An Annotation Language for Optimizing Software Libraries*

Samuel Z. Guyer
The University of Texas at Austin
sammy@cs.utexas.edu, http://www.cs.utexas.edu/users/sammy

Calvin Lin
The University of Texas at Austin
lin@cs.utexas.edu, http://www.cs.utexas.edu/users/lin

Abstract

This paper introduces an annotation language and a compiler that together can customize a library implementation for specific application needs. Our approach is distinguished by its ability to exploit high level, domain-specific information in the customization process. In particular, the annotations provide semantic information that enables our compiler to analyze and optimize library operations as if they were primitives of a domain-specific language. Thus, our approach yields many of the performance benefits of domain-specific languages, without the effort of developing a new compiler for each domain.

This paper presents the annotation language, describes its role in optimization, and illustrates the benefits of the overall approach. Using a partially implemented compiler, we show how our system can significantly improve the performance of two applications written using the PLAPACK parallel linear algebra library.

1 Introduction

Software libraries are a common mechanism for re-using code. Like a domain-specific language, libraries can provide high-level abstractions that empower the programmer and hide implementation details. Unlike a domain-specific language, libraries do not introduce new syntax and receive no direct support from the compiler. These differences have two consequences:

- **Compilers have limited ability to improve performance.** Compilers cannot exploit the domain-specific information that is only implicitly encoded in a library's implementation. Thus, many opportunities for optimization are lost. Since library code is written, compiled and optimized in isolation, such optimizations are important as a means of customizing a library implementation for different application needs.
- **Performance improvements are exposed through the interface.** The only way to offer both generality and performance is to provide wide interfaces with specialized routines for different contexts. Unfortunately, these specialized routines are typically more difficult to use correctly. Moreover, the specialized routines typically improve performance by exposing implementation decisions. Thus, they intertwine the interface and the implementation, which inhibits code reuse in the long run.

Our approach to mitigating these problems is to give libraries some of the compiler support enjoyed by domain-specific languages. The key is an annotation language that captures expert knowledge about libraries and enables our compiler to customize library implementations for different situations. Library users can then focus on application design, relying on our compiler to optimize performance.

Figure 1 shows the overall architecture of our system. The annotations are supplied by a library expert in a separate specification file that accompanies the usual header files and source code. The annotations convey two kinds of information about library routines: (1) basic dataflow information, which is sometimes difficult to obtain through static analysis, and (2) high level domain-specific information. Our compiler, which we

*This work was supported in part by an NSF Research Infrastructure Award CDA-9624082, NSF grant CCR–9707056, and ONR grant N00014-99-1-0402.

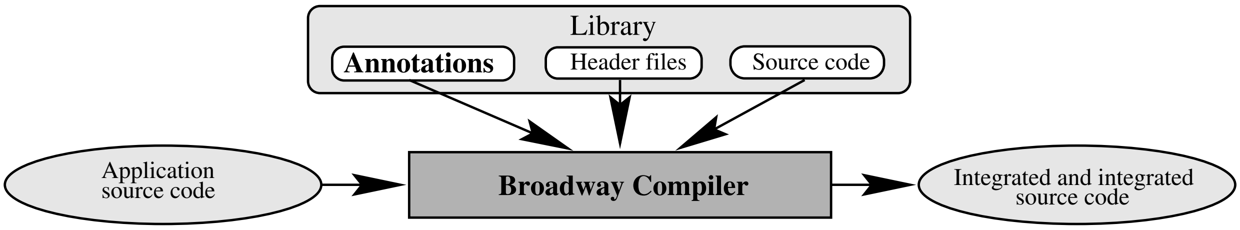


Figure 1: Architecture of the Broadway Compiler system

have named the Broadway compiler, reads the annotations and applies a series of source-to-source transformations to both the library and application source. The result is an integrated system of library and application code, which is ready to be compiled and linked using conventional tools.

Our system offers many practical benefits. First, the annotations are specified in a separate file from the library source, so our approach applies to existing libraries and existing applications. Second, the annotations describe the library, not the application, so the application programmer does nothing more than use the Broadway Compiler in place of a standard C compiler. Finally, the non-trivial cost of writing the library annotations can be amortized over many applications.

When applied to a Cholesky factorization program that uses the PLAPACK parallel linear algebra library [25], our system improves performance by 26% for large matrices and 195% for small matrices. Both the library and application are written in ANSI C, and neither has been modified to facilitate our results. This paper will explain how our solution is able to obtain these results. While these same optimizations could be performed manually by using a wider interface and expert knowledge of the PLAPACK implementation, our approach offers significant advantages:

- Both approaches require semantic expertise about the PLAPACK implementation, but manual optimization embeds this knowledge implicitly in the optimized program, while our annotations encapsulate such knowledge for use in optimizing other PLAPACK applications.

- Manual optimization is feasible only for PLAPACK experts. By contrast, once an expert has provided annotations, even casual users can optimize their PLAPACK applications by invoking our compiler.

- Manual optimization directly modifies the source code, which complicates subsequent modification, reuse and maintenance. Our annotations instead provide a clean separation of the optimization information from the basic implementation.

- The explicit representation of semantic information allows it to be checked for correctness. This is an open issue which we leave as future work.

The performance improvements mentioned above cannot be obtained with conventional compiler technology because the optimizations require semantic information about the PLAPACK implementation that cannot be derived automatically. For example, one transformation requires knowledge of a PLAPACK object's data distribution. This information is implicitly represented in the values of four object attributes and the value of a global variable. To further complicate matters, PLAPACK is written in C and is difficult to analyze because of its pervasive use of pointers. In addition, certain def/use information is impossible to obtain because PLAPACK makes calls to the sequential BLAS library [11], whose source code is unavailable.

The primary contributions of this paper are (1) the introduction of a new technique for optimizing software libraries, (2) the demonstration that this technique provides performance benefits when applied to a production-quality library, and (3) an evaluation of our annotation language based on experiments with PLAPACK applications.

This paper is organized as follows. Section 2 contrasts our work with related efforts. Section 3 describes our annotation language and its design philosophy. Section 4 explains our compilation strategy, and Section 5 offers an empirical evaluation of our language. Finally, we and draw conclusions and discuss future work.

2 Related Work

Our work builds upon the tremendous amount of previous research in program analysis and program transformations. In particular, we attempt to extend classical analyses and transformations to semantically higher level operations that are encapsulated in library functions. For example, one of our annotations annotation specifies an abstract interpration [9, 17] and another draws from the pointer analysis work of Wilson and Lam [28].

Compilers have long used hints and pragmas to guide optimizations such as register allocation and inlining, and to summarize procedure information such as whether a function has side effects. More recently, annotations have been used to guide dynamic compilation [13]. While annotations are not new, our use of them is new. First, our annotations describe function implementations, rather than call site-specific information. This means that application programs do not require annotations, so our annotations are hidden from the everyday user. Second, and more fundamentally, our advanced annotations can convey domain-specific information that other languages cannot. For example, annotators can define concepts, such as data distribution, that extend beyond those of the base language. However, unlike most hints and pragmas, the incorrect use of our annotations can lead to transformations that do not preserve the library's semantics.

Our work is closely related to partial evaluation [5, 6, 10], which improves performance by specializing routines for specific inputs. Partial evaluation combines inlining, constant propagation and constant folding to evaluate as much of the program as possible at compile time. Recent work in program specialization has generalized partial evaluation to the notion of staged optimizations, which can take place at compile time, link time or runtime [13, 14], and which can be applied to class libraries in object oriented programs [27]. All of these approaches specialize based on values of variables that are constant for some duration of the program. By contrast, our approach can specialize based on other criteria: For example, specialization can occur at a particular program point when the program moves into a particular program state. Our approach also can perform optimizations such as loop-invariant code motion that cannot be expressed using partial evaluation.

Software generators [23, 24] and program transformation systems [22] are compilers for domain-specific programming languages. While these systems provide sophisticated transformations of high level language constructs, they typically manipulate programs only at the syntactic level. Semantic properties, such as those resulting from dataflow analysis, are either awkward to express or completely unavailable. Our approach instead focuses on the exploitation of semantic, rather than syntactic, information.

There has been considerable work in formal semantics and formal specifications. In particular, Vandevoorde uses powerful analysis and inference capabilities to specialize procedure implementations [26]. However, complete axiomatic theories are difficult to write and do not exist for many domains. In addition, this approach depends on theorem provers, which are computationally intensive and only partially automated. Our work differs from these primarily in the scope and completeness of our annotations, which describe only specific implementation properties instead of complete behaviors.

Open and extensible compilers give the programmer complete access to the internal representation of the program [16, 12]. While these systems are quite general, they impose a considerable burden. To use them, the programmer needs to understand (1) general compiler implementation techniques, (2) how to configure the specific compiler they are using, and (3) how to express and execute their optimizations. Similarly, meta-object protocols provide sophisticated mechanisms for modifying the compilation of object oriented programs [8, 19], but they can be difficult to use. Our compiler limits configurability to a small but powerful set of capabilities, and provides a simple way to access them.

Finally, we note that our system is an instance of aspect-oriented programming [18]. In our case, the cross-cutting aspect is performance improvement, and our annotation language and compiler are specific mechanisms for implementing this aspect. An important feature of aspects is that they be separated from the rest of the code, and in our case this is achieved by placing the annotations in a separate file.

3 Annotation Language

The goal of the annotation language is to convey library-specific information to the compiler in a simple declarative manner. While it's clear that more sophisticated specifications could support more sophisticated optimizations, our goal is to show that a few simple annotations can enable many useful optimizations. Simplicity is important because we expect our language users to be library experts who do not necessarily have expertise in compilers or formal specifications.

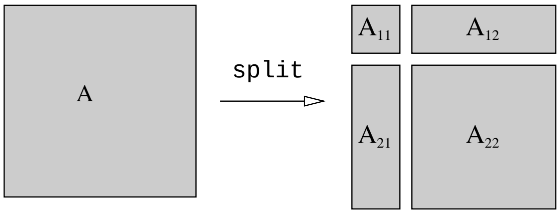
In designing the language, we studied several libraries to determine the most useful ways of optimizing them. We noticed that library operations could easily be integrated into many traditional optimizations, such as dead-code elimination, copy propagation and loop-invariant code motion. These optimizations are effective and well understood, and they require only minimal information to enable. For example, to enable loop invariant code motion, the annotations need to indicate which library procedures have no side-effects. We also observed that many library-specific optimizations replace a general-purpose library call with a more specific one that takes advantage of information about the calling context. This form of specialization not only improves performance, it often creates additional opportunities for traditional optimizations. Thus, our annotation language consists of two classes of annotations: *basic annotations* for enabling traditional optimizations, and *advanced annotations* for specifying library-specific specialization.

We present the annotation language by first describing the target library: the PLAPACK parallel linear algebra library [25]. The remainder of the section then describes the language constructs in detail, using a fragment of the PLAPACK annotations as a source of examples. These annotations capture the information used to produce the results in Section 5. A complete grammar is presented in the appendix.

3.1 The PLAPACK library

PLAPACK is a production-quality library for coding parallel linear algebra algorithms in C. It consists of approximately 40,000 lines of C code and provides parallel versions of the same kernel routines found in the BLAS [11] and LAPACK [2]. At the highest level, it provides an interface that hides much of the parallelism from the programmer.

A PLAPACK application operates on linear algebra objects, such as matrices and vectors, that are partitioned and distributed over the processors of the target computer. The application manipulates these objects indirectly though handles called *views*. A view specifies an index range that selects some or all of a distributed object for subsequent computations. PLAPACK contains routines to create new views, shift views, and split views into pieces. The following figure shows a four-way split that logically divides a matrix into four smaller ones:



A typical algorithm starts with an entire object, like A, and splits it into manageable pieces. It computes directly on A_{11}, A_{12} and A_{21}, and then continues recursively by splitting the large remaining piece, A_{22}, until the entire data set has been visited.

Often, a view captures part of a matrix or vector that has special properties. Understanding and exploiting these properties can lead to significant performance improvements. For example, a view can select a region that resides entirely on one processor. Any computations on the data within this *local* view can be performed locally, without involving other processors. In the figure above, the four-way split yields one local view (A_{11}), one *column panel* (A_{21}), which resides on a column of processors, one *row panel* (A_{12}), which resides on a row of processors, and a large fully-distributed matrix (A_{22}). A view might also specify a region that is in tridiagonal form, allowing the use of specialized compute functions.

Our goal is to identify the library-specific properties that are relevant to optimization, and track them through the application program. For example, if an application splits a local view into two pieces, we can infer that the two new views are also local. The result of this analysis describes how the application manipulates objects with respect to library-specific properties such as distribution or data content. We can use this information to customize the library, or to select library routines that are better suited to the application.

Figure 2 shows a fragment of the annotations for PLAPACK. It specifies the two properties described above (distribution and data content) and gives the semantics of three PLAPACK routines: `PLA_Matrix_create`, which creates a new matrix, `PLA_Obj_vert_split_2` which splits a view into left and right pieces, and `PLA_Gemm`, which multiplies matrices.[1]

[1]In this figure, the `PLA_Gemm` interface has been simplified in insignificant ways to clarify the presentation. The actual routine accepts seven arguments instead of three.

```
     %{
(1)  #include "PLA.h"
     %}

     // --- Special case matrix distributions
     property  Distribution = { General = none, ColPanel, RowPanel,
(2)                               Local, Empty };
     // --- Special case properties of the data
     property  Contents  = { Dense = none, Zero, Identity, Upper, Lower };

     // --- Procedure: Create a new distributed matrix

     procedure PLA_Matrix_create  ( datatype, length, width, template,
                                    align_row, align_col, new_matrix)
     {
       on_exit { new_matrix --> view_1,
(3)              DATA of view_1 --> data_1 }
       access { datatype, length, width, template, align_row, align_col }
       modify { new_matrix }

       analyze Distribution  {  view_1 = General; }
     }

     // --- Procedure: Split a matrix logically into two pieces

     procedure PLA_Obj_vert_split_2( obj, length, left, right)
     {
       on_entry { obj   --> view_1, DATA of view_1 --> data_1 }
(4)    on_exit  { left  --> view_L, DATA of view_L --> data_1,
                  right --> view_R, DATA of view_R --> data_1 }
       access { view_1, length }
       analyze Distribution {
         (view_1 == General)   =>  view_L = ColPanel, view_R = General;
         (view_1 == ColPanel)  =>  view_L = ColPanel, view_R = Empty;
(5)      (view_1 == RowPanel)  =>  view_L = Local,    view_R = RowPanel;
       }
       specialize {
         (view_1 Distribution == ColPanel)  => replace "PLA_Copy_view(obj, view_L)";
       }
     }

     // --- Procedure: Compute C <- A * B

     procedure PLA_Gemm( A, B, C)
     { on_entry { A --> view_A, DATA of view_A --> data_A,
                  B --> view_B, DATA of view_B --> data_B,
                  C --> view_C, DATA of view_C --> data_C }
       access { data_A, data_B }
       modify { data_C }

       analyze Contents {
         ((data_A == Upper) && (data_B == Upper)) =>  data_C == Upper;
         ((data_A == Zero)  || (data_B == Zero))  =>  data_C == Zero;
       }
       specialize  {
         ((view_A Distribution == Empty) ||
          (view_B Distribution == Empty))    =>  remove;
         ((view_A Distribution == Local) &&
          (view_B Distribution == Local))    =>  replace "PLA_Local_gemm( A, B, C)";
         (view_A Contents == Upper)          =>  replace "PLA_Trmm( A, B, C)";
       }
     }
```

Figure 2: Part of the annotations for the PLAPACK parallel linear algebra library. (1) The header provides access to definitions in the library header files. (2) The `property` annotations define abstract object states which are used for analysis and specialization. (3) Each library procedure has its own set of annotations. (4) The basic annotations summarize the dataflow and pointer behavior of the procedure. (5) The advanced annotations specify analysis rules for abstract interpretation and specialization rules that use the resulting information.

3.2 Basic annotations

Each library procedure can have a set of basic annotations that provides the information needed to support the Broadway compiler's dataflow analysis framework. This information allows the compiler to properly interpret library calls, and to integrate them into traditional optimization passes such as code motion, copy propagation and redundancy elimination.

A library procedure has access to many different data objects in the application program, including the arguments passed into it, and possibly global objects as well. In addition, many libraries create and manage complex pointer-based data-structures that are built up from many objects. We have found that in order to correctly analyze library calls, it is essential to accurately model these data-structures. Thus, the basic annotations provide two kinds of information: (1) a list of the objects that are accessible to the procedure and describe their structure, and (2) a list of those objects whose contents are accessed or modified by the procedure (the "uses" and "defs").

The information is specified using a technique similar to interval analysis [20]. Interval analysis concisely summarizes the effects of a procedure, so that the compiler can analyze any code that calls the procedure without reanalyzing the procedure itself. Our language allows the library annotator to explicitly summarize the dataflow and pointer effects for each library procedure [28]. In some cases, a modern compiler could derive this information automatically from the library source. However, there are conditions under which this is infeasible or impossible. Many libraries encapsulate functionality for which no source code is available, such as low-level I/O or communication routines. Even if source is available, it may be simpler to provide the information declaratively, especially if it is well known.

3.2.1 `on_entry` and `on_exit`

The `on_entry` and `on_exit` annotations specify the effects of a library procedure on objects that are organized into data structures. We model data structures by adding edges between the objects. The edges are directed and can be roughly interpreted as "points to". Each identifier in these annotations is either an input to the procedure (a formal parameter), or gives a name to an object that is reachable by following edges from an input. Like the formal parameters, each name is arbitrary and is bound to actual objects at each procedure call site. The behavior of the procedure is summarized by showing the configuration before and after execution.

The `-->` operator indicates that the operand on the left points to the operand on the right. For example, in PLAPACK, each matrix parameter is passed as a pointer to a view structure, which in turn points to the underlying data. We can label an edge by providing an additional identifier followed by the`of` keyword. In the example, we label each edge from a view to its data with the label `DATA`. This distinguishes it from any other things that a view might point to. Figure 3 depicts the pointer structure given by the annotations labeled **(4)** in Figure 2.

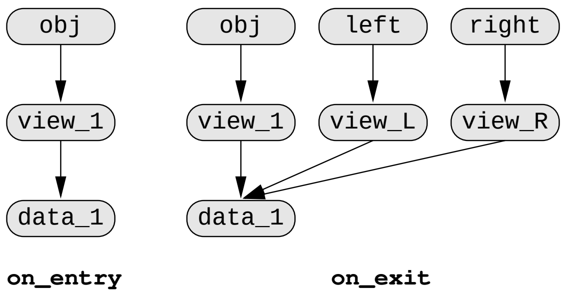


Figure 3: The effect of split on PLAPACK data structures.

We can use the keyword `null` on the right side to indicate the removal of an edge.

The data structures described in these annotations need not correspond exactly to the underlying implementation. In fact, it is often useful to make explicit some of the relationships that are only represented implicitly in the implementation. Many libraries contain objects that behave logically like pointers, such as handles, references and descriptors. We can use `on_entry` and `on_exit` to model all of these structures.

In addition to establishing new data structures, the `on_exit` annotation can declare that an object is a copy of another object, using the `copyof` keyword. We can exploit this information to perform high-level copy propagation on library objects.

3.2.2 `access` and `modify`

The `access` and `modify` annotations list the objects that are accessed or modified by the library procedure. The lists may contain formal parameters from the procedure input list, or object names introduced by the `on_entry` and `on_exit` annotations.

3.2.3 `global`

The `global` annotation declares global variables that can be analyzed along with the procedure parameters. These annotations simply provide a list of names that can be used to track global state information, and are not associated with a specific procedure. Like the `on_entry` and `on_exit` annotations, they need not correspond to actual global variables in the implementation. It is often useful to define global variables that model system states not explicitly represented by variables in the program. As examples, a global variable annotation can be used to track whether a library is properly initialized, or to maintain a record of outstanding asynchronous operations.

3.3 Advanced annotations

The advanced annotations define library-specific analyses and optimizations. The annotations are used to define a dataflow analysis problem consisting of a set of abstract object states and the effects of each library procedure on those states. The abstract states form a dataflow lattice and the library procedure effects serve as dataflow transfer functions. The analyzer propagates this information through the program to derive the abstract states of the actual program variables. A separate set of annotations uses this information to trigger library procedure specializations. Each specialization tests the abstract states of its input parameters to determine if the library call can be replaced by code that takes advantage of the context.

3.3.1 `property`

Each `property` annotation defines an abstract interpretation over objects in the program. The set of abstract values given in the curly braces form a two-level dataflow lattice. Figure 4 shows the lattice specified by the `Distribution` property given in Figure 2.

Because the lattices are only two levels high, whenever two program paths disagree on the state of an object the resulting *meet* results in lattice value $\perp$. We are considering ways to allow more complex lattices, such as multiple level lattices or infinite lattices, while still ensuring convergence. The keyword `none` allows a symbolic name to be assigned to the value $\perp$.

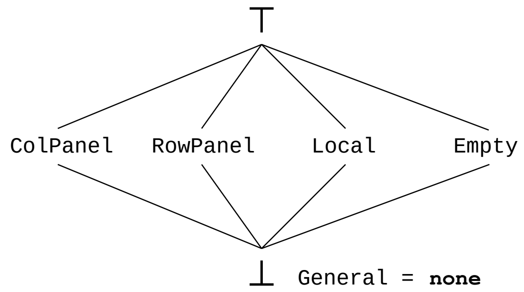


Figure 4: Latticed defined by the `Distribution` property.

3.3.2 `analyze`

Each library procedure can have a set of `analyze` annotations that describe how that procedure affects the properties of the objects it manipulates. Collectively, these annotations compose the dataflow transfer function for each abstract interpretation. Each statement in this annotation behaves as a logical implication: if the conditions on the left of the `=>` operator are true, then we conclude that the facts on the right are true. Each term in the condition is limited to testing the current property value of an object, or comparing to a constant. Each condition is a logic expression made up of these terms. In the absence of the `=>` operator, the facts are assumed without condition.

The PLAPACK annotations in Figure 2 show several examples of the `analyze` annotation. The part labeled **(5)** describes the effect of a vertical split on the distribution of various input view types. The matrix multiply procedure, `PLA_Gemm`, analyzes the contents of the matrices involved. For example, it expresses the fact that multiplying two upper-triangular matrices yields an upper-triangular matrix as a result.

When more than one analysis statement applies, the most specific one is chosen: the statement with the greatest number of conditions that are true, minus any that are false. For example, given an `analyze` annotation of the follow form:

```
analyze Foo {
  (A)      => C1;
  (A && B) => C2;
  (A || B) => C3;
}
```

If only `A` is true, then we would conclude `C1`. If both `A` and `B` are true, then we choose either `C2` or `C3`. Ties are broken by preferring the statement that occurs earlier in the annotation.

3.3.3 `specialize`

Each procedure can specify a set of specializations that is triggered by the properties assigned to the input objects. The specializations modify the call site in the application code. Like the `analyze` annotations, each specialization is guarded by a condition, but these conditions are evaluated after abstract interpretation is complete. Unlike the `analyze` annotations, these conditions can refer to any combination of properties, and thus must provide the specific property name. The right side of the `=>` specifies either a literal code replacement, indicated by the `replace` keyword, or that the library call should simply be removed as indicated by the `remove` keyword.

The PLAPACK annotations in Figure 2 show three specializations for the matrix multiply procedure. The first causes the call to be removed whenever either of the inputs `A` or `B` refers to empty views. The second replaces the parallel matrix multiply routine with a local version if both `A` and `B` refer to local views. Finally, if the data indexed by `A` is upper-triangular, we can replace the general matrix multiply call with a call to a special triangular form that requires half the number of floating point operations.

4 The Broadway Compiler

This section describes the compiler's overall optimization strategy. The compiler consists mostly of traditional analysis and optimization algorithms, extended to use information from our annotation language. The individual transformations are straightforward and are not discussed. During a particular pass, the compiler refers to the annotations to find the information needed. Figure 5 shows the internal structure of the compiler and how the annotations are incorporated. We use a particular ordering of the passes that provides the most information for specialization, and then cleans up the customized code using traditional optimizations.

Pointer analysis. The first phase of the compiler performs pointer analysis. It not only tracks pointers in the application code, but also uses the `on_entry` and `on_exit` annotations to determine the data structures manipulated by the library calls. Our pointer analysis algorithm builds a flow-sensitive "points-to" graph using the strategy described by Chase, et al [7].

Abstract interpretation. The second phase solves the analysis problems specified by the `property` annotations. The analysis framework assigns an abstract state to each object in the program and uses the `analyze` annotations to propagate this information through the program.

Enabling transformations. Dataflow analysis often loses interesting information because it acts conservatively with respect to control flow. For example, if a library procedure is used in two different ways, the analyzer will attempt to unify the information from both contexts. Thus, in the third phase the compiler uses any loss of information as a heuristic to drive enabling transformations, such as procedure integration, procedure cloning, loop peeling and node splitting. Since the properties are used to trigger specializations, using them to trigger these transformations is likely to enable many more specializations.

Specialization. In the fourth phase, the compiler uses the results of analysis along with `specialize` annotations to perform code customization. At each call site, the compiler looks for a specialization that matches the state of the variables. If a match is found, the call site is replaced. We have found that after specialization, it is often beneficial to repeat the abstract interpretation phase because the program modifications reveal new opportunities for optimization.

Traditional optimizations. Specialization often enables many opportunities for traditional optimizations. When a general library call is replaced by a special-case call, any arguments that are no longer used become candidates for dead-code elimination. Similarly, inlining a library procedure often reveals redundant computations and unnecessary copies of objects. Thus, in the final phase, we iterate over a small group of traditional optimization passes until no more improvements can be made.

The traditional optimization passes are extended to include library procedures. The basic annotations make this possible by providing the necessary information. During copy propagation, the `copyof` terms tell the compiler when copies of objects are created, and the `modify` annotation tells the compiler when those copies become invalid. Similarly,

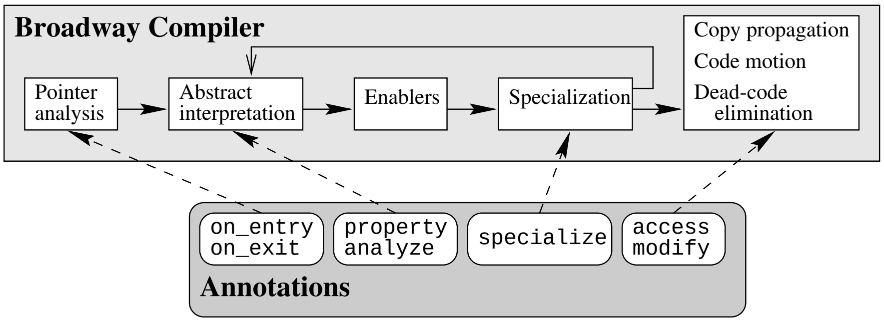


Figure 5: Annotations are incorporated into each phase of the compilation process.

the basic annotations indicate the lifetimes of the objects, allowing the dead-code elimination pass to properly identify dead library calls.

5 Results with PLAPACK

This section describes our experiences in applying our system to portions of two PLAPACK applications, a Cholesky factorization program and a code for solving Lyapunov equations [4].

For these experiments, our compiler performs all analysis automatically. Except for inlining, we perform the transformations manually according to the strategy described in Section 4. While our compiler is not yet complete, the individual transformations are all well-understood. Since the analysis and the overall compilation strategy are the enabling technologies behind these results, our manual transformations should not affect the results. The PLAPACK annotations were written by a person who is not a member of the PLAPACK implementation team. For purposes of comparison, the baseline programs were supplied by the PLAPACK group and written using the cleanest PLAPACK interface. The hand-optimized programs were written by PLAPACK experts. All results were obtained on a 40 node Cray T3E.

To gather these results we annotated 29 of PLAPACK's 113 externally visible routines, yielding an annotation file that was 323 lines. Our Broadway-optimized results focused on customizing one PLAPACK routine, the `PLA_Trsm()` routine, which is common to both the Cholesky and Lyapunov applications. The hand-optimized Lyapunov program did not limit itself to this same scope. Details concerning the hand-optimized version of the Cholesky program can be found in the literature [3].

Our annotations mimicked the hand optimizations by defining an abstract interpretation for describing the distribution of PLAPACK objects, leading to optimizations like those described in Section 3.1. (Unlike the example in Figure 2, we did not define the `Contents` property.) The basic idea is that while most PLAPACK procedures are designed to accept any type of view, the actual parameters often have special distributions. When this information is propagated into the procedure, it yields a variety of specialization opportunities. Uncovering these opportunities requires the compiler to analyze multiple layers of nested procedure calls. It is the encapsulation of these layered routines that makes the unoptimized routines both general and inefficient.

5.1 Performance Evaluation

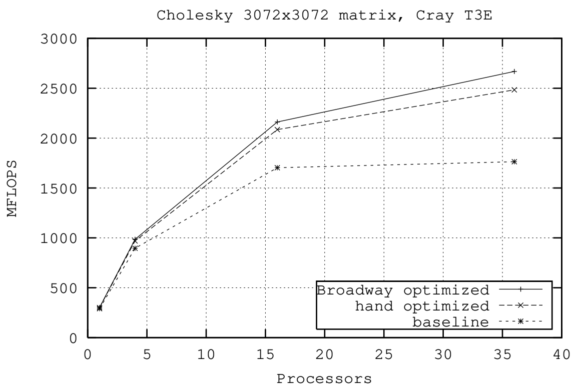


Figure 8: Scalability of the Cholesky programs as the number of processors grows.

Figure 6 shows the performance improvement of the Cholesky and Lyapunov programs. For fairly large matrices (6144×6144), the Broadway-optimized Cholesky

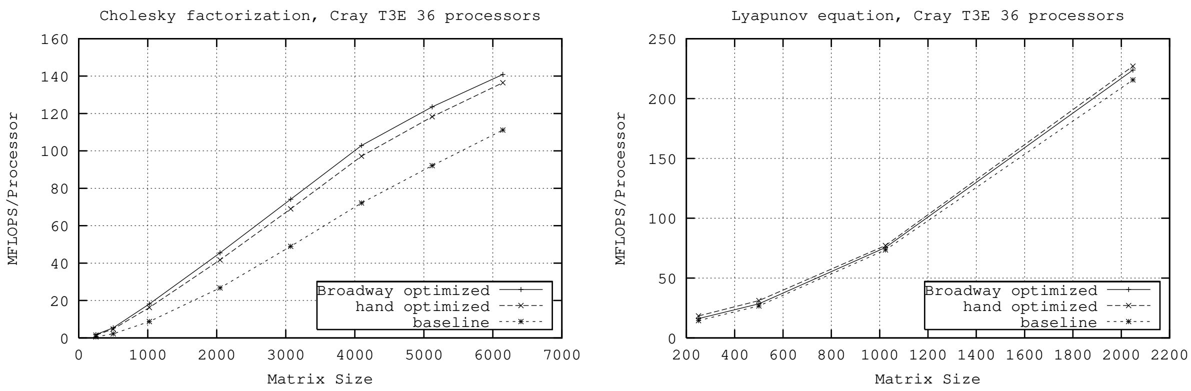


Figure 6: Performance comparison of hand-optimized and Broadway-optimized PLAPACK applications.

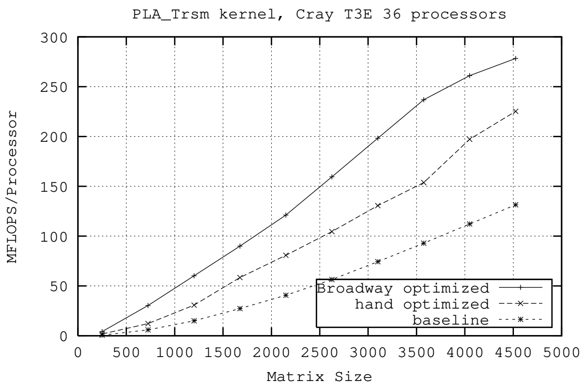


Figure 7: Performance comparison of hand-customized and Broadway-customized PLA_Trsm() function for the Cholesky program. For the Lyapunov program, the hand-customized PLA_Trsm() function matched the performance of the Broadway-customized version.

program is 26% faster than the baseline and the hand-optimized program is 22% faster than the baseline. For the Lyapunov program, the Broadway system does not perform as well as the manual approach, improving performance by 9.5% compared to the hand-optimized improvement of 21.5% for 250×250 matrices, and improving performance by 5.8% compared to 6.1% for 2000×2000 matrices. *The two approaches obtain identical performance on the `PLA_Trsm()` kernel, but the hand-optimized program performs a few additional optimizations to other parts of the code.*

Note that there is considerable room for further improving the Lyapunov program, since `PLA_Trsm()` only accounts for 11.6% of the execution time for 250×250 matrices, and only 5.8% of the time for 2000×2000 matrices. When our compiler is complete, we will apply our optimizations to all parts of the PLAPACK library, including the `PLA_Gemm()` routine, where Lyapunov spends a majority of its time.

Since our experiment focuses on the benefits of specializing the PLA_Trsm() routine, Figure 7 shows the performance difference between the generic PLA_Trsm() routine and the version that was customized for Cholesky by our compiler. Notice that we observe similar results for different numbers of processors. Figure 8 shows how the performance of the various Cholesky programs scale with the number of processors.

The results reveal several interesting points.

- A small effort yields a large benefit because the annotations only contain library knowledge, while all compilation expertise resides in the Broadway Compiler. The library annotator supplies the small but critical bits of information—such as specifying the conditions required to substitute a specific PLAPACK routine in place of a more general one—while the compiler analyzes the program, identifies opportunities for transformations, and manages a number of optimization passes. This separation of concerns is beneficial because the performance improvements shown in Figure 7 come from the repeated application of a small number of transformations.

- Automation is desirable. Both the Cholesky and Lyapunov programs specialize the same PLAPACK routine, but they do so in slightly different ways because they invoke it in different contexts.
- An automated approach can apply all optimizations uniformly. There is no fundamental reason why the hand-optimized Cholesky factorization is not as efficient as ours, but the manual approach, which is quite invasive, did not employ one transformation that it could have.
- The effect of customization is more important for small matrices. For example, for a 1024×1024 matrix, the Broadway-optimized Cholesky factorization is 2.95 times faster than the base, and the hand-optimized is 2.47 times faster than the base. When matrices are small the improvements are larger because there is more overhead relative to matrix operations. Because dense linear algebra problems do not typically involve huge matrices, the small matrix cases is important for scaling to larger numbers of processors, and for supporting sparse matrix operations.

Closer examination of the Cholesky results reveal that specialization and dead code elimination account for almost all of the performance benefits, while high level copy propagation (where the copy operations are library routines) contributes insignificantly.

5.2 Language Evaluation

Simplicity. Our annotation language is small and simple. There are 15 keywords and a small number of simple concepts. The basic annotations require a knowledge of C and the library's data structures. The advanced annotations require a deeper knowledge of the library's implementation. Anecdotal evidence suggests that the language is intuitive. When shown the advanced annotations for PLAPACK, the head of the PLAPACK project claimed that they seemed "very natural."

While our language is quite simple, we believe that we can simplify the *use* of the language. Eventually, we imagine that basic annotations will only be specified where static analysis fails. For example, a static analysis tool could guide the annotation by identifying routines that must be manually annotated.

Separation of Concerns. Our annotation language clearly separates the optimization information from the basic algorithm. By contrast manual optimization directly modifies the application source code, which complicates subsequent modification, reuse and maintenance. Moreover, we attempt to separate domain-specific information, which we place in the annotations, from compilation-specific information, which is embedded in the compiler. This separation of concerns simplifies both the library implementation and the specification of the annotations.

Generality. Our experiments show that our annotation language is effective when applied to PLAPACK. We believe that the language will also be effective for other libraries because the information conveyed by the basic annotations is fundamental to the analysis of any software, and the advanced annotations support abstract interpretation [9, 17], which is useful for modeling domain-specific information. In particular, such analysis is useful to any library that provides specialized routines that are tailored for specific contexts. For example, the Open GL graphics standard [21] can customize various matrix transformations to exploit particular properties of matrices and matrix operations. In operating systems, specialized file system I/O routines can be produced that are optimized for specific system states [10]: a specialized read routine can be created for the common situation in which the file is known to be open and the file position is correctly positioned to the next unread byte. As a final example, most layered systems can benefit from passing state information across layers [1], providing contextual information that can trigger the use of specialized routines.

Expressiveness. Because we have traded off generality for simplicity, our language is limited in the types of abstract interpretation that are supported. For example, our `property` annotations only allow enumerated lists of values, which correspond to finite lattices. In addition, our lattices have a fixed height of two. These restrictions ensure that our dataflow framework will converge, at the same time hiding the lattice-theoretic foundation of dataflow analysis from the annotator. We anticipate supporting more complex lattices, including integer ranges and restricted classes of infinite lattices. We will enforce termination by putting bounds on the number of iterations of our dataflow analysis.

Our language is also restricted in the sense that there is no way to create dependences between different abstract interpretations.

6 Conclusions and Future Work

We have introduced a system that allows libraries to be both general and efficient. Applications can use a library's most general interface, and our compiler can customize the library implementation for different application needs. The key to our solution is an annotation language that conveys domain-specific information to the Broadway Compiler. The cost is that of annotating libraries, but the benefits are many: (1) Our compiler can perform domain-specific optimizations that are not possible without annotations; (2) our approach supports the use of cleaner, simpler interfaces, which leads to application code that is easier to maintain; (3) our approach provides a clear separation of concerns, as optimization information is encapsulated in the annotations rather than embedded in the application source code. In effect, the annotation language allows our compiler to treat libraries as semantically-rich but syntactically poor languages.

We have tested our technique by applying it to two programs written using the PLAPACK library. Our experience shows that (1) pointer-based C code can be analyzed with the help of our annotations, (2) our technique can produce significant performance improvements, even for a library that has already been carefully designed to achieve good performance, (3) a small number of simple annotations can be effective, and (4) the same set of annotations can be used to optimize multiple applications.

This work can be extended in many directions. When our compiler implementation is complete we will apply our transformations uniformly to a wider body of PLAPACK routines and a larger number of PLAPACK applications. We also plan to annotate other libraries, such as the standard math library, the MPICH [15] implementation of the Message Passing Interface, and perhaps Open GL [21]. More fundamentally, we are developing compilation strategies that allow us to optimize across multiple layers of libraries, and we are also exploring ways to extend our annotation language to support machine-specific customization.

Acknowledgments. We thank Robert van de Geijn for many insightful discussions about PLAPACK. We thank E Christopher Lewis and Yannis Smaragdakis for valuable comments on preliminary versions of this paper.

References

[1] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[2] 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.

[3] 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.

[4] P. Benner and E.S. Quintana-Orti. Parallel distributed solvers for large stable generalized Lyapunov equations. In *Parallel Processing Letters*, 1998 (to appear).

[5] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990.

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

[7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.

[8] 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.

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

[10] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems*, May 1996.

[11] 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.

[12] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 103–118, Berkeley, October15–17 1997. USENIX Association.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. DyC: An expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, to appear.

[14] B. Grant, M. Philipose, M. Mock, C. Chambers, and S.J. Eggers. An evaluation of staged run-time optimizations in DyC. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–233, 1999.

[15] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[16] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.

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

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1999. Finland, Springer-Verlag LNCS 1241.

[19] 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.

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

[21] Jackie Neider, Tom Davies, and Mason Woo. *Open GL Programming Guide*. Addison-Wesley, 1996.

[22] J. N. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In T. J. Biggerstaff and C. Richter, editors, *Software Reusability*, volume I — Concepts and Models, chapter 12, pages 295–319. ACM press, 1989.

[23] Y. Smaragdakis and D. Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, to appear.

[24] Yannis Smaragdakis and Don Batory. DiSTiL: a transformation library for data structures. In *USENIX Conference on Domain-Specific Languages (DSL-97)*, October 1997.

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

[26] 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.

[27] Eugen N. Volanschi, Charles Consel, and Crispin Cowan. Declarative specialization of object-oriented programs. *SIGPLAN Notices, Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, 39(1):286–300, October 1997.

[28] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, La Jolla, California, 18–21 June 1995.

A Annotation language grammar

This appendix presents the complete grammar for the annotation language. We use the following type face conventions: *Italic font* for non-terminals, **`bold typewriter font`** for literal terminals including keywords, and SMALL CAPS for the lexicographic terminals such as identifiers and C code fragments. In addition, we use the square brackets to represent optional components, and the star to represent repetition of a component.

A.1 Overall format

Annotations	→	*Header*
		Annotation ⋆
Header	→	**`%{`**
		C-CODE
		`%}`
Annotation	→	*Property_ann*
	|	*Global_ann*
	|	*Procedure*

A.2 Globals and properties

Global_ann	→	**`global`** { *Identifiers* }
Property_ann	→	**`property`** { *Properties* }
Properties	→	*Property* [**`,`** *Properties*]
Property	→	IDENTIFIER [**`= none`**]

A.3 Procedures

Procedure	→	**`procedure`** IDENTIFIER **`(`** *identifiers* **`)`**
		{ *Proc_ann* ⋆ }
Proc_ann	→	*Structure_ann*
	|	*Def_use_ann*
	|	*Analyze_ann*
	|	*Specialize_ann*

A.4 Object structure

Structure_ann	→	**`on_entry`** { *Structures* }
	|	**`on_exit`** { *Structures* }
Structures	→	*Structure* [**`,`** *Structures*]
Structure	→	*Source* **`-->`** *Target*
	|	IDENTIFIER **`copyof`** IDENTIFIER
Source	→	[IDENTIFIER **`of`**] IDENTIFIER
Target	→	IDENTIFIER
	|	**`null`**

A.5 Definitions and uses

Def_use_ann	→	**`access`** { *Identifiers* }
	|	**`modify`** { *Identifiers* }
Identifiers	→	IDENTIFIER [**`,`** *Identifiers*]

A.6 Analyze

Analyze_ann	→	**`analyze`** IDENTIFIER { *Rule* ⋆ }
Rule	→	[*Condition* **`=>`**] *Consequence* **`;`**
Condition	→	IDENTIFIER [IDENTIFIER] **`==`** IDENTIFIER
	|	IDENTIFIER [IDENTIFIER] **`==`** CONSTANT
	|	**`(`** *Condition* **`)`**
	|	*Condition* **`&&`** *Condition*
	|	*Condition* **`||`** *Condition*
Results	→	*Result* [**`,`** *Results*]
Result	→	IDENTIFIER **`=`** IDENTIFIER

A.7 Specialize

Specialize	→	**`specialize`** { *Spec* ⋆ }
Spec	→	*Condition* **`=>`** *Replacement* **`;`**
Replacement	→	**`remove`**
	|	**`replace`** C-CODE