BROADWAY: A SOFTWARE ARCHITECTURE FOR SCIENTIFIC COMPUTING*

Samuel Z. Guyer
Dept. of Computer Sciences
The University of Texas
Austin, TX 78712 USA
sammy@cs.utexas.edu

Calvin Lin
Dept. of Computer Sciences
The University of Texas
Austin, TX 78712 USA
lin@cs.utexas.edu

Abstract Scientific programs rely heavily on software libraries. This paper describes the limitations of this reliance and shows how it degrades software quality. We offer a solution that uses a compiler to automatically optimize library implementations and the application programs that use them. Using examples from the PLAPACK parallel linear algebra library, we present our solution, which includes a simple declarative annotation language that describes certain aspects of a library's implementation. We also show how our approach can yield simpler scientific programs that are easier to understand, modify and maintain.

Keywords: software libraries, optimization, meta-interfaces

*This work was supported in part by NSF CAREER Grant ACI-9984660, DARPA Contract #F30602-97-1-0150 from the US Air Force Research Labs, and an Intel Fellowship.

1. INTRODUCTION

The goal of a software architecture is to promote code reuse and to allow programs to be easily maintained and modified. These goals are particularly difficult to achieve in the context of scientific computing, which can be characterized by three properties: (1) efficient runtime performance and efficient memory usage are critical, (2) the practitioners of scientific computing are typically not schooled in software engineering, and (3) deep knowledge of the scientific domain is required. The first property tempts programmers to emphasize performance over clarity, which often complicates the long term maintenance and portability of scientific codes. The second property explains why scientific programmers are typically unwilling to try novel languages or to use sophisticated design methodologies. In particular, it explains why scientific computing relies so heavily on software libraries. The third property, the requirement of deep domain knowledge, represents an underutilized opportunity that we will attempt to exploit.

Software libraries offer several strengths. They do not require the user to learn new language syntax, they can raise the level of abstraction to support common operations, and they provide a simple means of reusing code. Thus, software libraries have become a de facto software architecture for scientific programming. Unfortunately, libraries place the burden of optimization on the library user and force optimizations to be implemented directly in the application's source code. As this paper will illustrate, these manual optimizations adversely affect the application program by decreasing clarity, reusability, and portability, while increasing program complexity.

This paper describes a method of automating the optimization of library implementations and the application programs that use them. This new approach allows applications to use simpler interfaces to existing libraries, and it yields cleaner application programs that are easier to understand and maintain. Furthermore, our approach allows scientific programmers to continue using libraries in the same manner with which they have become accustomed. In essence, we are proposing a method of transforming software libraries into a viable and effective software architecture.

Figure 1 shows the overall architecture of our system. At the core is the Broadway compiler, which takes as input the application source code, the library source code, and a set of annotations that describe the library. The compiler produces as output an integrated, optimized library and application program.[1] The annotation language is critical because it conveys to the com-

[1]Many variations of this system are possible. For example, the library source might be encoded to prevent general access to the source, and the output code does not necessarily need to be produced as a single unified piece of code.

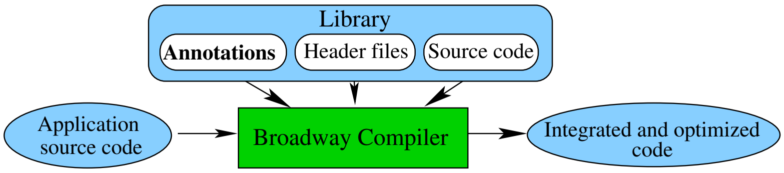


Figure 1 Architecture of the Broadway Compiler system

piler domain-specific information that can be used in the optimization process. These annotations allow the Broadway compiler to analyze and manipulate library operations in the same way that ordinary C compilers analyze and manipulate the primitives of the C language.

This paper makes the following contributions.

- We illustrate the long term maintenance and portability problems caused by the use of libraries in high performance programs.
- We describe the Broadway annotation language as a meta interface and explain how it improves the maintenance and portability of applications that use libraries.

The remainder of this paper is organized as follows. Section 2 explains the weaknesses of using software libraries as an architecture for creating performance-critical applications. Section 3 then explains how performance optimizations are typically applied to traditional libraries, and Section 4 explains how our solution uses a meta interface to address the weaknesses of existing software libraries. Section 5 discusses the long term benefits of our solution and its meta interface. We distinguish our work from related work in Section 6 and conclude in Section 7.

2. WEAKNESSES OF SOFTWARE LIBRARIES

Software libraries lead to a number of closely related performance problems:

1. **Different clients have different needs.** An implementation that is appropriate for one client can be inappropriate for another. Here we use the term “client” to refer to an application program that invokes library routines.
2. **“Separation of concerns” inhibits information flow across interfaces.** The performance of a library can typically be improved if the implementor is made aware of the client’s needs.
3. **Worst case assumptions provide generality at the expense of performance.** To provide correct behavior in all situations, libraries make

worst case assumptions, which can lead to excessive copying of data, excessive synchronization, and unnecessary initialization of data.

4 **Modular structure leads to poor resource management.** To provide encapsulation and safety, memory management is typically performed by library routines. However, resource management can often be improved by giving the application program control so that resources can be managed globally.

These performance problems are significant because they lead to a phenomenon that we call *Interface Bloat*. The only way that libraries can support a diverse set of clients is to provide a wide interface that includes a large number of specialized routines. Such interfaces can often be separated into two groups, a *Core* interface that provides all of the basic functionality of the library, and an *Advanced* interface that provides specialized routines that are applicable only in specific situations.

Interface Bloat leads to both short term and long term problems. The first short term problem is that large, complex interfaces are difficult to use. For example, MPI provides 12 ways to perform point-to-point communication [18]. These routines don't differ in their functionality, but differ in their buffering of data, their completion semantics, *etc.* The second short term problem is that the routines in the Advanced interface are typically more difficult to use, which increases the complexity of application programs. For example, MPI's Ready-Send assumes that the sending and receiving processes are already synchronized and that the receiver has prepared a sufficient buffer for the receipt of the message. Thus, Ready-Send requires the careful orchestration of the sending and receiving processes. Another example comes from the GNU Multi-Precision Library [11]:

> The `mpn` functions [*the Advanced interface*] are designed to be as fast as possible, **not** to provide a coherent calling interface. The different functions have somewhat similar interfaces, but there are variations that make them hard to use. These functions do as little as possible apart from the real multiple precision computation, so that no time is spent on things that not all callers need.

More seriously, Interface Bloat leads to long term software engineering problems with respect to both portability and maintenance:

No performance portability. Ready-Send is typically the most efficient form of point-to-point communication on distributed memory machines, but on machines with hardware support for shared memory, `MPI_Get()` and `MPI_Put()` are faster. Thus, programmers must recode their application to optimize the communication for different machines. This means, for example, that the invasive changes required to use Ready-Send can be counterproductive, as they complicate any subsequent porting and tuning efforts.

Premature Optimization Complicates Maintenance. The use of specialized routines represents a form of premature optimization, which is a common source of problems [16]. Because the optimizations are embedded in the source code, the program's overall logic can be obscured, making programs more difficult to read and maintain. For example, to be profitable, an asynchronous receive requires that some computation be moved above the `wait()` to hide the latency of the message:

```
send()              send()
recv()        =>    irecv()
compute();          compute1();
                    wait()
                    compute2();
```

This restructuring of the computation can make the program more difficult to understand since it breaks a single logical unit of computation into two pieces. It also implicitly introduces new dependence relations among the different pieces of code that must now be maintained. In the above example, the code in `compute1()` cannot be dependent on the data that is being sent.

Interface Bloat Defeats Modularity. Bloated interfaces often expose implementation details to the client. This violation of Parnas' modularity principle [19] leads to an overly strong coupling between modules. Whereas a buffered Send routine encapsulates all synchronization, Ready-Send scatters it throughout the program. Strong coupling defeats portability, as different hardware environments can prefer different versions of the point-to-point communication routines [6].

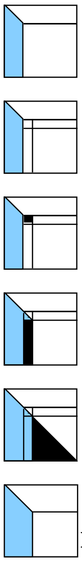
3. LIBRARY-LEVEL OPTIMIZATION

This section explains how the use of libraries can be optimized without incurring the penalties described in the previous section. We present a detailed example using a parallel linear algebra library, and we use this example to draw conclusions about library-level optimization and to characterize our compiler-based solution.

3.1. PLAPACK EXAMPLE

The PLAPACK library is a set of routines for coding parallel linear algebra algorithms in C or Fortran [21]. PLAPACK aims to provide high performance, and the library has been carefully designed by experts in the area of parallel linear algebra. PLAPACK consists of parallel versions of the same routines found in BLAS [8] and LAPACK [1]. At the highest level, it provides an interface that hides much of the parallelism from the programmer.

PLAPACK provides abstractions that can be useful for performing optimizations. For example, PLAPACK programs manipulate linear algebra objects indirectly though handles called *views*. A view consists of data, possibly distributed across processors, and an index range that selects some or all of the data. A typical algorithm operates by partitioning the views and working on one piece at a time. While most PLAPACK procedures are designed to accept any type of view, the actual parameters often have special distributions. Recognizing and exploiting these special distributions can yield significant performance gains [2].



(1) Find best partition size:

(2) Logically partition A:

(3) Factor A11:

(4) Solve A21 <– A21*A11^{-T}:

(5) A22 <– A22–A21*A21^{T}:

(6) Continue using A22 as A:

```
while (1) {
  PLA_Obj_split_size(A, PLA_SIDE_LEFT, &size_l, &tmp);
  PLA_Obj_split_size(A, PLA_SIDE_TOP, &size_r, &tmp);
  size = min3(nb, size_l, size_t);
  if (size == 0) break;

  PLA_Obj_split_4(A, size, size, &A11, &A12,
                                     &A21, &A22);

  Factor(A11);

  PLA_Trsm(PLA_SIDE_RIGHT, PLA_LOW_TRIAN,
           PLA_TRANS, PLA_NONUNIT_DIAG,
           one, A11, A21);

  PLA_Syrk(PLA_LOW_TRIAN, PLA_NO_TRANS,
           minus_one, A21, one, A22);

  A = A22;
}
```

Figure 2 Cholesky factorization using PLAPACK.

Figure 2 shows a Cholesky factorization program written with PLAPACK, along with graphical depictions of the matrix at each step. The `PLA_Obj_split_size` routines ensure that the split occurs on a processor boundary. Thus, the smallest piece, `A11` (the black view in step 3), resides entirely on a single processor, and `A21` (the black view in step 4) resides on a column of processors. We can exploit these two facts by replacing the general-purpose `PLA_Trsm` and `PLA_Syrk` routines with customized routines that run as much as three times faster [12].

3.2. LESSONS FROM OUR EXAMPLE

A key concept in the above optimization is the replacement of general routines with specialized routines that can make stronger assumptions about their calling context, and thus can execute more efficiently. Such optimizations are possible because most bloated library interfaces provide many specialized routines in their Advanced interface. In the case of PLAPACK, the interface is technically an "open infrastructure," which allows library users to see the lower levels of the library.

Another key to this optimization lies in analyzing the program to discover the special case matrix distributions. Human programmers who are facile with PLAPACK can perform such analysis manually. Conventional compilers, however, cannot perform such analysis because most programming languages have no notion of a matrix, let alone matrix distributions. Thus, to perform the types of optimizations described above, the compiler must be informed of the relevant domain-specific abstractions so that program analysis can be phrased in these terms.

Our compiler-based solution thus uses an annotation language to describe domain-specific information. The language provides a mechanism for identifying important library-specific concepts, such as the notion of a view in PLAPACK, and for enumerating important properties of those concepts, such as the fact that a view can reside on a single processor. For example, the following annotation identifies four important properties of views:

```
property Distribution = {Local, Empty, Matrix, ColPanel, RowPanel };
```

The annotations can also describe how the various library routines manipulate these properties and how such properties can be used to replace a general routine with a more specific and efficient one. For example, the `PLA_Obj_vert_split_2()` routine might have the following annotation:

```
int PLA_Obj_vert_split_2(obj, length, left, right)
{
  ...  // other annotations omitted

  property Distribution {
    (View1.Distribution == Matrix)   => left = Local, right = Matrix;
  }
  specializations {
    (View1.Distribution == Empty) => NOOP;
  }
}
```

The `property` construct indicates that this routine creates two views, `left` and `right`, with the specified properties; the `specializations` construct indicates that if `View1` (which is associated with `obj` through an annotation that is elided from this figure) is `Empty`, then an invocation of `PLA_Obj_vert_split_2()` can be removed since it is a no-op. Our annotation language also provides other features that facilitate program analysis. Details of our language can be found elsewhere [12, 13].

While the optimizations described in this section can be performed manually, two points are significant. First, such optimizations are tedious and require intimate knowledge of the PLAPACK library. Second, manual optimization is limited by the library's interface, but compiler-based optimization is not. In particular, the Broadway compiler can specialize library routines in ways that the library designer did not foresee, producing inlined or cloned versions that are optimized for their specific calling context.

4. BROADWAY AS A META INTERFACE

Section 2 enumerated four weaknesses of software libraries. The first of these has previously been identified as a limitation of *black boxes* [14, 15, 17]. In particular, the use of black boxes leads to performance problems because the implementation and interface that black boxes provide will inevitably be inappropriate for some client. One solution to this problem is to provide two interfaces, a *base interface*, which most clients use, and a separate *meta interface*, which allows the black box to adapt to the needs of different clients [14]. Figure 3 shows a Black Box and a Black Box that has been augmented with a meta interface.

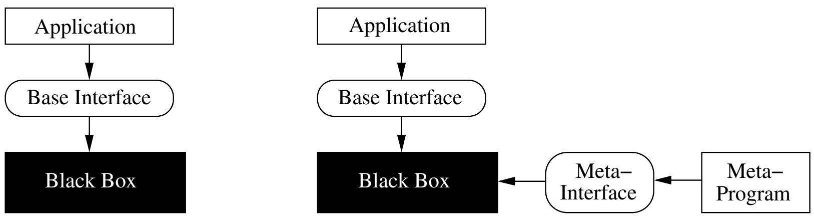


Figure 3 Black Boxes (left) and Black Boxes with Meta Interfaces (right).

The meta interface provides a controlled method of exposing the innards of a black box. The separation of the two interfaces is significant because each has different goals and each is aimed at a different user. The meta interface is aimed at sophisticated users and is typically accessed much less frequently than the base interface. Meanwhile, the base interface is aimed at the typical user who does not want to modify the black box. The separation of the two interfaces allows the base interface to retain the simplicity of an idealized black box interface.

The remainder of this section evaluates libraries and the Broadway compiler with respect to meta interfaces. We identify the different types of users in each system, the interfaces that are presented to these users, and the type of expertise that is expected of these users.

Traditional Libraries. Traditional libraries (Figure 4) have no meta interface. In such systems, there are only two users: the applications programmer who uses the library, and the library creator. The only way to provide customized implementations is for the library creator to expand the base interface, which forces the library user to deal with all of the problems of interface bloat. Bloated interfaces are poor substitutes for meta interfaces because they do not provide any mechanism for changing the implementation. This means that all specialized routines must be anticipated in advance by the library creator, rather than created in response to specific client needs.

The shaded boxes in Figure 4 represent the amount of expertise that is required to implement the various components. For example, with traditional libraries we see that the library writer must have considerable expertise in the library domain and must have some understanding of performance and application needs to implement algorithms efficiently. Significantly, we see that the C/Fortran compiler is given no knowledge of the library domain, so any library-level optimizations must be performed by the applications programmer. Thus, considerable burden is placed upon the applications programmer, who must not only understand the application domain, but must also possess considerable library, performance, and compiler expertise to achieve good performance.

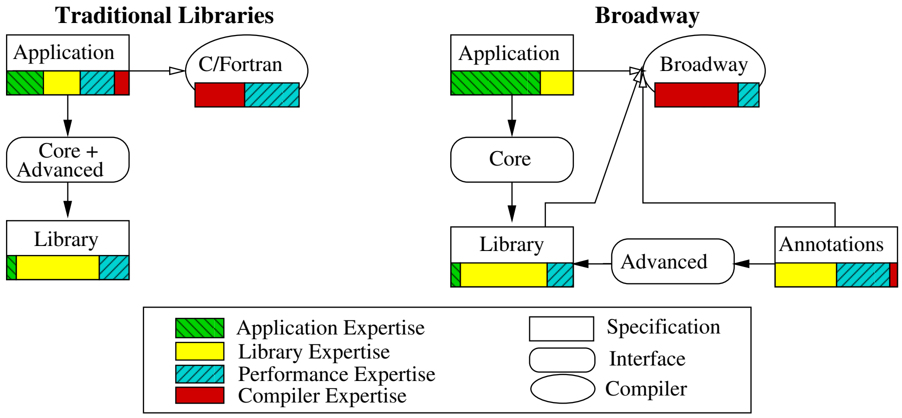


Figure 4 Comparison of Software Architectures

Broadway. The Broadway Architecture provides a meta interface to software libraries: The annotation language provides a way to change the library's implementation so that it is more suitable for a specific client. In this approach, there is, in addition to the library writer and user, a library expert who creates the annotations. This person may or may not be the same as the library creator. While the Broadway architecture shown in Figure 4 is more complex than the traditional library architecture, the added complexity is completely hidden from the applications programmer and the library writer. For example, the figure shows how the Advanced interface can be considered a part of the meta interface, rather than exposed to the applications programmer.

The Broadway meta interface is a language for describing domain-specific analysis and domain-specific transformations. For example, the language can easily configure an analysis that determines the data distribution of matrices in a PLAPACK program, as described in Section 3.1. The annotations can also concisely specify code transformations that are triggered by the results of this analysis [12, 13].

5. RESULTS AND DISCUSSION

This section evaluates our solution. We provide experimental evidence that our solution is effective, and we explain how our system's meta interface provides many benefits over traditional libraries.

Figure 5 [12] shows the result of applying our techniques to the `PLA_Trsm` routine of the Cholesky factorization program described in Section 3. The baseline measures the performance of the high quality but general purpose `PLA_Trsm` routine. The hand-optimized routine was optimized by members of the PLAPACK development team to exploit the specific distribution of matrices found in the Cholesky factorization program. Finally, the Broadway-optimized version represents a compiler-based approach that uses the same principles. The gap between the hand-optimized and Broadway-optimized approaches shows an important benefit of automated approaches—they can apply tedious transformations uniformly and completely.

5.1. BENEFITS OF THE BROADWAY ARCHITECTURE

Provides a mechanism for improving performance. The Broadway meta interface improves performance by addressing all four weaknesses of traditional software libraries (Section 2). First, our solution can create different library implementations and interfaces for different clients. Second, our solution conveys library-specific information to the compiler and uses this information to customize the library for different users. Thus, information flows across the meta interface through the Broadway compiler. Third, our solution replaces invocations of general routines with invocations to specialized routines, thereby relaxing worst case assumptions. These specialized routines might already exist in the library's Advanced interface, or these specialized routines might be created by the Broadway compiler. Finally, by integrating library and client code, our compiler can schedule operations globally, removing redundant operations across procedure call boundaries. While conventional compilers can

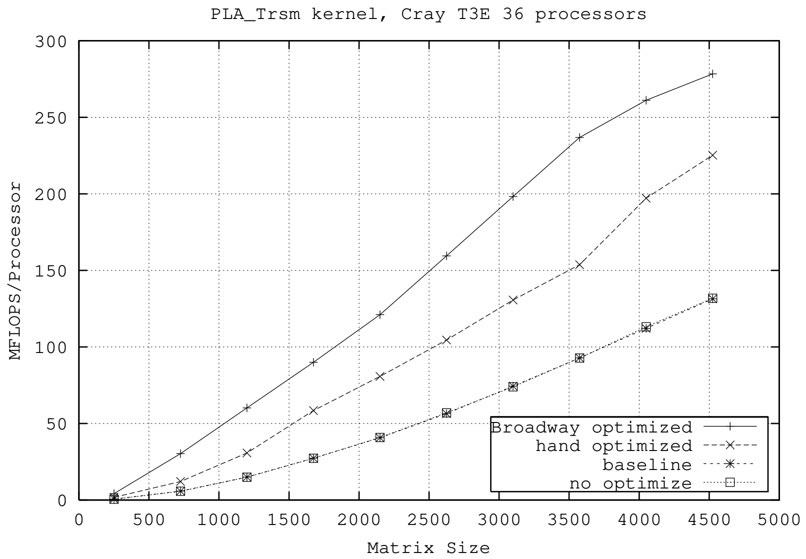


Figure 5 Performance comparison of baseline, hand-customized and Broadway-customized PLA_Trsm() function for the Cholesky program.

perform interprocedural analysis to remove redundant primitive operations, our compiler can remove redundant domain-specific operations, which typically leads to much greater runtime savings.

Improves the maintenance and portability of applications. The Broadway architecture provides long term benefits in terms of maintenance and portability. The existence of the meta interface allows the Broadway compiler to perform library level optimizations, reducing the application programmer's temptation to perform premature optimizations. By avoiding the Advanced interface, the programmer improves maintenance and portability. For existing libraries, our solution allows the Core and Advanced interfaces to be separated, with the Advanced interface being considered a part of the meta interface. This separation gives the programmer a simpler view of the library. For future libraries, our solution allows library designers to create simpler library interfaces. Thus, as shown in Figure 4, the applications programmer's task is considerably reduced, so the predominant expertise required of the library user is application expertise.

Enhances the value of legacy codes. The annotations are stored separately from the application source code and are not visible to the applications programmer, so our solution applies to existing libraries and existing applications without modification to the vast base of existing source code. Thus, by separating the annotation language from the base interface, the Broadway architecture enhances the value of legacy codes.

Amortizes costs. From the compiler writer's point of view, the Broadway compiler is ideally written once, and this cost is amortized across many different libraries. From the library annotator's point of view, the meta interface is ideally used once to create a set of annotations, and this cost is amortized over the lifetime of the library and across many applications. By contrast, the effort to perform manual library level optimization improves the performance of only a single application.

Provides clean division of labor. Finally, our architecture separates the roles of the compiler writer, the library writer, and the application writer so that each task is simplified. All of the domain-specific expertise is localized in the annotations, which are supplied once by a library expert. The annotation language has been designed to minimize the amount of compiler expertise required to use it. Thus, all of the static analysis and optimization strategies are encapsulated in our Broadway compiler, as specific analyses and optimizations are implicitly configured by the information supplied by the annotations. Together, the annotation language and Broadway compiler free the application programmer to focus on designing clean applications and to resist the temptation to prematurely optimize their source code.

6. RELATED WORK

There has been considerable work in optimizing and customizing software libraries. The related work can be grouped into two categories. The first maintains the traditional library structure as shown in Figure 4, while the second uses a meta interface approach that is similar to ours. Among the meta interface systems, our approach has the advantage of preserving the existing base interface exactly.

Smart Libraries. A number of libraries have been built that attempt to select efficient implementations based on the specific values of input parameters [3, 5, 20]. These libraries provide a restricted degree of customization that is limited to a pre-defined set of implementations.

Automatically Generated Libraries. ATLAS [23], PHiPAC [4], and FFTW [10] have shown that efficient machine-specific libraries can be automatically generated. As with the "smart libraries," these automatically generated libraries preserve the traditional library structure. These approaches address the issue of portability but do not provide a mechanism for customizing libraries for specific clients.

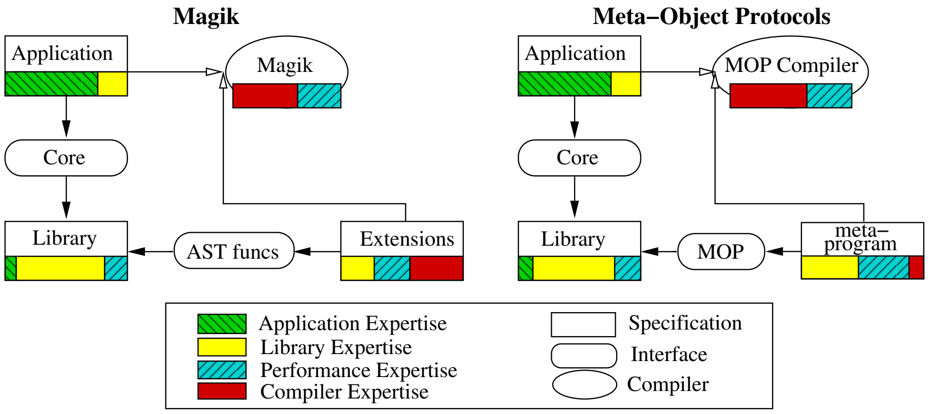


Figure 6 Comparison of Software Architectures

Magik. Engler's Magik system [9] has a structure that is very similar to ours (see Figure 6). Magik gives the programmer access to a C compiler's internal representation and symbol table. Thus, Magik can be used to perform certain compiler transformations, as well as to extend the C language in limited ways. Magik differs significantly from Broadway in two ways. First, Magik theoretically provides more powerful transformational capabilities since it exposes all of the compiler's internals to the meta programmer. However, this power comes at a cost: the meta programmer must possess both compiler expertise and library domain expertise. Second, Magik does not provide the ability to define new domain-specific analyses, which are central to library-level optimizations.

Meta-Object Protocols. The notion of meta interfaces was pioneered in the domain of object oriented languages and Meta-Object Protocols (MOPs) [7]. Like Magik, these systems provide a mechanism to change the way a language is compiled, which provides both optimization and extension capabilities. In comparison to Broadway, MOPs provide more limited support for analysis and transformations. Most MOPs also provide ways to change the syntax of the base language.

Formal Semantics. Vandevoorde [22] defines a system whose structure is almost identical to Broadway's, but whose approach is fundamentally different. Vandevoorde optimizations are based on formal semantics and theorem proving, so the transformations require complete formal semantics of a procedure's behavior, and they depend on theorem proving, which can only be partially automated.

7. CONCLUSION

In this paper we have explained how the lack of a meta interface encourages library designers to produce bloated interfaces. These bloated interfaces in turn create long term portability and maintenance problems. We have shown how the Broadway solution provides a meta interface that yields a desirable division of labor—among the library writer, the compiler writer, and the applications programmer—that is essential in the domain of scientific computing in which high performance is critical and both libraries and applications require a large degree of domain expertise. Finally, we have argued that Broadway's meta interface enhances the use of software libraries and improves the quality of application code.

Acknowledgments. We are happy to thank Brad Chamberlain and E Christopher Lewis for their many helpful comments on the presentation, and we are grateful to Robert van de Geijn for his invaluable help in explaining the innards of PLAPACK.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[2] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: high performance through high level abstractions. In *Proceedings of the International Conference on Parallel Processing*, 1998.

[3] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library. In *Proceedings of Supercomputing '94*, November 1994.

[4] Jeff Bilmes, Krste Asanovic, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[5] Eric A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the Fifth Symposium on Principles of Parallel Programming*, July 1995.

[6] Bradford Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July-September 1998.

[7] S. Chiba. A metaobject protocol for C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 285–299, October 1995.

[8] J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–28, 1990.

[9] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 103–118, October, 1997.

[10] Matteo Frigo and Stephen G. Johnson. An adaptive software architecture for the FFT. In *IEEE Int'l Conference on Acoustics, Speech and Signal Processing, vol 3*, pages volume 3, pp 1381–1384, 1998.

[11] Torbjorn Granlund. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, April 1996.

[12] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.

[13] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Languages and Compilers for Parallel Computing*, August 2000.

[14] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[15] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.

[16] Donald Knuth. Literate programming. *Computer Journal*, 27(2):97–111, May 1984.

[17] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[18] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[20] Anthony Skjellum and Chuck Baldwin. The Multicomputer Toolbox: scalable parallel libraries for large-scale concurrent applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[21] Robert van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.

[22] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, MIT, Department of Electrical Engineering and Computer Science (also MIT/LCS/TR-598), 1994.

[23] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *SC'98*, 1998.