# Optimizing the Use of High Performance Software Libraries*

Samuel Z. Guyer and Calvin Lin

The University of Texas at Austin, Austin, TX 78712

**Abstract.** This paper describes how the use of software libraries, which is prevalent in high performance computing, can benefit from compiler optimizations in much the same way that conventional programming languages do. We explain how the compilation of these informal languages differs from the compilation of more conventional languages. In particular, such compilation requires precise pointer analysis, domain-specific information about the library's semantics, and a configurable compilation scheme. We describe a solution that combines dataflow analysis and pattern matching to perform configurable optimizations.

## 1 Introduction

High performance computing, and scientific computing in particular, relies heavily on software libraries. Libraries are attractive because they provide an easy mechanism for reusing code. Moreover, each library typically encapsulates a particular domain of expertise, such as graphics or linear algebra, and the use of such libraries allows programmers to think at a higher level of abstraction. In many ways, libraries are informal domain-specific languages whose only syntactic construct is the procedure call. This procedural interface is significant because it couches these informal languages in a familiar form without imposing new syntax. Unfortunately, libraries are not viewed as languages by compilers. With few exceptions, compilers treat each invocation of a library routine the same as any other procedure call. Thus, many optimization opportunities are lost because the semantics of these informal languages are ignored.

As a trivial example, an invocation of the C standard math library's exponentiation function, `pow(a,b)`, can be simplified to `1` when its second argument is `0`. This paper argues that there are many such opportunities for optimization, if only compilers could be made aware of a library's semantics. These optimizations, which we term *library-level* optimizations, include choosing specialized library routines in favor of more general ones, eliminating unnecessary library calls, moving library calls around, and customizing the implementation of a library routine for a particular call site.

Figure 1 shows our system architecture for performing library-level optimizations [13]. 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.

* 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.

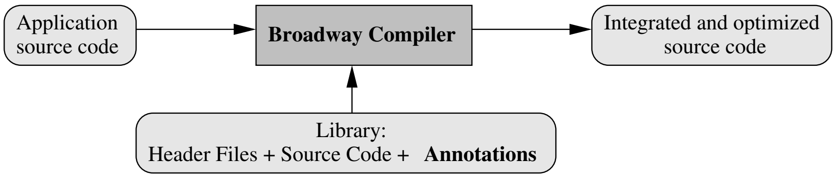


**Fig. 1.** Architecture of the Broadway Compiler system

This system architecture offers three practical benefits. First, because the annotations are separate from the library source, 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.

This architecture also provides an important conceptual benefit: a clean separation of concerns. The compiler encapsulates all compiler analysis and optimization machinery, while the annotations describe all library knowledge and domain expertise. Together, the annotations and compiler free the applications programmer to focus on application design rather than on performing manual library-level optimizations.

The annotation language faces two competing goals. To provide simplicity, the language needs to have a small set of useful constructs that apply to a wide range of software libraries. At the same time, to provide power, the language has to convey sufficient information for the Broadway compiler to perform a wide range of optimizations. The remainder of this paper will focus on the annotation language and its requirements.

This paper makes the following contributions. (1) We define the distinguishing characteristics of library-level optimizations. (2) We describe the implications of these characteristics for implementing library-level optimizations in a compiler. (3) We present formulations of dataflow analysis and pattern matching that address these implications. (4) We extend our earlier annotation language [13] to support the configuration of the dataflow analyzer and pattern matcher.

## 2 Opportunities

This section characterizes library-level optimizations and their requirements.

*Conceptually similar to traditional optimizations.* Library-level optimizations are conceptually similar to traditional optimizations, which can be grouped into the following classes of optimizations. (1) *Eliminate redundant computations:* Examples include partial redundancy elimination, common subexpression elimination, loop-invariant code motion, and value numbering. (2) *Perform computations at compile-time:* This may be as simple as constant folding or as complex as partial evaluation (3) *Exploit special cases:* Examples include algebraic identities and simplifications, as well as strength reduction. (4) *Schedule code:* For example, exploit non-blocking loads and asynchronous

I/O operations to hide the cost of long-latency operations. (5) *Enable other improvements:* Use transformations such as inlining, cloning, loop transformations, and lowering the internal representation. These same categories form the basis of our library-level optimization strategy. In some cases, the library-level optimizations are identical to their classical counterparts. In other cases we need to reformulate the optimizations for the unique requirements of libraries.

*Significant opportunities exist.* Most libraries receive no support from traditional optimizers. For example, the code fragments in Figure 2 illustrate the untapped opportunities of the standard C Math Library. A conventional C compiler will perform three optimizations on the built-in operators: (1) strength reduction on the computation of `d1`, (2) loop-invariant code motion on the computation of `d3`, and (3) replacement of division with bitwise right-shift in the computation of `int1`. The resulting optimized code is shown in the middle fragment. However, there are three analogous optimization opportunities on the math library computations that a conventional compiler will not discover: (4) strength reduction of the power operator in the computation of `d2`, (5) loop-invariant code motion on the cosine operator in the computation of `d4`, and (6) replacement of sine divided by cosine with tangent in the computation of `d5`. The code fragment on the right shows the result of applying these optimizations.

Original Code

```
for (i=1; i<=N; i++) {
  d1 = 2.0 * i;              1
  d2 = pow(x, i);
  d3 = 1.0/z;                2
  d4 = cos(z);
  int1 = i/4;                3
  d5 = sin(y)/cos(y);
}
```

Conventional

```
d1 = 0.0;
d3 = 1.0/z;
for (i=1; i<=N; i++) {
  d1 += 2.0;
  d2 = pow(x, i);            4
  d4 = cos(z);               5
  int1 = i >> 2;
  d5 = sin(y)/cos(y);        6
}
```

Library−level

```
d1 = 0.0;
d2 = 1.0;
d3 = 1.0/z;
d4 = cos(z);
d5 = tan(y);
for (i=1; i<=N; i++) {
  d1 += 2.0;
  d2 *= x;
  int1 = i >> 2;
}
```

**Fig. 2.** A conventional compiler optimizes built-in math operators, but not math library operators.

Significantly, each application of a library-level optimization is likely to yield much greater performance improvement than the analogous conventional optimization. For example, removing an unnecessary multiplication may save a few cycles, while removing an unnecessary cosine computation may save hundreds or thousands of cycles.

*Specialized routines are difficult to use.* Many libraries provide a *basic* interface which provides basic functionality, along with an *advanced* interface that provides specialized routines that are more efficient in certain circumstances [14]. For example, the MPI message passing interface [11] provides 12 variations of the basic point-to-point communication operation. These advanced routines are typically more difficult to use than the basic versions. For example, MPI's Ready Send routines assume that the communicating processes have already been somehow synchronized. These specialized routines represent an opportunity for library-level optimization, as a compiler would ideally translate invocations of basic routines to specialized routines.

*Domain-specific analysis is required.* Most libraries provide abstractions that can be useful for performing optimizations. For example, the PLAPACK parallel linear algebra library [19] manipulates linear algebra objects indirectly though handles called *views*. A view consists of data, possibly distributed across processors, and an index range that selects some of the data. 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]. For example, in some cases, calls to the general-purpose `PLA_Trsm()` routine can be replaced with calls to a specialized routine, `PLA_Trsm_Local()`, which assumes the matrix *view* resides completely on a single processor. This customized routine can run as much as three times faster [13].

The key to this optimization is to analyze the program to discover the special case matrix distributions. While compilers can perform many kinds of dataflow analysis, most compilers have no notion of "matrix," let alone PLAPACK's particular notion of matrix distributions. Thus, to perform this kind of optimization, there must be a mechanism for telling the compiler about the relevant abstractions and for facilitating program analysis in those terms.

*Challenges.* To summarize, while there are many opportunities for library-level optimization, there are also significant challenges. First, while library-level optimizations are conceptually similar to traditional optimizations, library routines are typically more complex than primitive language operators. Second, a library typically embodies a high-level domain of computation whose abstractions are not represented in the base language, and effective optimizations are often phrased in terms of these abstractions. Third, the compiler cannot be hardwired for every possible library because each library requires unique analyses and optimizations. Instead, all of these facilities need to be configurable. The next two sections address these issues in more detail.

## 3 Dependence Analysis

Almost any kind of program optimization requires a model of dataflow dependences to preserve the program's semantics. The use of pointers, and particularly pointer-based data structures, can greatly complicate dependence analysis, as pointers can make it difficult to determine which memory objects are actually modified. While many solutions to the pointer analysis problem have been proposed, we now argue that optimization at the library level requires the most precise and most aggressive.

### 3.1 Why Libraries Need Pointers

Libraries use pointers for two main reasons. The first is to overcome the limitations of the procedure call mechanism, as pointers allow a procedure to return more than one value. The second reason is to build and manipulate complex data structures that represent the library's domain-specific programming abstractions. It is important to understand these structures because data dependences may exist between internal components of these data structures which, if violated, could change the program's behavior. As an example, consider the following PLAPACK routine:

```
PLA_Obj_horz_split_2(size, A, &A_upper, &A_lower);
```

This routine logically splits the matrix into two pieces by returning objects that represent ranges of the original matrix index space. Internally, the library defines a *view* data structure that consists of minimum and maximum values for the row and column indices, and a pointer to the actual matrix data. To see how this complicates analysis, consider the following code fragment:

```
PLA_Obj A, A_upper, A_lower, B;
PLA_Create_matrix(num_rows, num_cols, &A);
PLA_Obj_horz_split_2(size, A, &A_upper, &A_lower);
B = A_lower;
```

The first line declares four variables of type `PLA_Obj`, which is an opaque pointer to a *view* data structure. The second line creates a new matrix (both *view* and data) of the given size. The third line creates two *views* into the original data by splitting the rows into two groups, upper and lower. The fourth line performs a simple assignment of one *view* variable to another. Figure 3 shows the resulting data structures graphically.

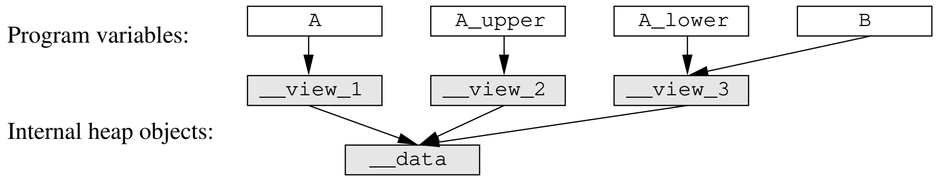


**Fig. 3.** Library data structures have complex internal structure.

The shaded objects are never visible to the application code, but accesses and modifications to them are still critical to preserving program semantics. For example, regardless of whether `A`, `A_lower`, or `B` is used, the compiler cannot change the order of library calls that update the data.

### 3.2 Pointer Analysis for Library-Level Optimizations

Pointer analysis attempts to determine whether there are multiple ways to access the same piece of memory. We categorize the many approaches to pointer analysis along the following dimensions: (1) points-to versus alias representation, (2) heap model, and (3) flow and context sensitivity. This section describes the characteristics of our compiler's pointer analysis and explains why they are appropriate for library-level optimization.

*Representation.* Heap allocation is common for library code because it allows data structures to exist throughout the life of the application code and it supports complex and dynamic structures. By necessity, components of these structures are connected by pointers, so a points-to relation provides a more natural model than alias pairs. Furthermore, in C the only true variable aliasing occurs between the fields of a union; all other aliases occur through pointers and pointer expressions.

*Heap Model.* The heap model determines the granularity and naming of heap allocated memory. Previous work demonstrates a range of possibilities including (1) one object to represent the whole heap [9], (2) one object for each connected data structure [12], (3) one object for each allocation call site [4], and (4) multiple objects for a malloc call site with a fixed limit ("k-limiting") [6]. In the matrix split example above, we need a model of the heap that distinguishes the *views* from the data. Thus, neither of the first two approaches is sufficiently precise. We choose approach (3) because it is precise, without the unnecessary complexity of the k-limiting approach.

Often, a library will provide only one or two functions that allocate new data structures. For example, the above `PLA_matrix_create` function creates all matrices in the system. Thus, if we associate only one object (or one data structure, e.g., one *view* and one data object) with the call site, we cannot distinguish between individual instances of the data structure. In the example, we would have one object to represent all *views* created by the split operation, preventing us from distinguishing between `__view_2` and `__view_3`. Therefore, we create a new object in the heap model for each unique execution path that leads to the allocation statement.

This naming system leads to an intuitive heap model where objects in the model often represent conceptual categories, such as "the memory allocated by foo()." Note that when allocation occurs in a loop, all of the objects created during execution of the loop are represented by one object in the heap model.

*Context and Flow Sensitivity.* Libraries are mechanisms for software reuse, so library calls often occur deep in the application's call graph, with the same library functions repeatedly invoked from different locations. Without context and flow sensitivity, the analyzer merges information from the different call sites. For example, consider a simple PLAPACK program that makes two calls to the split routine with different matrices:

```
PLA_Obj_horz_split_2(size, A, &A_upper, &A_lower);
PLA_Obj_horz_split_2(size, B, &B_upper, &B_lower);
```

Context insensitive analysis concludes that all four outputs might point to either `A`'s data or `B`'s data. While this information is conservatively correct, it severely limits optimization opportunities by creating an unnecessary data dependence. Any subsequent analyses that use this information suffers the same merging of information. For example, if the state of `A` is unknown, then analysis cannot safely infer that `B_upper` and `B_lower` contain all zeros, even if analysis concludes that `B` contains all zeros before the split.

The more reuse that occurs in a system, the more important it is to keep information separate. Thus, we implement full context sensitivity. While this approach is complex, recent research shows that efficient implementations are possible [22]. Recent work also shows that more precise pointer analysis not only causes subsequent analyses to produce more precise results, but also causes them to run faster [18].

### 3.3 Annotations for Dependence Analysis

Our annotation language provides a mechanism for explicitly describing how a routine affects pointer structures and dataflow dependences. This information is integrated into the Broadway compiler's dataflow and pointer analysis framework. The compiler builds a uniform representation of dependences, regardless of whether they involve built-in operators or library routines. When annotations are available, our compiler reads that information directly. For the application and for libraries that have not been annotated, our compiler analyzes the source code using the pointer analysis described above.

```
procedure PLA_Obj_horz_split_2( obj, height, upper, lower)
{
  on_entry { obj    --> __view_1, DATA  of __view_1 -->__data }
  access { __view_1, height }
  modify { }
  on_exit  { upper  --> new __view_2, DATA  of  __view_2 --> __data,
             lower  --> new __view_3, DATA  of  __view_3 --> __data }
}
```

**Fig. 4.** Annotations for pointer structure and dataflow dependences.

Figure 4 shows the annotations for the PLAPACK matrix split routine. In some cases, a compiler could derive such information from the library source code. However, there are situations when this is impossible or undesirable. Many libraries encapsulate functionality for which no source code is available, such as low-level I/O or interprocess communication. Moreover, the annotations allow us to model abstract relationships that are not explicitly represented through pointers. For example, a file descriptor logically refers to a file on disk through a series of operating system data structures. Our annotations can explicitly represent the file and its relationship to the descriptor, which might make it possible to recognize when two descriptors access the same file.

*Pointer Annotations:* `on_entry` *and* `on_exit`. To convey the effects of the procedure on pointer-based data structures, the `on_entry` and `on_exit` annotations describe the pointer configuration before and after execution. Each annotation contains a list of expressions of the following form:

[ *label* `of` ] *identifier* `-->` [ `new` ] *identifier*

The `-->` operator, with an optional label, indicates that the object named by the identifier on the left logically points to the object named on the right. In the `on_entry` annotations, these expressions describe the state of the incoming arguments and give names to the internal objects. In the `on_exit` annotations, the expressions can create new objects (using the `new` keyword) and alter the relationships among existing objects.

In Figure 4, the `on_entry` annotation indicates that the formal parameter `obj` is a pointer and assigns the name `__view_1` to the target of the pointer. The annotation also says that `__view_1` is a pointer that points to `__data`. The `on_exit` annotation declares that the split procedure creates two new objects, `__view_2` and `__view_3`. The resulting pointer relationships correspond to those in Figure 3.

*Dataflow Annotations:* `access` *and* `modify`. The `access` and `modify` annotations declare the objects that the procedure accesses or modifies. These annotations may refer to formal parameters or to any of the internal objects introduced by the pointer annotations. The annotations in Figure 4 show that the procedure uses the `length` argument and reads the input *view* `__view_1`. In addition, we automatically add the accesses and modifications implied by the pointer annotations: a dereference of a pointer is an access, and setting a new target is a modification.

### 3.4 Implications

As described in Section 2, most library-level optimizations require more than just dependence information. However, simply having the proper dependence analysis information for library routines does enable some classical optimizations. For example, by separating accesses from modifications, we can identify and remove library calls that are dead code. To perform loop invariant code motion or common subexpression elimination, the compiler can identify purely functional routines by checking whether the objects accessed are different from those that are modified.

## 4 Library-Level Optimizations

Our system performs library-level optimizations by combining two complementary tools: dataflow analysis and pattern-based transformations. Patterns can concisely specify local syntactic properties and their transformations, but they cannot easily express properties beyond a basic block. Dataflow analysis concisely describes global context, but cannot easily describe complex patterns. Both tools have a wide range of realizations, from simple to complex, and by combining them, each tool is simplified. Both tools are configurable, allowing each library to have its own analyses and transformations.

The patterns need not capture complex context-dependent information because such information is better expressed by the program analysis framework. Consider the following fragment from an application program:

```
PLA_Obj_horz_split_2( A, size, &A_upper, &A_lower);
...
if ( is_Local(A_upper) ) { ... } else { ... }
```

The use of `PLA_Obj_horz_split_2` ensures that `A_upper` resides locally on a single processor. Therefore, the condition is always true, and we can simplify the subsequent `if` statement by replacing it with the then-branch. The transformation depends on two conditions that are not captured in the pattern. First, we need to know that the `is_Local` function does not have any side-effects. Second, we need to track the library-specific *local* property of `A_upper` through any intervening statements to make sure that it is not invalidated. It would be awkward to use patterns to express these conditions.

Our program analysis framework fills in the missing capabilities, giving the patterns access to globally derived information such as data dependences and control-flow information. We use *abstract interpretation* to further extend these capabilities, allowing each library to specify its own dataflow analysis problems, such as the matrix distribution analysis needed above. This approach also keeps the annotation language itself simple, making it easier to express optimizations and easier to get them right. The following sequence outlines our process for library-level optimization:

1. **Pointers and dependences.** Analyze the code using combined pointer and dependence analysis, referring to annotations when available to describe library behavior.
2. **Classical optimizations.** Apply traditional optimizations that rely on dependence information only.
3. **Abstract interpretation.** Use the dataflow analysis framework to derive library-specific properties as specified by the annotations.
4. **Patterns**. Search the program for possible optimization opportunities, which are specified in the annotations as syntactic patterns.
5. **Enabling conditions.** For patterns that match, the annotations can specify additional constraints, which are expressed in terms of data dependences and the results of the abstract interpretation.
6. **Actions.** When a pattern satisfies all the constraints, the specified code transformation is performed.

### 4.1 Configurable Dataflow Analysis

This section describes our configurable interprocedural dataflow analysis framework. The annotations can be used to specify a new analysis pass by defining both the library-specific flow value and the associated transfer functions. For every new analysis pass, each library routine is supplied a transfer function that represents its behavior.

The compiler reads the specification and runs the given problem on our general-purpose dataflow framework. The framework takes care of propagating the flow values through the flow graph, applying the transfer functions, handling control-flow such as conditionals and loops, and testing for convergence. Once complete, it stores the final, stable flow values for each program point.

While other configurable program analyzers exist, ours is tailored specifically for library-level optimization. First, we would like library experts, not compiler experts, to be able to define their own analyses. Therefore, the specification of new analysis problems is designed to be simple and intuitive. Second, we do not intend to support every possible analysis problem. The annotation language provides a small set of flow value types and operators, which can be combined to solve many useful problems. The lattices implied by these types have predefined meet functions, allowing us to hide the underlying lattice theory from the annotator.

For library-level optimization, the most useful analyses seem to fall into three rough categories: (1) analyze the objects used by the program and classify them into library-specific categories, (2) track relationships among those objects, and (3) represent the overall state of the computation. The PLAPACK distribution analysis is an instance of the first kind. To support these different kinds of analysis, we propose a simple type system for defining flow values.

*Flow Value Types.* Our flow value type system consists of primitive types and simple container types. A flow value is formed by combining a container type with one or two of the primitive types. Each flow value type includes a predefined meet function, which defines how instances of the type are combined at control-flow merge points. The other operations are used to define the transfer functions for each library routine.

**number** The *number* type supports basic arithmetic computations and comparisons. The meet function is a simple comparison: if the two numbers are not equal, then the result is lattice bottom.

**object** The *object* type represents any memory location, including global and stack variables, and heap allocated objects. The only operation supported tests whether two expressions refer to the same object.

**statement** The *statement* type refers to points in the program. We can use this type to record where computations take place. Operations include tests for equality, dominance and dependences.

**category** *Categories* are user-defined enumerated types that support hierarchies. For example, we can define a `Vehicle` categorization like this:

```
{ Land { Car, Truck { Pickup, Semi}}, Water { Boat, Submarine}}
```

The meet function chooses the most specific category that includes the two given values. For example, the meet of `Pickup` and `Semi` yields `Truck`, while the meet of `Pickup` and `Submarine` yields lattice bottom. The resulting lattice forms a simple tree structure, as shown in Figure 5.

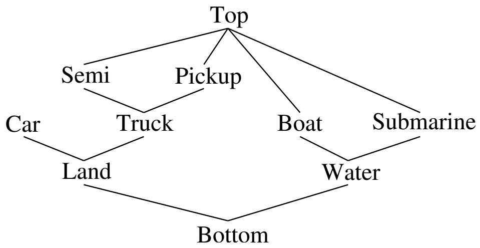


**Fig. 5.** The lattice induced by the example vehicle categories.

Operations on the categories include testing for equality and for category membership. For example, we may want to know whether something is a `Truck` without caring about its specific subtype.

**set-of<T>** Set is the simplest container: it holds any number of instances of the primitive type T. We can add and remove elements from the set, and test for membership.

We support two possible meet functions for sets: set union for "optimistic" analyses, and set intersection for "pessimistic" analyses. It is up to the annotator to decide which is more appropriate for the specific analysis problem.

**equivalence-of**<**T**> This container maintains an equivalence relation over the elements that it contains. Operations on this container include adding pairs of elements to indicate that they are equivalent, and removing individual elements. The basic query operator tests whether two elements are equivalent by applying the transitive closure over the pairs that have been added. Like the set container, there are two possible meet functions, one optimistic and one pessimistic, that correspond to the union and intersection of the two relations.

**ordering-of**<**T**> The ordering container maintains a partial order over the elements it contains. We add elements in pairs, smaller element and larger element, to indicate ordering constraints. We can also remove elements. Like the equivalence container, the ordering container allows ordering queries on the elements it contains. In addition, it ensures that the relation remains antisymmetric by removing cycles completely.

**map-of**<**K,V**> The map container maintains a mapping between elements of the two given types. The first type is the "key" and may only have one instance of a particular value in the map at a time. It is associated with a "value."

We can model many interesting analysis problems using these simple types. The annotations define an analysis using the `property` keyword, followed by a name and then a flow value type expression. Figure 6 shows some example property definitions. The first one describes the flow value for the PLAPACK matrix distribution analysis. The equivalence `Aligned` could be used to determine when matrices are suitably aligned on the processors. The partial order `SubmatrixOf` could maintain the relative sizes of matrix views. The last example could be used for MPI to keep track of the asynchronous messages that are potentially "in flight" at any given point in the program.

```
property Distribution :
           map-of  < object , {  General { RowPanel, ColPanel, Local },
                                 Vector,
                                 Empty } >
property Aligned : pessimistic  equivalence-of < object >
property SubMatrixOf :  ordering-of < object >
property MessagesInFlight :  optimistic  set-of < object >
```

**Fig. 6.** Examples of the `property` annotation for defining flow values.

*Transfer Functions.* For each analysis problem, transfer functions summarize the effects of library routines on the flow values. Transfer functions are specified as a case analysis, where each case consists of a condition, which tests the incoming flow values, and a consequence, which sets the outgoing flow value. Both the conditions and the consequences are written in terms of the functions available on the flow value type.

```
procedure PLA_Obj_horz_split_2( obj, height, upper, lower)
{
  on_entry { obj    --> __view_1, DATA  of __view_1 -->__data }
  access { __view_1, height }
  modify { }

  analyze Distribution {
   (__view_1 == General)  => __view_2 = RowPanel, __view_3 = General;
   (__view_1 == RowPanel) => __view_2 = RowPanel, __view_3 = RowPanel;
   (__view_1 == ColPanel) => __view_2 = Local, __view_3 = ColPanel;
   (__view_1 == Local)    => __view_2 = Local, __view_3 = Empty;
  }

  on_exit  { upper  --> new __view_2, DATA  of  __view_2 --> __data,
             lower  --> new __view_3, DATA  of  __view_3 --> __data }
}
```

**Fig. 7.** Annotations for matrix distribution analysis.

Figure 7 shows the annotations for the PLAPACK routine `PLA_Obj_horz_split_2`, including those that define the matrix distribution transfer function. The `analyze` keyword indicates the property to which the transfer function applies. We integrate the transfer function with the dependence annotations because we need to refer to the underlying structures. Distribution is a property of the *views* (see Section 2), not the surface variables. Notice the last case: if we deduce that a particular *view* is `Empty`, we can remove any code that computes on that *view*.

### 4.2 Pattern-Based Transformations

The library-level optimizations themselves are best expressed using pattern-based transformations. Once the dataflow analyzer has collected whole-program information, many optimizations consist of identifying and modifying localized code fragments. Patterns provide an intuitive and configurable way to describe these code fragments. In PLAPACK, for example, we use the results of the matrix distribution analysis to replace individual library calls with specialized versions where possible.

Pattern-based transformations need to identify sequences of library calls, to check the call site against the dataflow analysis results, and to make modifications to the code. Thus, the annotations for pattern-based transformations consist of three parts: a *pattern*, which describes the target code fragment, *preconditions*, which must be satisfied, and an *action*, which specifies modifications to the code. The pattern is simply a code fragment that acts as a template, with special meta-variables that behave as "wildcards." The preconditions perform additional tests on the matching application code, such as checking data dependences and control-flow context, and looking up dataflow analysis results. The actions can specify several different kinds of code transformations, including moving, removing, or substituting the matching code.

*Patterns.* The pattern consists of a C code fragment with meta-variables that bind to different components in the matching application code. Our design is influenced by the issues raised in Section 3. Typical code pattern matchers work with expressions and rely on the tree structure of expressions to identify computations. However, the use of pointers and pointer-based data structures in the library interface presents a number of complications, and forces us to take a different approach.

The parameter passing conventions used by libraries have several consequences for pattern matching. First, the absence of a functional interface means that a pattern cannot be represented as an expression tree; instead, patterns consist of a sequence of statements with data dependences among them. Second, the use of the address operator to emulate pass-by-reference semantics obscures those data dependences. Finally, the pattern instance in the application code may contain intervening, but computationally irrelevant statements. Figure 8 depicts some of the possible complications by showing what would happen if the standard math library did not have a functional interface.

Basic Pattern

```
y = sin(x)/cos(x);

    →

y = tan(x);
```

Non−functional Interface

```
sin(x, &t0);
cos(x, &t1);
y = t0/t1;

    →

tan(x, &y);
```

Access Complications

```
p = &x;
sin(x, &t0);
cos(*p, &t1);
y = t0/t1;

    →

tan(*p, &y);
```

**Fig. 8.** The use of pointers for parameter passing complicates pattern matching.

To address these problems, we offer two meta-variable types, one that matches objects (both variables and heap-allocated memory), and one that matches constants. The object meta-variable ignores the different ways that objects are accessed. For example, in the third code fragment in Figure 8, the same meta variable would match both `x` and `*p`. The constant meta-variable can match a literal constant in the code, a constant expression, or the value of a variable if its value can be determined at compile time.

For a pattern to match, the application code must contain the specified sequence of statements, respecting any data dependences implied by the meta-variable names. The matching sequence may contain intervening statements, as long as those statements have no dependences with that sequence. We would like to weaken this restriction in the future, but doing so raises some difficult issues for pattern substitution.

*Preconditions.* The preconditions provide a way to test the results of the pointer analysis and user-defined dataflow analyses, since these can't be conveniently represented in the syntactic patterns. These dataflow requirements can be complicated for libraries, because important properties and dependences often exist between internal components of the data structures, rather than between the surface variables. For example, as shown in Figure 3, two different PLAPACK views may refer to the same underlying matrix data. An optimization may require that a sequence of PLAPACK calls all update the same matrix. In this case the annotations need a way to access the pointer analysis information and make sure that the condition is satisfied. To do this, we allow the preconditions to refer to the `on_entry` and `on_exit` annotations for the library routines in the pattern. To access the dataflow analysis results, the preconditions can express queries using the same flow value operators that the transfer functions use. For example, the preconditions can express constraints such as, "the view of matrix A is empty."

*Actions.* When a pattern matches and the preconditions are satisfied, the compiler can perform the specified optimization. We have found that the most common optimizations for libraries consist of replacing a library call or sequence of library calls with more specialized code. The replacement code is specified as a code template, possibly containing meta variables, much like the patterns. Here, the compiler expands the embedded meta-variables, replacing them with the actual code bound to them. We also support queries on the meta-variables, such as the C datatype of the binding. This allows us to declare new variables that have the same type as existing variables.

In addition to pattern replacement, we offer four other actions: (1) remove the matching code, (2) move the code elsewhere in the application, (3) insert new code, or (4) trigger one of the enabling transformations such as inlining or loop unrolling.

When moving or inserting new code, the annotations support a variety of useful *positional indicators* that describe where to make the changes relative to the site of the matching code. For example, the earliest possible point and the latest possible point are defined by the dependences between the matching code and its surrounding context. Using these indicators, we can perform the MPI scheduling described in Section 2: move the `MPI_Isend` to the earliest point and the `MPI_Wait` to the latest point. Other positional indicators might include enclosing loop headers or footers, and the locations of reaching definitions or uses. Figure 9 demonstrates some of the annotations that use pattern-based transformations to optimize the examples presented in this paper.

## 5 Related Work

Our research extends to libraries previous work in optimization [17], partial evaluation [3, 7], abstract interpretation [8, 15], and pattern matching. 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 [21]. 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 [16] 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 [5], programmable syntax macros [20], and the Magik compiler [10], can be used to create customized library

```
pattern {  ${obj:y} = sin(${obj:x})/cos(${obj:x})   }
{
  replace { $y = tan($x)  }
}

pattern {
    MPI_Isend( ${obj:buffer}, ${expr:dest}, ${obj:req_ptr})
  }
{
  move @earliest;
}

pattern {
    PLA_Obj_horz_split_2( ${obj:A}, ${expr:size},
                          ${obj:upper_ptr}, ${obj:lower_ptr})
  }
{
  on_entry { A --> __view_1, DATA of __view_1 --> __data }
  when (Distribution of __view_1 == Empty) remove;
  when (Distribution of __view_1 == Local)
    replace {
      PLA_obj_view_all($A, $upper_ptr);
    }
}
```

**Fig. 9.** Example annotations for pattern-based transformations.

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.

## 6 Conclusions

This paper has outlined the various challenges and possibilities for performing library-level optimizations. In particular, we have argued that such optimizations require precise pointer analysis, domain-specific information, and a configurable compilation scheme. We have also presented an annotation language that supports such a compilation scheme.

A large portion of our Broadway compiler has been implemented, including a flow- and context-sensitive pointer analysis, a configurable abstract interpretation pass, and the basic annotation language [13] without pattern matching. Experiments with this basic configuration have shown that significant performance improvements are possible for applications that use the PLAPACK library. One common routine, `PLA_Trsm()`, was customized to improve its performance by a factor of three, yielding speedups of 26% for a Cholesky factorization application and 9.5% for a Lyapunov program [13].

While we believe there is much promise for library-level optimizations, several open issues remain. We are in the process of defining the details of our annotation language extensions for pattern matching, and we are implementing its associated pattern matcher. Finally, we need to evaluate the limits of our scheme—and of our use of abstract interpretation and pattern matching in particular—with respect to both optimization capabilities and ease of use.

## References


1. P.V. Artigas, M. Gupta, S.P. Midkiff, and J.E. Moreira. High performance numerical computing in Java: language and compiler issues. In *Workshop on Languages and Compilers for Parallel Computing*, 1999.
2. G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: high performance through high level abstractions. In *Int'l Conf. on Parallel Processing*, 1998.
3. A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer*, 23(12):23–37, December 1990.
4. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.
5. 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.
6. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.
7. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.
8. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
9. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
10. D. 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.
11. Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.
12. R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.
13. 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.
14. S. Z. Guyer and C. Lin. Broadway: A software architecture for scientific computing. In *IFIPS Working Group 2.5: Software Architectures for Scientific Computing Applications*, October 2000.
15. N. D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford Univ. Press, 1994. 527–629.
16. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
17. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, 1997.
18. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *4th International Static Analysis Symposium, Lecture Notes in Computer Science, Vol. 1302*, 1997.
19. R. van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.
20. D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.
21. D. Whitfield and M. L. Soffa. Automatic generation of global optimizers. *ACM SIGPLAN Notices*, 26(6):120–129, June 1991.
22. R. P. Wilson. *Efficient, Context-sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, Department of Electrical Engineering, 1997.