

Zoran Budimlić† Ken Kennedy†

Abstract

Since the introduction of the Java programming language, there has been widespread interest in the use of Java for high performance scientific computing. One major impediment to such use is the performance penalty paid relative to Fortran. Although Java implementations have made great strides, they still fall short on programs that use the full power of Java's object-oriented features. In this paper, we present an analysis of the cost associated with polymorphic, object-oriented scientific programming in Java, and discuss the compiler strategies that would reduce the overhead of such programming style. Our ultimate goal is to foster the development of compiler technology that will reward, rather than penalize good object-oriented programming practice.

1 Introduction

When one considers Java as a platform for high-performance scientific applications, several general performance considerations quickly come into focus. Today's Java implementations are not yet on par with the native compiled, optimized Fortran or C code. There is some additional overhead associated with Java portability, security and multi-threaded model [15]. Finally, Java is an object-oriented language, and as such it encourages programmers to use object-oriented style when writing scientific programs. Although Java implementations have matured considerably, they still fall short on programs that use the full power of Java's object-oriented features.

In this paper, we present an analysis of the cost associated with polymorphic, object-oriented scientific programming in Java, and discuss the compiler strategies that would reduce the overhead of such programming style. The ultimate goal of our research is to develop compiler technology that will reward, rather than penalize good object-oriented programming practice.

There are three main reasons why Java programs do not achieve high performance relative to 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, 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. The Java portability model, based on the Java VM bytecodes, requires that important

*This work is sponsored by the Center for Research on Parallel Computation

†Center for Research on Parallel Computation, Rice University

optimizations be delayed until run time [2]. Garbage collection, synchronization, and the exception mechanism all require additional overhead for their implementation. The Java security model requires the Java VM 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 programmers to use object-oriented style when writing scientific programs. It is far more powerful for the programmers to think of matrices, vectors, and complex numbers as objects, accessing them only through the associated methods, rather than perform all the operations directly on the underlying Fortran-style arrays.

A great deal of progress has been made toward addressing the first two problems. Performance of Java programs that are written in non object-oriented, Fortran-style is within a factor of four of equivalent native compiled and optimized Fortran code. However, in our recent study [4], we have shown 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. The loss in performance due to polymorphism, memory allocation and garbage collection, and the added indirection due to encapsulation was up to two orders of magnitude.

In this paper, we suggest some compiler strategies for *whole-program* and *almost whole-program* compilation that would help reduce the overhead of polymorphic, object-oriented design.

Almost whole program compilation is a strategy in which the compiler assumes a static class hierarchy at the compile time [3] and the programmer specifies the classes that would be publicly visible. Applied to Java, this allows for extensive program optimization, with certain limitations when compared to *whole program compilation* [11, 6]. The generated code is fully portable and verifiable Java bytecode, in JAR archive form.

Class specialization is an optimization where class is specialized based on the exact type of the polymorphic data it contains [2, 7, 10]. Several specialized versions of the original class are generated, effectively pushing the type distinction upwards in the class hierarchy.

Object inlining is an optimization in which whole objects that are contained inside a given class are inlined and replaced with the more primitive data that comprise those objects [2, 12]. All operations on the objects are replaced by direct operations on inlined data.

These optimizations, along with other techniques will help bridge the gap between polymorphic, object-oriented scientific programs and their more efficient, Fortran-style equivalents. Used in the context of *almost whole program compilation* of Java they will automate the process of generating efficient, Fortran-style bytecode from more general, elegant, object-oriented polymorphic source code.

2 Motivation

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

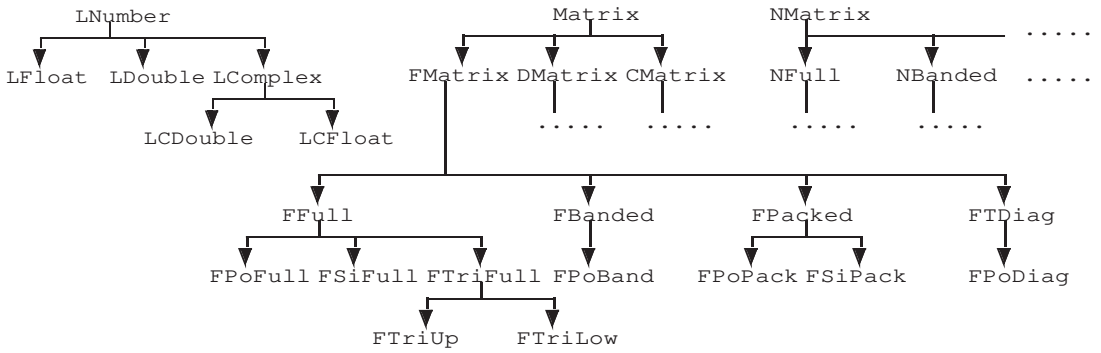

FIG. 1. *OwlPack class hierarchy*

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 (automatic [5], semi-automatic or manual [14]). Neither of these closely resemble the programs that Java programmers 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 [4]. 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 library 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.

Figure 1 shows the class hierarchy of OwlPack. `LNumber` is an abstract number class that describes general numbers. `LFloat`, `LDouble` and `LComplex` implement single and double precision floating point numbers, as well as single and double precision complex numbers. `NMatrix` class is designed to use `LNumber` class hierarchy, and effectively implements the LINPACK library for all four number types at once, using polymorphism of the `LNumber` class hierarchy. We refer to this version of the library as the "OO style" implementation.

We have also implemented a specialized version of this code that is reflected in the `Matrix` class hierarchy. Instead of working with polymorphic, general numbers, the classes in the `Matrix` hierarchy push this type distinction up to the matrix level: `FMatrix` implements the LINPACK library for single precision floating point numbers, `DMatrix` handles the double precision numbers e.t.c. The code is essentially quadrupled when compared to our "OO style" version. It reflects the style of code that we believe programmers in high performance, scientific computing in Java are forced to write today given the status of the current Java implementations. We refer to this code as the "Lite OO" version.

As we will show in Section 5, the performance of the OO version of the code suffers

drastically when compared to the "Lite OO" version. 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 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.

3 Class Specialization

Procedure cloning [8, 9], is a well-known interprocedural optimization, in which distinct copies of a procedure are made for call chains with distinct parameter values. For example, if two different call chains deliver different procedure values to a procedure argument, cloning can disambiguate those call chains. Another example is constant propagation. If there are call chains for which the value of a parameter is a known constant, cloning permits that constant value to be used in optimizations in the clone specialized to that value. Typically performance improvements derive from later interprocedural optimizations, such as dead code elimination and constant folding, rather than from cloning itself.

For object-oriented languages, the concept of cloning has has been been further generalized. For those languages, the optimization is known as *customization* [7, 17] or *specialization* [10]. Specialization clones methods based on the *instantiation type* of the object on which these methods are executed—the object referred to by "*this*". Paralleling cloning, the main contribution of specialization is generation of a more precise call graph and elimination of dynamic dispatches that could appear at every call site.

Class specialization is a generalized version of the optimization we have proposed before [2, 3]. Instead of specializing a method only on the exact type of the *this* argument, the class is specialized based on the exact types of the polymorphic data it contains.

Figure 2 shows the result of the process of class specialization on a part of our OwlPack code. Result of specializing the class on figure 2a are the classes on figures 2b and 2c. Class `NMatrix` contains variable `Mat`, which is a reference to an array of `LNumber` objects. `LNumber` class is polymorphic, so according to it's class hierarchy shown on figure 1 four specialized versions of the class `NMatrix` are created: `LFloat_NMatrix`, `LDouble_NMatrix`, `LCDouble_NMatrix` and `LCFloat_NMatrix` that contain arrays of objects that have corresponding exact type. Only two of the four generated classes are shown on figure 2.

```
class NMatrix{
  LNumber[][] Mat;
  int rows, cols;
  int[] pivot;
  void NMatrix(LNumber[][] F)
  {
    rows = F.length;
    cols = (F[0]).length;
    Mat = F;
    pivot = new int[cols];
  }
 ...
}
```

a. Original class

```
class LDouble_NMatrix{
  LDouble[][] Mat;
  int rows, cols;
  int[] pivot;
  void LDouble_NMatrix
        (LDouble[][] F)
  {
    rows = F.length;
    cols = (F[0]).length;
    Mat = F;
    pivot = new int[cols];
  }
 ...
}
```

b. Specialized for type LDouble

```
class LCFloat_NMatrix{
  LCFloat[][] Mat;
  int rows, cols;
  int[] pivot;
  void LCFloat_NMatrix
        (LCFloat[][] F)
  {
    rows = F.length;
    cols = (F[0]).length;
    Mat = F;
    pivot = new int[cols];
  }
 ...
}
```

c. Specialized for type LCFloat

FIG. 2. *Class Specialization of NMatrix class*

A whole-program type inference [1, 19] or a set-based analysis [13] is needed to provide necessary information in order to perform this transformation. The creation points and exact types of the variables declared as `NMatrix` have to be determined. All the references to those variables are then changed to refer to the corresponding exact type.

Unfortunately, a naive use of this transformation would result in an exponential code growth in the worst case. Heuristics have to be applied to isolate objects that would be the most profitable to inline using the optimization described in the next section prior to performing class specialization only on their classes. For example, our heuristics may determine that most objects we are interested in inlining are instantiated as `LDouble`. In this case, only one specialized version of the `NMatrix` class would be generated (`LDouble_NMatrix`), and would be used wherever `NMatrix` was used with `LDouble` numbers. All other uses of `NMatrix` would remain untouched.

4 Object Inlining

Object inlining as discussed in this section is a natural generalization of the object inlining optimization we described in our earlier work [2, 3]. Instead of inlining only objects that are live inside a single method and do not escape the context of the method where inlining is done, we now concentrate on a more global application of this optimization, in which objects contained within a particular class are inlined and replaced with their data. All the references to the previous array of objects are transformed to references to the newly created array(s) of inlined data.

On our example from figure 2c, the array of `LCFloat` numbers inside the `LCFloat_NMatrix` class is transformed into arrays that contain the appropriate data from the `LCFloat` class, as shown on figure 3b.

In order to perform object inlining on the `LCFloat_NMatrix` class, all the creation points of that class are identified and arguments to its methods (including constructors) are to be object inlined as well.

Adequate attention has to be paid to the aliasing problem. All references to an object that is to be inlined have to be known exactly. Conservative alias analysis would not suffice here, the aliasing information has to be precise. If the alias analysis cannot determine this information exactly for a particular object, that object cannot be inlined.

Figure 4 shows the transformation of the code where specialized classes and inlined

```
public class LCFloat
   extends LComplex {

  private float real;
  private float imag;

  public LCFloat() {
    data = 0;
    imag = 0;
  }
  ...
}
```

a. Class of the inlined object

```
class LCFloat_NMatrix{
  float[][] Mat_real;
  float[][] Mat_imag;
  int rows, cols;
  int[] pivot;
  void LCFloat_NMatrix
    (float[][] real, float[][]imag)
  {
    rows = real.length;
    cols = (real[0]).length;
    Mat_real = real;
    Mat_imag = imag;
    pivot = new int[cols];
  }
 ...
}
```

b. Container class after object inlining

FIG. 3. *Object Inlining*

```
LNumber [][] array = new LNumber[n][n];

LNumber [] Vec = new LNumber[n];

for(i=0;i<n;i++){
  Vec[i] = new LCFloat(5.0*i,4.0*i);
  for(j=0;j<n;j++){
    array[i][j]=new LCFloat(2.0*i,3.0*j);
  }
}


NMatrix Mat = new NMatrix(array);


Mat.axpy(n,Mat[4][5],1,1,1,Vec,1,1,1);
```

a. Original code

```
float [][] array_real = new float[n][n];
float [][] array_imag = new float[n][n];
float [] Vec_real = new float[n];
float [] Vec_imag = new float[n];
for(i=0;i<n;i++){
  Vec_real[i] = 5.0*i; //inlined
  Vec_imag[i] = 4.0*i; //constructor
  for(j=0;j<n;j++){
    array_real[i][j]= 2.0*i;
    array_imag[i][j]= 3.0*j;
  }
}

LCDouble_NMatrix Mat = new
  LCDouble_NMatrix(array_real,array_imag);

Mat.axpy(n,Mat[4][5],1,1,1,Vec_real,
                      Vec_imag,1,1,1);
```

b. Transformed code

FIG. 4. *Using Inlined Objects*

objects are used. Arrays `array` and `Vec` are object inlined directly inside the method that instantiates them. The contents of the object `Mat` are object inlined (instead of an array of `LNumber` objects it contains two arrays of a primitive type `float`). The method interfaces have to be changed to accommodate passing of these inlined (or "unboxed" [18]) objects. A global interprocedural *mod* analysis has to prove that the parts of the objects that are inlined are not modified inside the called methods, otherwise these objects have to be boxed before the call and unboxed afterwards. Alternative solution is to perform object inlining on the object whose method is being called (`Mat` in our example) which would make the context of the calling method available to the inlined called method (`axpy` in our example).

An analysis based on the *regular section analysis* [16, 20] has to prove that arrays of objects that we want to inline are instantiated completely. If there are "holes" in the arrays, or if the arrays are not of a uniform type, object inlining cannot be performed on them.

Ultimately, if there is no recursion (other than tail recursion) and no arrays with

polymorphic elements or holes, it would be possible to inline all the objects and their methods inside the `main` method of the main class. There would be no method calls and no object instantiations. All the computation that the program performs would then be expressed inside the `main` method. Of course, this is not a very practical solution since there is a potential for exponential code growth. In addition to that, some JIT compilers might encounter problems compiling such a huge method and revert to interpretation of the code (as we have experienced with the JDK 1.2 production release VM for Windows platform), which would greatly reduce the effectiveness of the optimization. A careful application of object inlining with limited total code expansion and moderate method size increase would give the best results.

5 Experimental Results

We compared our implementation of the LINPACK library with the partial Java version from FPL Statistics Group [14], 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 benchmarks is given below:

- *dpofa* factors a 300x300 random generated positive definite matrix
- *dposl* solves the equation $A * x = B$, where A is a factored matrix, B is a size 300 vector
- *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

Although the Lite OO version of OwlPack has an enormous performance advantage (up to two orders of magnitude) over the OO version on current Java platforms, the polymorphic object-oriented style results in a code that is much smaller in size – a factor of 4 for OwlPack. In addition it is easier to maintain because algorithmic changes need only to be made once. Compiler technologies that we are proposing in this paper will be able to automatically transform the OO style OwlPack code into something that approaches the Lite OO version in performance.

As a proof of concept, we have performed class specialization and object inlining by hand on a part of our OwlPack library. The "Optimized OO style" column in tables 1 and 2 show the execution times for our optimized version of the OO style code. We have achieved an improvement of up to two orders of magnitude over the object-oriented version of the OwlPack. As expected, optimized code approaches the performance of the Fortran style

TABLE 1
Pentium Pro Execution Times

	jdk 1.2				**Microsoft VM 4.79.2405**				**F90**
	Fortran style	*"Lite" OO*	*OO style*	*Optimized OO style*	*Fortran style*	*"Lite" OO*	*OO style*	*Optimized OO style*	*Win32 native*
dpofa	0.511	0.741	75.849	-	0.721	0.711	52.966	-	0.210
dposl	6.91	7.13	102.307	-	6.69	7.011	40.278	-	0.781
dpodi	1.292	1.763	226.275	-	1.792	1.943	164.547	-	0.601
dgefa	0.411	0.551	69.039	0.521	0.541	0.581	41.420	0.570	0.311
dgesl	0.510	0.681	20.870	0.571	0.531	0.561	8.912	0.581	0.481
dgedi	0.841	1.031	124.730	1.052	1.051	1.101	68.509	1.121	0.441
dqrdc	2.854	4.046	437.149	-	3.725	4.477	275.226	-	1.211
dqrsl	1.782	1.983	155.674	-	2.123	2.103	67.307	-	0.711
dsvdc	0.952	2.233	170.586	-	1.052	2.734	91.542	-	0.421

TABLE 2
Sun Ultra 5 Execution Times

	jdk 1.1.5 interpreter				**jdk 1.2 JIT**				**F90**
	Fortran style	*"Lite" OO*	*OO style*	*Optimized OO style*	*Fortran style*	*"Lite" OO*	*OO style*	*Optimized OO style*	*Solaris native*
dpofa	4.669	5.548	115.155	-	1.353	1.584	21.862	-	0.352
dposl	8.82	10.362	127.392	-	2.582	2.831	12.036	-	0.643
dpodi	12.736	14.315	297.346	-	2.614	3.462	70.965	-	0.649
dgefa	3.226	3.511	81.981	3.889	0.698	0.838	18.496	0.796	0.194
dgesl	4.079	4.640	35.868	4.481	0.865	0.967	4.488	1.162	0.307
dgedi	6.234	6.971	161.868	7.798	1.428	1.677	33.288	1.693	0.364
dqrdc	21.197	25.139	538.53	-	6.921	8.118	123.438	-	1.120
dqrsl	14.504	15.757	162.044	-	3.953	3.903	15.459	-	0.908
dsvdc	9.008	15.439	226.495	-	1.456	3.043	64.054	-	0.347

code. We expect additional reduction in the execution times of the optimized code that will bring it even closer to the Fortran style after some obvious opportunities for "classical" local optimizations (namely constant and copy propagation) that are created by straightforward inlining are exploited.

6 Almost Whole Program Compilation

The optimizations described in the previous sections have one major impediment: they require whole program analysis and compilation. Although this is not a problem in other environments [11, 6], Java presents a serious obstacle to this approach. If we want to use Java's portability and security, these optimizations either have to be performed at a bytecode level or implemented in a JIT compiler. Unfortunately, the second option is not feasible at this point, since the whole program analysis and optimization techniques are very expensive (running times for fastest implementations measure in tens of minutes for programs of more than ten thousand lines).

On the other hand, we prefer not to restrict a user to only download and run the existing program over the internet. That would considerably limit the flexibility of the usage of Java bytecodes and undermine their original purpose.

To address this limitation, we have proposed *almost whole program compilation*, a model

for bytecode distribution that would enable most optimizations discussed in the previous section *and* preserve some of the flexibility of the current bytecode distribution model. This strategy would employ the standard JAR file convention where the distributor specifies the classes that are to be publicly available. The user of such a package can download it over the Internet and freely instantiate and use its public classes. However, the user *cannot* add new classes to the hierarchy in the package or replace the existing classes without recompilation of the source code. On the other hand, there is no restriction on the creation of new classes (outside the class hierarchy of the JAR file) that would use public classes from the JAR file.

It is clear that the achievable level of optimization will depend on the amount of code the distributor wishes to make public. If the distributor designates only the main class from the package to be public, the compiler can use the equivalent of *whole program compilation*. On the other hand, if the programmer designates *all* of the classes from the package to be public, very little class specialization and in-class object inlining can be performed (see our previous work [2] for a discussion of this case).

We believe almost whole program compilation could be particularly useful for distribution of libraries with scientific code, where all of the code for the computation is in the library, but there is a need for a mechanism that would enable free usage of the top level public classes from the library *and* enable some very aggressive interprocedural optimizations like the ones we described above.

7 Conclusions

There are several important conclusions that can be drawn from the experiments and optimizations discussed in this paper.

- The current Java Virtual Machine implementations do a moderately good job of eliminating most of the cost of interpretation in Java execution. For linear algebra calculations coded in a style similar to Fortran, the penalty is no more than a factor of four over native Fortran 90. We expect the gap to be reduced to approximately a factor of two in the near future.

- For scientific code that employs the full polymorphic object-oriented style encouraged by Java, the picture is not so rosy. In our experiments, the performance of such a code was up to two orders of magnitude worse than the performance of Java code written in the Fortran style. Much of the problem can be attributed to the use of objects in place of built-in numeric types so that a single implementation can be used for all types.

- Fortunately, promising interprocedural optimizations such as class specialization and object inlining, when supported by an accurate interprocedural type analysis, can overcome much of the overhead of polymorphic style. Our hand experiments have demonstrated that the performance can be improved to be competitive with Java programs that are coded in the Fortran style.

- To achieve these benefits, a truly interprocedural compiler for Java is needed. The implication is that the entire program will need to be fixed at compile time so that the interprocedural analysis is accurate. If this is a major disadvantage, it is possible that the use of "almost whole program compilation" can ameliorate the problem in many cases.

It is clear that a great deal of additional research on optimization of Java programs is needed to make to make the implementation of serious scientific codes practical. The good news is that the strategies described here and elsewhere in the literature, if carefully implemented, are likely to bridge the gap in the not-too-distant future.

References

[1] O. Agesen, *Concrete Type Inference: Delivering Object-Oriented Applications*, PhD thesis, Stanford University, 1995.

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

[3] ——, *Static interprocedural optimizations in Java*, Tech. Rep. CRPC-TR98746, Rice University, Dec. 1998.

[4] Z. Budimlic, K. Kennedy, and J. Piper, *The cost of being object-oriented: A preliminary study*, in Workshop for Java for High Performance Network Computing, EuroPar, 1998. http://www.cs.cf.ac.uk/hpjworkshop/, to appear in Scientific Computing, 1999.

[5] H. Casanova, J. Dongarra, and D. M. Doolin, *Java access to numerical libraries*. http://www.cs.utk.edu/f2j/hpjhtml/.

[6] C. Chambers, J. Dean, and D. Grove, *Whole-program optimization of object-oriented languages*, Tech. Rep. TR-96-06-02, Department of Computer Science and Engineering, University of Washington, June 1996.

[7] C. Chambers and D. Ungar, *Customization: Optimizing compiler technology for Self, a dinamically-typed object-oriented programming language*, in Proceedings of SIGPLAN '89 Conference on PLDI, July 1989, pp. 146–160.

[8] K. Cooper, M. Hall, and K. Kennedy, *Procedure cloning*, in Proceedings of the 1992 International Conference on Computer Languages, Oakland, California, Apr. 1992, pp. 96–105.

[9] ——, *A methodology for procedure cloning*, Computer Languages, 19 (1993), pp. 105–117.

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

[11] J. A. Dean, *Whole Program Optimization of Object-Oriented Languages*, PhD thesis, University of Washington, 1996.

[12] J. Dolby, *Automatic inline allocation of objects*, in Proceedings of ACM SIGPLAN conference on POPL, Las Vegas, Nevada, June 1997.

[13] C. Flanagan and M. Felleisen, *Modular and polymorphic set-based analysis: Theory and practice*, Tech. Rep. TR96-266, Rice University, 1996.

[14] FPL Statistics Group, *Linear algebra for statistics Java package*. http://www1.fpl.fs.fed.us/linear_algebra.html.

[15] J. Gosling, B. Joy, and G. Steele, *The Java^{TM} Language Specification*, Mass.: Addison-Wesley, 1996.

[16] P. Havlak and K. Kennedy, *An implementation of interprocedural bounded regular section analysis*, IEEE Transactions on Parallel and Distributed Systems, (2)3 (1991).

[17] D. Lea, *Customization in C++*, in Proceedings of the 1990 Usenix C++ Conference, San Francisco, CA, Apr. 1990.

[18] X. Leroy, *Unboxed objects and polymorphic typing*, in In Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on POPL, Albequerque, New Mexico, Jan. 1992, pp. 177–188.

[19] J. Plevyak and A. Chien, *Precise concrete type inference for object-oriented languages*, in Proceedings of the Ninth Annual ACM Conference on OOPSLA, 1994, pp. 324–340.

[20] R. Triolet, F. Irigoin, and P. Feautrier, *Direct parallelization of CALL statements*, in Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, July 1986, pp. 176–185.