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 computer languages do. We explain how the compilation of these informal languages differs from the compilation of more conventional computer languages. In particular, such compilation requires precise pointer analysis, requires domain-specific information about the library's semantics, and requires a configurable compilation scheme. Finally, we show that the combination of dataflow analysis and pattern matching form a powerful tool for performing 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 simply 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, which is less imposing than a new computer language that introduces new syntax. Unfortunately, while libraries are not viewed as languages by users, they also are not viewed as languages by compilers. With a 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 [14]. In this approach, annotations capture semantic information about library routines. These annotations are provided by a library expert and placed in a separate file that accompanies the usual header files and source code. This information is read by our compiler, dubbed the Broadway compiler, which performs source-to-source optimizations that transform both the library and application code. The resulting integrated system of library and application code is then compiled and linked using conventional tools.

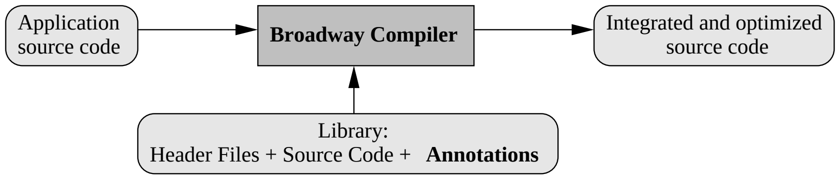

Application source code
Broadway Compiler
Integrated and optimized source code
Library:
Header Files + Source Code + Annotations

Fig. 1. Architecture of the Broadway Compiler system

This system architecture offers three practical benefits. First, because the annotations are specified in a separate file 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 system architecture also provides an important conceptual benefit in terms of separation of concerns. The compiler encapsulates all compiler analysis and optimization machinery, but no library-specific knowledge. Meanwhile, the annotations describe library knowledge and domain expertise. Together, the annotations and compiler free the applications programmer to focus on application design rather than on embedding library-level optimizations in the source code.

The annotation language faces two competing goals. To keep the annotations simple, 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 describe how we have designed an annotation language that addresses these goals.

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 automating 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 [14] to support the configuration of the dataflow analyzer and pattern matcher.

2 Opportunities

Traditional compilers employ a number of techniques to improve program performance. While not exhaustive, the following list covers most common 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 for 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. We now highlight the distinguishing characteristics of library-level optimizations.

Conceptually similar to traditional optimizations. We can use the same five categories to classify library-level optimizations. (1) Redundant operations can be eliminated. These operations can be entire library calls, or they can be hidden inside of library routines, as is often the case with copying of data, synchronization, etc. (2) Computations can be performed at compile time. In some cases these computations require domain-specific information that involve more than primitive arithmetic on scalars. (3) Special cases can be exploited by either cloning and specializing library routines, or by replacing an invocation of a general routine with an invocation of a more specific one. (4) Scheduling opportunities often exist. For example, in a communications library, when blocking send is replaced with a non-blocking send and corresponding wait, subsequent rescheduling is needed to allow overlap of communication and computation. (5) Enabling transformations, such as inlining, can also enable further library-level optimizations.

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.

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 handful of cycles, while removing an unnecessary cosine computation may save hundreds or thousands of cycles. Other mathematical libraries such as the GNU Multiple Precision Library (GMP) [13], exhibit similar opportunities.

Specialized routines are difficult to use. Many libraries provide a *basic* interface which provides basic functionality, along with an *advanced* interface that provides specialized

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 will optimize built-in math operators, but not math library operators.

routines that are more efficient in certain circumstances [15]. For example, the MPI message passing interface [11] provides 12 ways to send point-to-point messages. As another example, GMP's `mpn` interface is efficient but difficult to use:

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

Thus, these specialized routines represent an opportunity for library-level optimization, as a compiler would ideally translate invocations of basic routines to these 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 [20] 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 or all of the data. A typical algorithm operates by partitioning the views and working on one small piece at a time. While most PLAPACK procedures are designed to accept any type of view, the actual parameters often have special distributions. Recognizing and exploiting these special distributions can yield significant performance gains [2]. For example, in some applications, calls to the general-purpose `PLA_Trsm()` routine can be replaced with calls to a specialized routine, `PLA_Trsm_Local()`, that only works where the matrix *view* resides completely on a single processor. This customized routine can run as much as three times faster [14].

The key to this optimization lies in analyzing the program to discover the special case matrix distributions. Conventional compilers already perform many kinds of dataflow analysis to determine how the program manipulates data. Unfortunately, most compilers have no notion of "matrix," let alone PLAPACK's particular notion of matrix distributions. Thus, in order 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.

2.1 Challenges

To summarize, automatic library-level optimization presents several challenges that distinguish it from traditional optimization. First, library routines often perform more complex tasks and have more complex usage than built-in operators. In particular, the use of pointers and pointer-based data structures requires sophisticated analysis to determine dataflow dependences precisely. Second, a library typically embodies a high-level domain of computation whose abstractions are not represented in the base language (e.g., matrices). Effective library-level optimizations often depend on predicates written in terms of these abstractions. Third, each library has its own set of code transformations, beyond the traditional optimizations, that lend themselves to performance improvement. Since the compiler cannot be hard-wired for any specific library, all of these facilities need to be configurable. The next two sections address these issues in more detail.

3 Dependence Analysis

Program optimization of almost any kind requires a model of dataflow dependences in the program. The dependences help identify optimization opportunities while preserving the semantics of the program. The use of pointers, and particularly pointer-based data structures, can greatly complicate dependence analysis. A computation may store its result indirectly through a pointer, making 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 raise the level of programming abstraction, which we now discuss further.

A naive solution to the pointer problem is to develop a simple alias analyzer that determines the mapping from formal parameters to actual parameters when pass-by-reference conventions are used (typically, using the “&” operator in C). Unfortunately, this solution is not sufficient because libraries often build and manipulate complex data structures that represent the domain-specific programming abstractions. 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 matrix split routine:

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

This routine does not literally split the input matrix into two pieces. Rather, it 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.

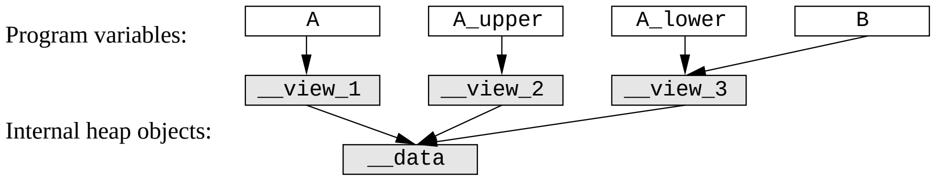


Fig. 3. Library data structures have complex internal structure.

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, the compiler cannot change the order of library calls that update the data, regardless of whether they use `A`, `A_lower`, or `B`.

3.2 Pointer Analysis for Library-Level Optimizations

At the most abstract level, all pointer analyses attempt to determine whether there are multiple ways to access the same piece of memory. However, there are many different approaches to pointer analysis that yield different kinds of information and are suitable for different optimization problems. We can categorize these according to a taxonomy based on the following design dimensions: (1) points-to versus alias representation, (2) “may” versus “must” information, (3) heap model, (4) 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. Library code typically allocates data structures on the heap because it allows them to exist throughout the application code, and it supports complex and dynamic structures. By necessity, components of these structures are connected by pointers. Thus, 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.

May versus Must. We choose to collect "may point to" information over "must point to" information because it handles subsequent dataflow analyses better. For example, if a pointer could take on two possible values, then we can conclude that an assignment through that pointer will modify one of the targets. In this case, we conservatively update the information for both possible targets. When a pointer has only one possible target, the "may" and "must" information are equivalent. When this occurs, assignments through the pointer are called *strong updates*, which can overwrite existing information about the target. Strong updates greatly improve analysis results [23].

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 a mechanism for software reuse. So library calls often occur deep in the application's call graph, and the same library functions are repeatedly invoked from different locations. Without context and flow sensitivity, the analyzer merges together information from the different call sites. For example, consider a simple PLAPACK program that makes two calls to the split routine with different matrices (different data objects):

```
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 analysis that use this information, such as constant propagation and available expressions, suffers the same merging of information. For example, even if analysis concludes that `B` contains all zeros before the split, if the state of `A` is unknown then it cannot safely infer that `B_upper` and `B_lower` contain all zeros.

The more reuse that occurs in a system, the more important it is to keep information separate. We implement full context sensitivity to prevent as much information merging as possible. Recent work has also shown that more precise pointer analysis not only causes subsequent analyses to produce more precise results, but it also causes them to run faster [19].

3.3 Annotations for Dependence Analysis

Our annotation language provides a mechanism for explicitly describing the effects of each routine on 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. Otherwise, for the application and for libraries that have not been annotated, our compiler analyzes the source code using the pointer analysis described above. Figure 4 shows the annotations for the PLAPACK matrix split routine.

```
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.

In some cases, a modern compiler could derive this information automatically from the library source code. However, there are situations when this is undesirable, infeasible, or impossible. Many libraries encapsulate functionality for which no source code is available, such as low-level I/O or interprocess communication. Even with the source code, it is easier to provide the information declaratively, especially if it is well known. Finally, we may want to model abstract relationships that are not explicitly represented in the library implementation. For example, a file descriptor logically refers to a file on disk through a series of operating system data structures. Using annotations, we can explicitly represent the file and it's 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`. The `on_entry` and `on_exit` annotations convey the effects of the procedure on pointer-based data structures by describing 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 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 can alter the relationships between the 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 and assigns the name __data to its target. The on_exit annotation declares that the split procedure creates two new objects, __view_2 and __view_3. The resulting pointer relationships correspond to the configuration depicted earlier 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 library calls that are dead code, and remove them. The compiler can also identify purely functional routines by looking at whether the objects accessed are different from those that are modified. Loop invariant code motion and common subexpression elimination can then process these routines as they would any other operators.

One issue with library-level optimizations is the complexity of obtaining such precise pointer information. Fortunately, recent research in pointer analysis has shown that efficient implementations are possible [23].

4 Library-Level Optimizations

Our system performs library-level optimizations by combining two tools: dataflow analysis and pattern-based transformations. The two tools are configurable, so that each library can have its own analyses and transformations, and the tools complement each other to enable powerful optimizations. Patterns work well for identifying local syntactic properties and for specifying transformations, but cannot easily express properties about the context within the overall program. Dataflow analysis concisely describes global context, but cannot easily describe complex patterns.

Both dataflow analysis and pattern matching have a wide range of realizations, from simple to complex. By combining them, each individual tool is simplified. The patterns need not capture complex context-dependent information because such information is better expressed by the program analysis framework. This combination also keeps the annotation language itself simple, making it easier to express optimizations and easier to get them right. 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 dataflow 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. The combined system allows the annotations to exploit the best attributes of each tool. The following sequence outlines our process for library-level optimization:

1. **Pointers and dependences.** Analyze the program using combined pointer and dependence analysis, referring to the annotations when necessary to incorporate library behavior.
2. **Classical optimizations.** Apply the 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 may specify additional constraints. The constraints are expressed in terms of dataflow dependences and the results of the abstract interpretation.
6. **Actions.** When a pattern satisfies all the constraints, perform the specified code transformation.

This process supports a variety of library-level optimizations for variety of different libraries. In next part of this section we describe the mechanisms in more detail and show how we can use them to take advantage of the opportunities mentioned in Section 2. Finally, we suggest the annotations needed to specify the optimizations.

4.1 Configurable Dataflow Analysis

In Section 3 we saw how annotations can integrate library functions into pointer and dependence analysis. However, such optimizations often depend on information that is specific to the library. The annotations provide a way to describe how the application uses the library routines and data structures. To accomplish this, we provide a configurable interprocedural dataflow analysis framework.

Dataflow analysis derives information by approximating the execution of the program. The state of the program is represented by an abstract *flow value*, and the computations are represented by *transfer functions* which operate on the flow values. 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 supplies 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 program 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 must 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 avoid exposing the underlying lattice theory to 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 between 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. The 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. The following table summarizes the available types. Each 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 a class of user-defined types that behave like enumerations in C, except that they also 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.

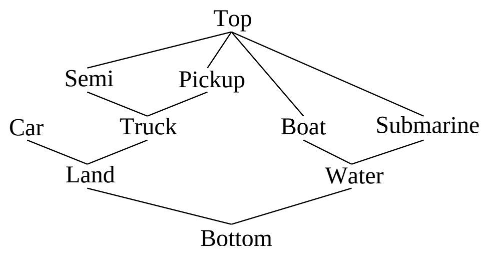


Fig. 5. The lattice induced by the example vehicle categories.

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, a transfer function is used to summarize the effects of that library routine 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, as 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 { }

  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. In this case, the possible property values are `General`, `ColPanel`, `RowPanel`, `Local` and `Empty`. 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

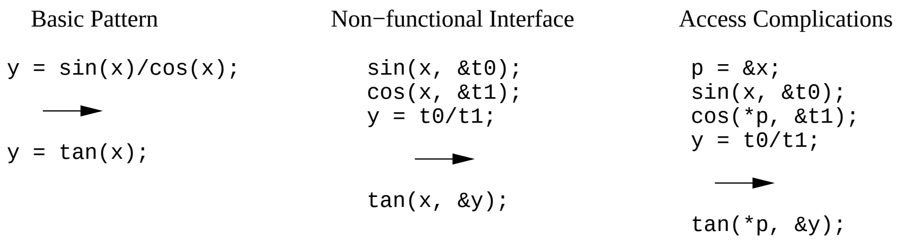
The library-level optimizations themselves are best expressed using pattern-based transformations. Once the dataflow analyzer has collected whole-program information, many optimizations consist of identifying and modifying localized code fragments. Patterns provide an intuitive and configurable way to 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 that take into account the states of the parameters.

For our pattern-based transformations, we need to identify library calls or sequences of library calls, we need to check the callsite against the dataflow analysis results, and we need 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 dataflow 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 differs somewhat from other pattern matching systems and is influenced by some of the same 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 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; rather, it will appear as a sequence of statements with data dependences between 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.

To address these problems, we only offer two meta-variable types, one that matches objects (both variables and heap-allocated memory), and one that matches constants. The object meta-variable elides 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`



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

y = tan(x);
```

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

tan(x, &y);
```

```
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.

and `*p`. The constant meta-variable can either match a literal constant in the code, an constant expression, or bind to 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 same sequence of statements, with any dataflow 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 it raises some difficult issues for pattern substitution.

Preconditions. The results of the pointer analysis and user-defined dataflow analyses can't be conveniently represented in the syntactic patterns. Therefore, the preconditions provide a way to test these additional constraints. 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 a column panel."

Actions. When a pattern matches and the preconditions are satisfied, then the compiler can perform the 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. For the *object* meta-type, we can choose to expand the meta-variable back to it's original code, such as `*p` in the example above, or to the actual objects, such as `x`.

In addition to pattern replacement, we offer four other actions: (1) remove the matching code, (3) move the code elsewhere in the application, (4) insert entirely new code, or (5) 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 it's 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.

```
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.

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 attempts to extend previous work in classical optimizations [18], partial evaluation [3, 7], abstract interpretation [8, 16], and pattern matching to library-level optimizations. In this section, we relate our work to other efforts aimed at providing configurable compilation technology.

The Genesis optimizer generator produces a compiler optimization pass from a declarative specification of the optimization[22]. 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 [17] 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 optimizing 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 invocations of `malloc()` when performing pointer analysis. Our goal is to provide compiler support in a configurable manner so that it can apply to many libraries, not just a favored few.

Meta-programming systems such as meta-object protocols [5], programmable syntax macros [21], and the Magik compiler [10], can be used to create customized library implementations, as well as to extend language semantics and syntax. While these techniques can be quite powerful, they require users to manipulate AST's and other compiler internals directly and with little dataflow information.

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 shown how configurable dataflow analysis and pattern matching can combine to form a powerful tool for performing library-level optimizations.

A large portion of our Broadway compiler is currently complete. We have implemented a flow- and context-sensitive pointer analysis, a configurable abstract interpretation pass, and the basic annotation language [14] without pattern matching. Experiments with this basic configuration have shown that significant performance improvements are possible for applications written in the PLAPACK parallel linear algebra library. One common routine, `PLA_Trsm()`, was customized to improve its performance by a factor of three, yielding overall speedups of 26% for one application (Cholesky factorization) and 9.5% for another (a Lyapunov program) [14].

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 *Proceedings of the International Conference 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. David R. Chase, Mark Wegman, and F. Kenneth 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. Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In ACM, editor, *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993*, pages 232–245, New York, NY, USA, 1993. ACM Press.
7. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, 1993.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
9. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994.
10. 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.
11. Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.
12. Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.
13. Torbjorn Granlund. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, April 1996.
14. Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, pages 39–52, October 1999.
15. Samuel Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In *IFIPS Working Group 2.5: Software Architectures for Scientific Computing Applications*, (to appear) October 2000.
16. 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.
17. Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
18. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, San Francico, CA, 1997.
19. 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.
20. Robert van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.

21. Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.
22. Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. *ACM SIGPLAN Notices*, 26(6):120–129, June 1991.
23. Robert P. Wilson. *Efficient, Context-sensitive Pointer Analysis for C Programs*. PhD thesis, Stanford University, Department of Electrical Engineering, 1997.