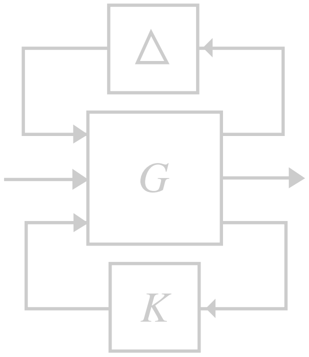

Chapter 1 Customizing Software Libraries for Performance Portability

1.1 Introduction

Software libraries are widely used, particularly in scientific computing, because they provide a convenient method of encapsulating and reusing collections of domain-specific code. Thus, for example, scientific programmers can use linear algebra libraries [7, 8, 18] to leverage the expertise of others. The problem with libraries is that they are typically designed to be general so that they can be reused in as many situations as possible. This generality represents a performance penalty, as there is great benefit to specializing a program for its specific calling contexts. The performance benefit of specialization might seem unimportant since most scientific libraries are designed by experts and carefully coded to be as efficient as possible, but Section 1.2 will show that specialization can improve by several hundred percent the performance of programs written with a high performance parallel dense linear algebra library.

In previous work, we have described the Broadway compiler system, which optimizes the use of software libraries by exploiting library-specific information that is expressed in the form of an annotation language [11, 13]. This paper describes how the Broadway system can be augmented to provide improved performance portability by exploiting a simple form of dynamic optimization that was introduced by Diniz and Rinard [6]. We begin by reviewing the Broadway system and its benefits. We then explain how performance portability can be difficult to achieve for certain parallel library routines. We then briefly describe our proposed approach. Finally, we conclude by contrasting our work with previous research and providing concluding remarks.

1.2 The Broadway Compiler

Figure 1.1 shows our system architecture for performing library-level optimizations [11]. In this approach, annotations capture semantic information about library routines. These annotations are provided by a library expert and placed in a separate file from the source code. This information is read by our compiler, dubbed the Broadway compiler, which performs source-to-source optimizations of both the library and application code. The resulting integrated system of library and application code is then compiled and linked using conventional tools. Our current implementation of the Broadway compiler takes ANSI C as input and produces ANSI C as output.

In addition to supporting the development of new libraries, this architecture is specifically designed to support existing libraries. In particular, by separating the annotations from the library source, our approach applies to existing libraries and existing library applications.

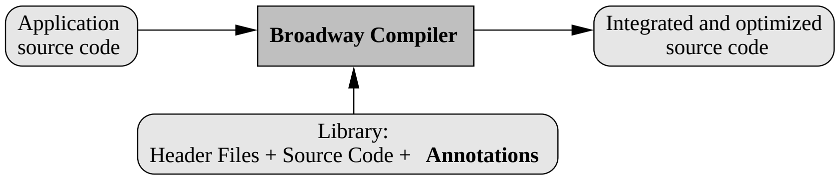
Application source code
Broadway Compiler
Integrated and optimized source code
Library:
Header Files + Source Code + Annotations

Figure 1.1. *Architecture of the Broadway Compiler system*

The annotations describe the library only, and not the application. While information about the application would certainly be useful, this restriction makes the system more usable, as applications programmers do not need to learn the annotation language. In fact, the annotations can be completely hidden from the library user, who only needs to compile with the Broadway compiler instead of a standard C compiler.

Moreover, there are several reasons why it is more beneficial to describe library information rather than application information. First, as mentioned in the Introduction, libraries are built to be general, but there is great benefit to specializing them for specific contexts. Applications, on the other hand, are typically not as concerned with generality. Second, libraries are mechanisms for reuse, so the cost of creating annotations for libraries can be amortized over many uses of the library. Third, libraries typically encapsulate a coherent set of domain-specific abstractions, which increases the likelihood that a small set of annotations can describe a useful set of information. Finally, libraries typically embody a rich amount of domain-specific knowledge, and these annotations encapsulate and exploit information that library writers already know and that is otherwise wasted.

Philosophically, our architecture attempts to provide a clean separation of concerns among the compiler writer, the library writer, and the applications programmer. The compiler encapsulates all compiler analysis and optimization machinery, but does not include any library-specific information or library-specific optimizations. Thus, the compiler is built to be as general as possible and is only configured for specific libraries through the annotation language. By contrast, the annotations describe library knowledge and domain expertise, but do not require deep compiler expertise to create. This separation of compiler expertise and library expertise is critical, because it is unreasonable to expect anyone to possess both types of expertise. Finally, the annotations and compiler together free the applications programmer to focus on application design rather than on performing manual library-level optimizations [12].

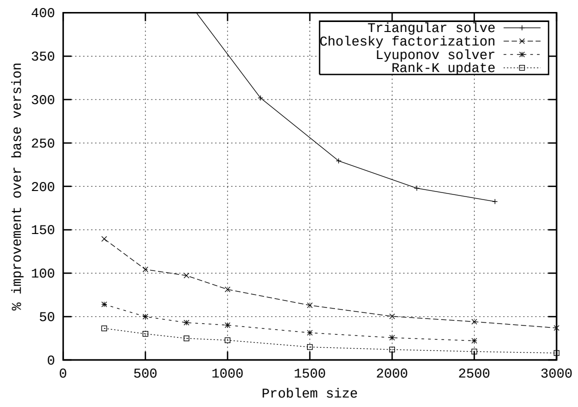

Figure 1.2. *Annotation-based optimizations improve PLAPACK parallel programs from 10% to 180% for large problems, and from 36% to 622% for small problems.*

Figure 1.2 shows the results of applying our techniques manually to four programs written with the PLAPACK parallel dense linear algebra library [18]. We see that significant performance improvements were obtained. For example, the lowest curve (rank-k update) indicates a performance improvement of 10% for large problem sizes and 180% for small problem sizes. The highest curve (triangular solve with multiple right hand sides) shows a performance improvement of 36% for large problem sizes and 622% for small problem sizes. In these cases, small problem sizes benefit more because the specializations tend to remove overhead. This overhead, such as communication, is significant because it limits, for a fixed problem size, scalability to large numbers of processors. In summary, these results illustrate the benefit of specializing library routines for specific calling contexts, even for libraries like PLAPACK that have been carefully designed to provide high performance.

1.2.1 Library-Level Optimizations

We now describe an example of the type of optimization that was used to produce the results shown in Figure 1.2. To understand the optimizations, we need to first understand that PLAPACK programs manipulate matrices through objects known as *views*, which represent the indices of a submatrix. PLAPACK thus provides routines for creating and manipulating views. During the course of a program's execution, these views can assume different properties. In the most general case, a view represents a matrix that is distributed across multiple processors. In some cases a view resides wholly on a single processor, in which case the view is said to be `local`, and in other cases the view represents the empty matrix, in which case we say the view is `empty`.

These properties of views are significant because they can be used to improve performance. Application programs typically invoke PLAPACK routines that work on any matrix view, as this greatly simplifies the programming, but routines are available that operate on specific types of views. For example, `PLA_Gemm()` performs matrix multiplication and makes no assumptions about the input matrices' views, but `PLA_Local_Gemm()` works only if the views are `local`. `PLA_Local_Gemm()` is the more efficient of the two routines, because it does not include any of the overheads required to deal with parallel objects. When a view is known to be `empty` even greater savings can be obtained, as many PLAPACK calls on `empty` views simply become no-ops.

One optimization, then, is to determine at each call site whether a view has one of the special properties, and if so to substitute the invocation of the general PLAPACK routine for an invocation of the appropriate specialized routine. This optimization requires a dataflow analysis that tracks the properties of views throughout the execution of the program. Our annotation language supports this type of library-specific analysis by allowing a library expert to define properties on views as follows:

```
property Distribution : map_of< object, {general, local, empty} >;
```

This annotation declares a property of matrices (a flow value in dataflow analysis terms [16]) that has three values: `general`, `local`, and `empty`. Other annotations then describe transfer functions [16] that describe how the various PLAPACK routines affect the properties of views. That is, the transfer functions indicate which routines create views, which ones shrink them, and so forth. Collectively, these annotations configure the Broadway compiler to perform a dataflow analysis on views. Once the analysis is performed, a third type of annotation indicates how the library invocations can be specialized by through pattern matching:

```
pattern PLA_Gemm(...)
{
   when ((Distribution[viewA] == Local) &&
         (Distribution[viewB] == Local) &&
         (Distribution[viewC] == Local))
   replace { PLA_Local_Gemm(...)
   ...
}
```

In these examples we elide details due to space limitations, but the full annotation language is described elsewhere [13].

The key points from this example are that library-specific analyses are needed to exploit library-specific abstractions such as views, and that our annotation language provides a mechanism for describing such analyses and abstractions.

1.3 Optimizing for Performance Portability

This section describes how the Broadway system can be extended to provide improved performance portability. We first argue that many optimizations on parallel programs require information that is difficult to obtain statically. We then explain the notion of dynamic feedback [6] and show how we can support this notion with annotations.

1.3.1 Classes of Optimizations

Broadway optimizations can be classified into three categories: (1) optimizations that the compiler assumes will always improve performance; (2) otimizations that apply to particular target machines but not to others; and (3) optimizations whose benefit depends on interactions between the application and the target machine, possibly requiring run-time information. Broadway's annotation language currently supports the first two types of annotations. In particular, class (2) machine-specific optimization can be turned on or off by manually including or excluding the relevant annotations for a particular target machine. New mechanisms are needed, however, to support class (3) optimizations and to provide an added degree of performance portability.

Many optimizations fall into class (3), including two types of optimizations that are fundamental to parallelism: optimizations that deal with granularity of parallelism, and optimizations that deal with the degree of parallelism.

Most parallel programs must be tuned for their granularity of parallelism. For example, the granularity of PLAPACK programs is guided by the concept of a *block size*. Larger block sizes provide larger units of communication, which reduces the overhead of communication. Smaller block sizes, however, tend to produce better load balance. Thus, there is a machine-specific tradeoff. Moreover, the choice of block size can interact with the algorithm, so the issue of granularity is sensitive to the communication characteristics of the target machine, as well as to application-specific and algorithmic characteristics.

The degree of parallelism is also machine-specific. For any given computation, there is a tradeoff between computing it sequentially or computing it in parallel. The parallel approach has the benefit of splitting the task across multiple processors, which will ideally decrease the computation time. However, the parallel approach typically requires communication that is not required for a sequential computation, so parallelism is not always a win. With PLAPACK, an application programmer can often choose to distribute a computation across all processors at the expense of added communication, or to perform the computation on a subset of processors at the cost of increased load imbalance. This tradeoff is tightly coupled to the program's granularity. For example, it is probably better to choose full parallelism when there is a large amount of work to do, while it is better to compute on the subset of processors if the amount of work is small compared to the overhead of re-distributing the work. This tradeoff can be complex. Worse, this tradeoff can change dynamically as the amount of work to perform often varies as the algorithm progresses.

1.3.2 Dynamic Feedback

The idea of dynamic feedback is simple. When optimization decisions cannot be made statically due to lack of information, the compiler creates multiple versions of the code and uses dynamic sampling to determine which is best. The best code is then executed for some duration, known as the production phase, which is typically much longer than the sampling period. To support situations where the relative performance of the different versions can vary dynamically, this cycle is repeated until the program completes.

Minor modifications to our annotation language can be made to support dynamic feedback. In particular, the `select` keyword can be introduced to instruct the compiler that multiple optimizations are possible for different situations. In the following example, `select` is used to indicate that there are three ways to specialize a Broadcast operation in MPI [10].

```
pattern {
   MPI_Bcast(...);
}
{
  when (Distribution[A] == ColumnPanel)
    select {
              { /* Bucket implementation */ ...}
              { /* MST implementation */ ...}
              { /* Scatter-gather */ ...}
           }
}
```

Annotations can also be used to guide policy decisions, such as how long the sample periods should be and how long the production period should be. For example, the following annotation fragment indicates that samples should be taken every 20 times that the routine is called.

```
    select every 20 { ...
    }
```

Annotations can be used to customize the feedback process in more sophisticated ways by indicating how the adaptivity depends upon various aspects of the library implementation. For example, the following annotation states that the adaptivity depends on the value of the program variable `blocksize`, which tells the compiler that adaptivity is not needed in areas where the value of `blocksize` does not change.

```
    select on blocksize { ...
    }
```

The Broadway compiler system greatly simplifies the production of dynamically adaptive library routines. The library implementation need not change. Instead, the annotations that describe them change, and these changes only express a few key bits of information. The remaining details are hidden in the Broadway compiler, which creates and optimizes the various code versions, and which inserts code into the application to sample the different versions and to select the most efficient version.

1.4 Related Work

Our research extends to libraries a considerable body of previous work in dynamic optimizations [6], partial evaluation [2, 4], abstract interpretation [5, 14], and pattern matching [17]. This section relates our work to other efforts that provide configurable compilation technology.

The Genesis optimizer generator produces a compiler optimization pass from a declarative specification of the optimization [20]. Like Broadway, the specification uses patterns, conditions and actions. However, Genesis targets classical loop optimizations for parallelization, so it provides no way to define new program analyses. Conversely, the PAG system is a completely configurable program analyzer [15] that uses an ML-like language to specify the flow value lattices and transfer functions. While powerful, the specification is low-level and requires an intimate knowledge of the underlying mathematics. It does not include support for actual optimizations.

Some compilers provide special support for specific libraries. For example, *semantic expansion* has been used to optimize complex number and array libraries, essentially extending the language to include these libraries [1]. Similarly, some C compilers recognize calls to `malloc()` when performing pointer analysis. Our goal is to provide configurable compiler support that can apply to many libraries, not just a favored few.

Meta-programming systems such as meta-object protocols [3], programmable syntax macros [19], and the Magik compiler [9], can be used to create customized library implementations, as well as to extend language semantics and syntax. While these techniques can be quite powerful, they require users to manipulate AST's and other compiler internals directly and with little dataflow information.

1.5 Conclusions

Software libraries are designed for semantic reuse and semantic portability, but not for performance portability. This paper has explained how the Broadway compiler framework can be extended to use dynamic optimizations to provide improved performance portability. In particular, a simple mechanism of dynamic feedback [6] allows multiple versions of optimized code to be dynamically selected. We have explained why this approach is an ideal extension of annotation-based optimization. Furthermore, the necessary extensions to our annotation language are minimal. We are currently conducting experiments to quantify the benefits of our proposed idea, and we are in the process of completing our compiler implementation so that we can obtain fully automated results.

Acknowledgments. 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

Bibliography

[1] P. Artigas, M. Gupta, S. Midkiff, and J. Moreira. High performance numerical computing in Java: language and compiler issues. In *Workshop on Languages and Compilers for Parallel Computing*, 1999.

[2] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer*, 23(12):23–37, December 1990.

[3] 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.

[4] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, 1993.

[5] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[6] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–84, 1997.

[7] 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.

[8] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[9] D. 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] M. P. I. Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

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

[12] S. Z. Guyer and C. Lin. Broadway: A software architecture for scientific computing. In *IFIPS Working Group 2.5: Working Conference on Software Architectures for Scientific Computing Applications*, October 2000.

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

[14] N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science.* Oxford University Press, 1994. 527–629.

[15] F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[16] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kauffman, San Francico, CA, 1997.

[17] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

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

[19] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

[20] D. Whitfield and M. L. Soffa. Automatic generation of global optimizers. *ACM SIGPLAN Notices*, 26(6):120–129, June 1991.