Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries

Samuel Z. Guyer, Calvin Lin

Abstract— **This paper describes the Broadway compiler and our experiences using it to support domain-specific compiler optimizations. Our goal is to provide compiler support for a wide range of domains and to do so in the context of existing programming languages. Therefore we focus on a technique that we call *library-level optimization*, which recognizes and exploits the domain-specific semantics of software libraries. The key to our system is a separation of concerns: compiler expertise is built into the Broadway compiler machinery, while domain expertise resides in separate *annotation* files that are provided by domain experts. We describe how this system can optimize parallel linear algebra codes written using the PLAPACK library. We find that our annotations effectively capture PLAPACK expertise at several levels of abstraction, and that our compiler can automatically apply this expertise to produce considerable performance improvements. Our approach shows that the abstraction and modularity found in modern software can be as much as asset to the compiler as it is to the programmer.**

Index Terms— **Library-level optimization, domain-specific, compiler**

As the study of compilers has matured, the capabilities of traditional optimizing compilers have been stretched to the limit, with increasingly complex optimization algorithms that yield diminishing returns. This situation reflects the extreme difficulty of wresting ever better performance from the same set of low-level programming language constructs. By contrast, domain-specific languages offer new high-level primitive operations for compilers to analyze and manipulate. Thus, domain-specific compilation is one of the few remaining frontiers for optimizing compilers.

A number of domain-specific compilers already exist. Some are tied to a particular language, such as MATLAB [8], [11], [31] and Spiral [29], while others are tied to particular domains, such as sparse matrices [9], [27]. A great strength of these approaches is the compiler's ability to make important assumptions about a specific domain. A problem with these approaches is their limited applicability: the capabilities that they provide do not translate to other languages or other domains. Such approaches make it difficult for programmers to use multiple domains in a single program, and make it difficult to develop compilers for new domains.

In this paper we describe an alternate approach to domain-specific compilation that supports a wide range of domains with a single compiler infrastructure. The key idea is to separate general-purpose compiler mechanisms from domain-specific information and expertise. Rather than hard-code domain knowledge into the compiler, our solution encapsulates such information through the use of a small annotation language. By using different annotations for different domains, our configurable compiler, called the Broadway compiler, can target multiple domains, even within the same program.

Our solution targets software libraries rather than domain-specific languages, and our annotations identify library routines that serve as domain-specific operations. During compilation, the annotations guide the Broadway compiler in its analysis and optimization of these operations.

Our design yields a number of benefits. First, as mentioned above, it allows us to handle multiple domains. A great many more domains are provided as libraries than are formulated as languages. Second, since libraries are already widely used, we can bring domain-specific compilation to existing software with little or no change to current programming practices. Third, the incremental cost of adding new domains or new optimizations is low because it does not require modifications to the compiler. The Broadway compiler consists primarily of domain-independent compiler machinery, such as a dataflow analysis framework, a code transformation system, and a set of traditional optimization passes.

In the spectrum of domain-specific compilers, Broadway represents a tradeoff that favors breadth of applicability over depth of capabilities. Our system produces competitive results for many domains but often cannot compete with custom compilers for individual domains. In deciding what to include in Broadway, we consider the general utility of the feature and the complexity of configuring it from an annotation language. Dataflow analysis, for example, is a general static analysis technique for collecting information about the dynamic behavior of a program. Our formulation of dataflow analysis only supports a subset of analysis problems, which limits the annotation writer's exposure to the underlying lattice theory. Nevertheless, these capabilities can express a wide variety of domain-specific compilation tasks, from optimizing parallel linear algebra codes to checking system software for security vulnerabilities [17], [21].

This paper presents an overview of our system, explains its design rationale, and summarizes our results and experiences. The remainder of this paper is organized as follows. Section I reviews related work. Section II describes the Broadway system and the general notion of library-level optimization. Section III then explains how the Broadway system can be applied to a parallel linear algebra library. Section IV presents performance results, and Section V concludes and identifies prospects for future work.

I. Related work

Many domain-specific compilers are designed expressly for particular domains. These systems include Matrix++ for

optimizing matrix operations based on a specification of the matrix structure [9], the Falcon compiler for optimizing MatLab programs [11], and the FFTW system for generating fast Fourier transform implementations [15]. By focusing on a single domain, such systems can employ the best representation and the most aggressive optimizations for that domain. By contrast, the Broadway compiler and annotation language are not tailored to any specific application domain, and thus trade off power for generality.

The Telescoping languages [23] project shares many goals and ideas with our research. A recent proposal describes an annotation language similar to ours that captures properties of library interfaces [5]. Currently, this work focuses on generating optimized code from scripting languages, such as MatLab [6]. The system uses offline compilation to generate alternative library implementations and uses a type inference system to select from among these. Broadway targets general-purpose programming languages—which support a wider range of domains—and uses dataflow analysis to drive optimizations. Dataflow analysis can solve many type inference problems but can also track the state of a computation, which is awkward to express using types.

It is useful to view Broadway as a specific instance of a system for supporting *active libraries* [36]. Active libraries represent a broad class of reusable software components that, unlike traditional libraries, are actively involved in the compilation process. One category of active libraries is self-tuning libraries, such as ATLAS [10]. Another category includes meta-programming techniques, such as expression templates [35], programmable syntax macros [37] and meta-object protocols [7], [24], which allow library developers to specify code customizations that take effect at application compile time. While these techniques provide powerful compile-time program manipulation capabilities, they suffer from two major drawbacks. The first is that the meta-program itself is often complex and error prone. The second is that these systems work primarily at the syntactic level—none of them include advanced program analysis. Our system drives program transformations using dataflow analysis, which captures deep semantic information about how programs work, and our system operates on a larger scope than typical meta-programs.

Previous research has also explored a number of different ways of making compilers more flexible, and often this work is motivated by the desire to support high-level and domain-specific compilation. Existing approaches, however, take a considerably different view of the tradeoff between usability and power. In particular, they often provide a more comprehensive set of tools for defining new compiler components, but using these tools can be difficult and error prone. Examples include open compiler infrastructures such as SUIF [22], optimizer generators such as Genesis [38], the Magik compiler [14], and analyzer generators such as Sharlit [33] and PAG [28]. Our view is that such approaches are not viable for domain-specific compilation because they require too much compiler expertise to be used by most domain experts.

II. Broadway

In this section we describe various opportunities for library-level optimization and we show how the Broadway compiler exploits them.

A. Library-level optimization

Software libraries provide a simple and flexible mechanism for including domain-specific functionality in general-purpose programming languages. A library represents the types and operators of a domain as a collection of ordinary datatypes and procedures, so programmers can add domain-specific functionality to any application simply by including the appropriate header files and making calls to the library interface. Programmers can develop new libraries for new domains at any time and can easily combine substantially different domains in a single application.

Unlike the built-in operators of a programming language, however, the domain-specific operators of a library have no special meaning to the compiler. As a result traditional compilers cannot analyze and optimize these operators, leaving the task of using libraries correctly and efficiently entirely to the programmer.

The goal of the Broadway project is to solve this problem by providing a new kind of compiler that can perform *library-level analysis* and *library-level optimization*. Our strategy is to extend existing compiler mechanisms so that they can recognize and manipulate library routine calls in much the same way that they do for built-in operators. We show that the same mechanisms that have proven effective for the built-in operators are also effective for library operators. The key is to capture some of the domain-specific information about each library in a form that a compiler can use.

1) Opportunities: Our compiler targets three kinds of library-level optimizations:

- **Domain-independent extensions.** These optimizations are direct extensions of traditional optimizations, such as dead-code elimination and loop-invariant code motion, applied to library routine calls. They are enabled by simple dependence information about each library routine. For example, by knowing that many of the math library operators have no side-effects the compiler can safely hoist them out of loops.
- **Single domain optimizations**. Library interfaces often include a range of different routines, many of which are designed to provide better performance for special cases. For example, a matrix library might have special routines for handling triangular matrices. Knowing when and how to use such routines can require significant expertise. This class of optimizations automates the selection of specialized routines, including identifying opportunities and ensuring correctness.
- **Cross domain optimizations**. The layering and encapsulation of library code often carries a performance penalty. Libraries are compiled ahead of time, making it difficult to take advantage of information about how they are used in particular applications. Cross domain optimizations provide a systematic way to break open the layers,

Library	Property	Action
Linear algebra	Special forms: triangular, tridiagonal	Replace general purpose routines with special-purpose routines
Graphics	Drawing state: lighting and shading options	Inline and specialize routines to avoid unused filters
File access	File state: open or closed	Report error if accessing a closed file
Threads	Lock state: locked, unlocked	Report double locking bugs
PLAPACK	Matrix distribution: row and column distribution	Remove unused code that handles distributed cases
(many libraries)	Validity: check or unchecked	Remove redundant error checking

Fig. 1. Libraries often have domain-specific properties that can be exploited by a compiler, but only if it is aware of them.

exposing a library's implementation to the compiler in the context of the application. These optimizations combine the application and libraries into a single integrated and customized piece of code.

B. Identifying optimization opportunities using typestate

The central task in library-level optimization is to identify places where it is both legal and profitable to change the library calls in an application. The changes themselves are typically straightforward: single domain optimizations replace a general-purpose library call with a special-purpose call or with a series of low-level library calls; cross domain optimizations consist of inlining followed by further optimization. The challenge lies in expressing the conditions under which an optimization applies and in identifying the parts of the application that satisfy the conditions.

The conditions that enable library-level optimizations are analogous to those of traditional optimizations: they depend on the states of the objects or on some global fact about the state of the computation. The difference is that for library-level optimization these states are expressed in terms of the library domain. For example, a series of linear algebra computations might result in matrices with special forms, such as triangular or tridiagonal. By knowing that the matrices have these characteristics, the compiler might be able to replace a general matrix multiply call with a more efficient one that exploits the special forms. We refer to these domain-specific attributes (such as "special form") as *properties*, and we refer to their possible states (such as triangular or tridiagonal) as *property values*. Many aspects of efficiency and correctness can be expressed as conditions on the property values of objects. Table 1 shows several concrete examples of libraries and associated properties, along with the actions that a compiler might take upon deriving this information. Our previous work on automatic error detection includes detailed descriptions of several such properties designed to model the domains of file manipulation and network security [17], [21].

Our notion of a property is a form of *typestate* [32]: it includes features of both types and states. They behave as types because property values can be related by a subtyping relation. For example, diagonal matrices can be defined as subtypes of triangular matrices. Specifically, the property values for each property are explicitly organized into a lattice (See Section II-D.2 for details). Properties can also behave as states, because they can have different values at different points in the program. For example, a matrix might start out empty, then be filled with values, then finally be factored into a triangular form.

The overall Broadway optimization process, based on typestate properties, proceeds as follows:

1) A library expert designs a set of domain-specific optimizations and encodes this information using the annotation language. This step involves:
   - Identifying abstract properties and property values that are significant for the library.
   - Specifying the behavior of each library routine in terms of these abstract properties.
   - Specifying code transformations for each library routine, which are predicated on the property values of the arguments.
2) The application programmer obtains the annotation file and passes it to Broadway.
3) During compilation, Broadway consults the annotations to determine how to interpret and manipulate the library routine calls:
   - It solves the typestate analysis problem, yielding an assignment of property values to objects in the application.
   - It evaluates the predicates at each call site and applies the code transformations where the predicates are true.

C. Beyond performance

Typestate information can also be used to detect *incorrect* or *unsafe* uses of library routines. In these cases, the results of typestate analysis are used to emit compile-time error messages rather than to guide code transformations. This feature helps produce more robust systems and improves programmer productivity. Library-specific error messages are useful because they provide immediate feedback on coding mistakes, rather than delaying problems to run-time. In addition, this facility can be used to check for deeper security flaws [17], [21]. Security has become more critical for high-performance computing with the growing popularity of grids and distributed computing.

D. Annotation language

In the Broadway system, all domain-specific information is provided as annotations: each library specifies its own analysis problems, code transformations, and error messages. Here we give a brief overview of the language; a more complete discussion can be found elsewhere [16], [19], [20]. The annotation language conveys four kinds of information to the compiler: (1) dependence information about the library interface, including the uses, defs, and pointer structures manipulated by

each routine, (2) domain-specific program analysis problems, which are solved by the compiler's analysis framework, (3) domain-specific optimizations, which are expressed as code transformations and are contingent on the analysis results, and (4) compile-time messages, which emit messages on the command line according to the analysis results.

The annotation file for a library is organized around the routines that make up the interface. Each routine has an entry in the file that contains all of the information related to the routine. In addition, there are several global annotations, which apply to all routines. We show the grammar for each kind of annotation and use the standard C library for examples.

The grammar descriptions below adopt the following conventions: *italics* indicate non-terminals, `teletype` indicates literal keywords, and anything in SMALLCAPS is an identifier.

procedure	→	**procedure** PROCNAME **(** *identifier-list* **)** {
		(*pointers* | *usesdefs* | *analyze* | *transform* | *report*) ⋆
		}
pointers	→	**on_entry** { *structure* ⋆ }
	|	**on_exit** { *structure* ⋆ }
structure	→	VARNAME
	|	VARNAME **-->** [**new**] *structure*
	|	VARNAME { *structure* ⋆ }
	|	**delete** VARNAME
usesdefs	→	**access** { *identifier-list* }
	|	**modify** { *identifier-list* }

(a) Dependence annotation grammar

```
procedure fopen(path, mode) {
  on_entry { path --> path_string
             mode --> mode_string }
  access   { path_string, mode_string }
  on_exit  { return --> new file_handle }
}

procedure fclose(file) {
  on_entry { file --> file_handle }
  on_exit  { delete file_handle }
}
```

(b) Example

Fig. 2. Pointer and dependence annotations provide a way to describe how the library traverses and updates pointer-based data structures. The --> operator declares a "points-to" relationship.

1) Library interface information: All of the annotations for a library routine are enclosed in a `procedure` annotation. Figure 2 shows the grammar for this part of the language, along with example annotations for two standard file I/O routines. Within the `procedure` annotation each library routine has a set of four pointer and dependence annotations that describes its behavior:

- The **on_entry** annotation describes the pointer structures expected as input to the library routine. This annotation tells the compiler how to traverse the pointer structures and provides names to the internal objects. The arrow **-->** can be thought of as a "points-to" operator, which leads to a natural declarative description of pointer structures.
- The **access** annotation identifies the objects that the routine accesses. This list can refer to variables from the interface or to names introduced by the **on_entry** annotations.
- The **modify** annotation identifies the objects that the routine modifies. Since this information is only used for dependence analysis there is no need to describe *how* the routine modifies them.
- The **on_exit** annotation describes any changes to the pointer structure effected by the routine.

The example in Figure 2 (b) indicates that the `fopen` routine takes two pointers as arguments, it accesses the targets of those two pointers, and it allocates and returns a new object. At each callsite, the compiler binds the actual objects involved to the names given in the annotations.

In compiler terms, the `access` and `modifies` annotations specify the "uses" and "defs" of the routine, respectively. Note that a pointer dereference is automatically recorded as an access, and a pointer update is automatically recorded as a modification.

With this information Broadway can compute data dependence information for the whole application, including library calls. This information includes a model of heap objects, pointer aliases, and use-def chains. This dependence information allows our compiler to perform on library calls a number of traditional optimizations, such as dead-code elimination.

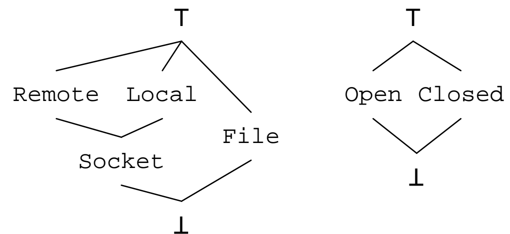
property	→	**property** PROPNAME **:** [*dir*] *property-vals*
		[**initially** PROPVAL]
property-vals	→	{ *property-val-list* }
property-val-list	→	*property-val* [**,** *property-val-list*]
property-val	→	PROPVAL [*property-vals*]
dir	→	**@forward**
	|	**@backward**

(a) Property annotation grammar

```
property FileState : { Open, Closed }  initially Closed

property Kind : { Socket { Local, Remote }
                  File }
```

(b) Example property definitions



(c) Lattices for the properties above

Fig. 3. Property annotations define flow-values for library-specific analysis passes. The syntactic structure implies the underlying lattice.

2) Library-specific analysis passes: As mentioned above, library-level optimizations and error messages are triggered by library-specific typestate information (properties). An annotation writer defines a property by specifying (1) the name of the property and its set of property values, and (2) the effects of each library routine on these values. The property values are

organized into a lattice, which expresses the subtyping relation. Broadway uses dataflow analysis to push the property values through the application and derive an assignment of property values to objects at different points in the program.

The property and property values are defined using a `property` annotation, and the effects of the library routines are defined using `analyze` annotations (one for each routine). Figure 3 shows the `property` annotation grammar, including examples of the annotations and the lattices that they imply. Properties also have a direction, forward or backward, which indicates which way the information flows in the program. Backward properties are useful for describing how objects will be used at a later point in the program's execution. The annotation writer chooses the direction of information flow, and the compiler uses the appropriate algorithm to propagate the information.

The `analyze` annotations describe how each library routine updates the property values of objects. Figure 4 shows the grammar for these annotations along with examples for the standard C library. Analyze annotations can be unconditional, as in the `fopen` example, which always produces a file handle in the "open" state. Analyze annotations can also be guarded, as in the `socket` example, which produces either a "remote" socket or a "local" socket, depending on the domain argument. The property value tests are described below.

```
analyze        →  analyze PROPNAME { analysis-rule * }
               |  analyze PROPNAME { effect * }

analysis-rule  →  if ( condition ) { effect * }
               |  default { effect * }

condition      →  [ PROPNAME : ] test
               |  numeric-comparison
               |  condition || condition
               |  condition && condition
               |  ! condition

test           →  VARNAME is-??
               |  VARNAME is-exactly PROPVAL
               |  VARNAME is-atleast PROPVAL
               |  VARNAME could-be PROPVAL
               |  VARNAME is-atmost PROPVAL

effect         →  VARNAME <- PROPVAL
               |  VARNAME <- VARNAME
```

(a) Analysis annotation grammar

```
procedure fopen(path, mode) {
  on_entry ...
  on_exit ...
  analyze Kind      { file_handle <- File }
  analyze FileState { file_handle <- Open }
}

procedure socket(domain, type, protocol) {
  on_entry ...
  on_exit ...
  analyze Kind {
    if (domain == AF_INET) { return <- Remote }
    if (domain == AF_UNIX) { return <- Local  }
  }
}
```

(b) Examples

Fig. 4. Analysis annotations specify how library routines affect property values. In the compiler these serve as transfer functions.

3) Optimizations and error reports: Library-level optimizations and error reports are defined on a per-routine basis. Figure 5 shows the grammar for these annotations, along with example uses. An optimization consists of a code transformation and a guarding condition. Broadway applies the code transformation at each callsite for which the condition is true. Similarly, an error report consists of a message to emit and a guarding condition.

The guarding conditions on these annotations test the results of the property analysis. We provide several operators to test the property values and their lattice relationships:

- **`is-exactly`**: evaluates to true only if the object on the left-hand side ends up with exactly the property value on the right-hand side.
- **`is-atleast`** and **`is-atmost`**: these two operators represent lattice "less-than-or-equal" and "greater-than-or-equal" tests, respectively. In the socket example of Figure 3, property value `Local` **`is-atleast`** `Socket`.
- **`is-??`**: this unary operator evaluates to true if the object ends up with lattice bottom.
- **`could-be`**: this operator evaluates to true if at any point in the program the object ever takes on the value on the right.

The optimization annotations can specify one of two transformations to apply at each satisfying callsite: (1) replace the library call with an arbitrary C code fragment, or (2) inline the library routine implementation (assuming library source is available). The replacement mechanism works much like hygienic macros [26]: the compiler parses and checks code fragments, and then ensures that expansion results in syntactically correct code. Figure 5 shows a code replacement example for the `fgets` routine: when the size is 1 we can just use `fgetc` and store the returned character directly in the string. The compiler will replace the `$s` token with the actual argument at the call site.

The code replacement facility is most useful for expressing single-domain optimizations, and the inlining facility is most useful for cross-domain optimizations because it exposes the implementation of a library routine. As with the code replacements, inlining can be made contingent on the results of dataflow analysis. This feature enables domain-specific inlining policies, which help ensure that inlining takes place only when it is likely to be beneficial.

Figure 5 also shows an error report example: the compiler will emit a message whenever a program attempts to read from a file handle that is not open. The compiler will replace the `@callsite` token with the line number and file name of the erroneous the call site.

E. Compiler design

Broadway is a source-to-source translator for C written in C++. It is built on top of the C-Breeze compiler infrastructure [18] and inherits many components from it, including the front-end parser, internal representation, and a suite of traditional compiler analysis and optimization passes. Figure 6 shows the overall architecture of the system. Broadway takes

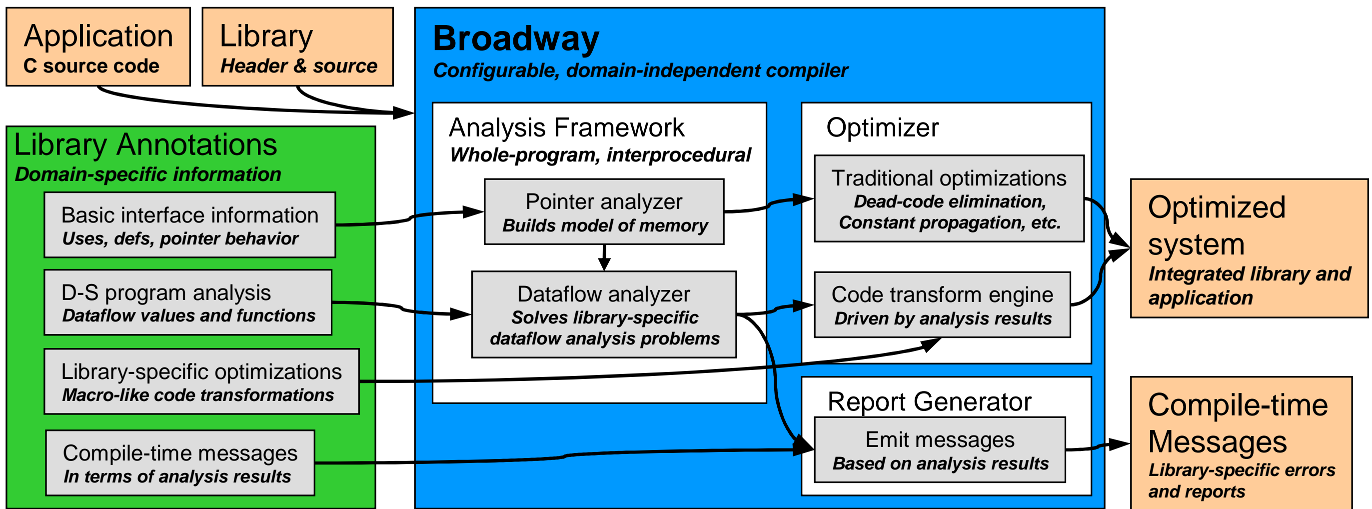
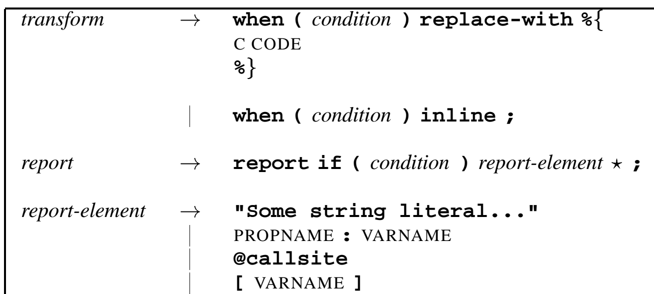


Fig. 6. The architecture of the Broadway compiler: an annotation file accompanies the usual library files and tells the compiler how to perform library-level analysis and optimization on applications that use the library.



(a) Action annotation grammar

```
procedure fgets(s, size, f) {
  on_entry ...
  on_exit ...

  when (size == 1)
    replace-with %{ (*${s}) = fgetc(f); }%

  report if ( ! FileState : file_handle is-exactly Open)
      "Error␣at␣" ++ @callsite ++ ":␣File␣is␣not␣open\\n";
}
```

(b) Examples

Fig. 5. Action annotations specify optimizations and error reports.

as input an application written in ANSI C along with annotation files that describe the libraries used by the application. During each compilation phase the compiler consults the annotations to determine the effects of each call to the library.

Two compiler mechanisms are central to our domain-specific optimizations: (1) a configurable dataflow analysis framework, which solves the domain-specific typestate analysis problems given by the annotations, and (2) a code transformation engine, which tests the conditions and applies the optimizations specified by the annotations. In addition, the error reporting mechanism visits all library call sites and emits any messages specified by the annotations.

The dataflow analysis framework uses a traditional iterative analysis algorithm to solve each library-specific analysis problem [1], [25]. It also includes a number of powerful features that improve both its precision and its scalability. We found these features critical for exploiting the opportunities presented by real, industrial-strength software.

First, the framework includes an integrated pointer analyzer that provides alias information for surface variables, as well as a detailed model of heap-allocated structures. Pointer information is critical for library-level optimization because almost all non-trivial library objects are accessed through pointers. Many of these objects also have internal structure and are represented as pointer-based data structures. Since dataflow dependences might exist between internal components we must have a sound model of memory to avoid applying optimizations incorrectly.

Second, the framework employs an interprocedural, whole-program analysis, allowing the compiler to gather information about library routine usage over a large scope. Unlike built-in language operators, library routines are not bound by simple lexical scoping rules. In conjunction with the use of pointers, library objects can flow throughout a program. Whole-program analysis is required not only for correctness, but is also valuable for exposing optimization opportunities.

Third, the framework supports a range of analysis precision policies, including our own client-driven analysis algorithm [21], which automatically adapts its precision in response to the needs of the analysis problem. At its most precise, the system is flow-sensitive and context-sensitive, which provides accurate analysis information even for non-trivial applications, such as those with complex software architectures and heavy code reuse. This level of precision, however, can increase analysis costs to an intolerable level. The client-driven analysis algorithm provides both accuracy and scalability by applying flow-sensitivity and context-sensitivity only to the parts of the program where they are needed.

III. Optimizing PLAPACK

In this section we demonstrate the application of our technique to the PLAPACK parallel linear algebra library [34].

We first provide background about PLAPACK abstractions and their role in optimization: we present a layered decomposition of the PLAPACK system and describe the abstractions at each layer. We then describe the library-level optimizations that we specified for PLAPACK. We will show the impact of these optimizations in Section IV.

To explain our technique, we go into considerable detail about the target library and its abstractions, the mechanics of the optimizations, and their representation in the annotations. Before diving into these details, we enumerate the important points of this section:

- Complex domains, like parallel linear algebra, contain a wide range of potential optimizations. We show that our annotations can capture many of these optimizations with a small number of language constructs.
- The complexity of the PLAPACK interface makes it challenging for programmers to apply optimizations. The compiler mechanisms we provide help to overcome these difficulties by automating the process.
- Most of these optimizations are valid only under particular conditions that are highly domain-specific. Without the configurable dataflow analyzer, the compiler could not collect the necessary information.

A. Concepts

PLAPACK is a library for writing parallel linear algebra programs in C. It consists of approximately 45,000 lines of C code and provides parallel versions of many of the same kernel routines found in the BLAS [12] and LAPACK [2]. At the highest level, it provides an interface that hides much of the parallelism from the programmer.

A PLAPACK application operates on linear algebra objects, such as matrices and vectors, that are partitioned and distributed over a grid of processors on the target computer. The application manipulates these objects indirectly though handles called *views*. A view specifies a set of matrix indices that can be used for subsequent computations. PLAPACK provides routines to create views, shift views, and split views into pieces. Figure 7 shows a split that logically divides a matrix A into four smaller ones.

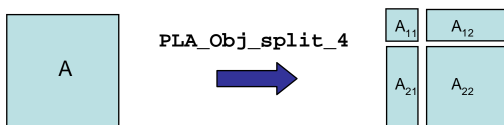


Fig. 7. PLAPACK algorithms operate at a higher level than traditional linear algebra algorithms by splitting matrices into logical pieces, called *views*, and operating on these views.

A typical algorithm starts with an entire object, such as A, and splits it into manageable pieces. It computes directly on A_{11}, A_{12} and A_{21}, and then continues iteratively by splitting the large remaining piece, A_{22}, until the entire data set has been visited. A view often captures part of a matrix that has special properties. Understanding and exploiting these properties can lead to significant performance improvements.

PLAPACK kernel routines, such as parallel matrix multiplication, are implemented using a lower level set of routines that make data distribution and movement explicit. At this level, the library creates objects with special distribution properties and then uses a communication routine, `PLA_Copy()`, to transfer data between them.

For example, Figure 8 shows how to compute an outer product from a matrix column panel and a matrix row panel. Initially, the column panel A resides on one column of processors, and row panel B resides on one row of processors. In the first step, B is duplicated on each row of processors. In the second step, A is duplicated on each column of processors. Both of these steps are accomplished using the `PLA_Copy()` routine. The result is that each processor contains the right pieces of A and B to compute its part of the outer product. The final step is for each processor to compute this part using a local matrix multiply routine. In PLAPACK parlance, *local* operations are sequential computations that serve as building blocks for their parallel counterparts.

B. Optimizations

We now describe the specific optimizations that we use to produce the results in Section IV. We categorize them according to the PLAPACK layer to which they apply. Figure 9 shows the three conceptual layers of the PLAPACK implementation. Each layer has its own programming abstractions, and thus its own optimizations. We derive our PLAPACK optimizations from a number of sources. In some cases, we codify techniques suggested in PLAPACK publications [3]. In other cases, we examine PLAPACK programs ourselves to determine possible performance improvements. When we discover a potential optimization, we determine the circumstance under which it applies and then formulate a program analysis pass to detect that circumstance.

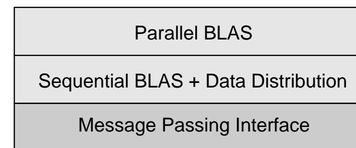


Fig. 9. Logical layers of the PLAPACK implementation. Many libraries consist of multiple layers, each with its own domain-specific semantics.

The simplest way to use PLAPACK is to program at the highest layer of abstraction because it provides the most powerful abstractions. It also leverages a large body of reusable code underneath. However, by working at this high level, programmers miss many optimization opportunities. Thus, programmers would ideally write code at the highest level and let a tool compile this code down to the lowest level. Our system provides a way to do this.

Global Layer: Parallel BLAS

The highest layer provides parallel linear algebra operations that hide parallelism from the application developer. It consists of operations that work on any view, regardless of where the data resides. At this level, optimizations work in terms of the matrix domain.

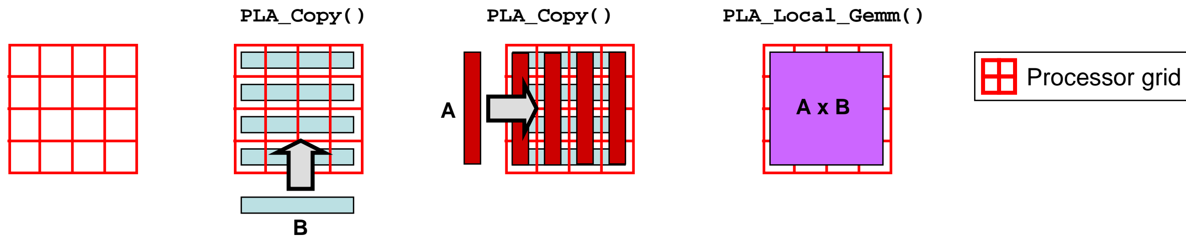


Fig. 8. Algorithm to compute distributed outer-product of two multi-vectors (matrix panels), A and B, using explicit data replication and local computation.

- **Scalar algebra.** The `PLA_Scal()` routine multiplies the elements of a matrix or vector by the given scalar constant. If the constant is known to be one, then the call has no effect and can be removed. If the constant is zero, we can replace the call with a special PLAPACK call that sets all elements to zero.
- **Matrix algebra.** Like the scalar algebra above, we can exploit the matrix multiplication identities. The `PLA_Gemm()` routine computes something of the form $C \leftarrow A * B + C$, so the optimizations are slightly different. If A or B are zero matrices, then the code has no effect and can be removed. However, if A or B is the identity matrix, then the call essentially computes a matrix addition. We can replace this call with code that explicitly adds the elements, which is an entirely local operation that requires no communication between processors.

Middle Layer: Data Distribution

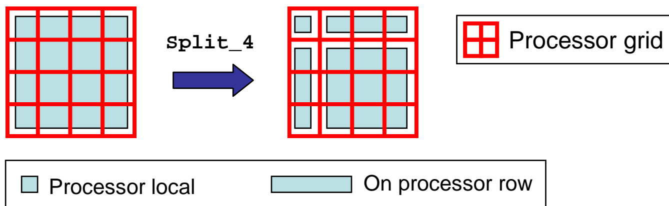


Fig. 10. PLAPACK distributes matrix data across the processors. Split operations often result in special-case distributions, such as sub-matrices that reside entirely on one processor.

The middle layer uses structured forms of communication to expose the notions of data distribution and locality. In Figure 10 we show the same four-way split as Figure 7 with an overlaid grid that represents the partitioning of the data over a grid of processors. The actual partitioning is more complex than a simple block distribution [13], but the basic observations still hold. The figure shows that a four-way split of A yields one *local* view (A_{11}), which resides entirely on one processor, one *column panel* (A_{21}), which resides entirely on a column of processors, one *row panel* (A_{12}), which resides entirely on a row of processors, and one view, A_{22}, which is a fully distributed submatrix. We can take advantage of such information to improve performance. In particular, algorithms that are designed to process distributed matrices can often be significantly simplified when customized for row panels or column panels. The middle layer is implemented as a number of sequential BLAS calls, which operate only on local pieces of data, and invocations of the `PLA_Copy()` routine, which move data around on the processor grid.

The most effective optimizations that we have found come from breaking open the global layer routines to expose their middle layer implementations. The reason is that the global layer routines are designed to work with any kind of linear algebra object, regardless of their size and distribution. However, applications often pass particular special distributions into these routines, and we can exploit this extra information to create a customized version of the routine for that particular distribution.

Rather than enumerate all of these special cases, we define a set of optimizations that together can transform a general-purpose implementation into a customized version:

- **Special-case routine selection.** Internally, many PLAPACK routines have multiple implementations that are specialized for different situations. For example, the general matrix multiply routine, `PLA_Gemm()`, is implemented internally as three different algorithms for different matrix shapes. At runtime the routine chooses from among the algorithms by comparing the relative sizes of the input matrices. Often we can use library-specific analysis to identify these cases at compile time, thus avoiding the runtime cost.
- **View optimizations.** We can often simplify the matrix splitting routines when the input view is already a special-case distribution. For example, there is no need to vertically split a column panel because it already resides on a single column of processors. Such optimizations can eliminate entire loops from the code.
- **Empty views.** Any computation on an empty view can be removed. The computational routines (for example, `PLA_Gemm()` and `PLA_Trsm()`) check for empty views already, but this is done at runtime and can incur synchronization overhead. Not only can we avoid this cost by removing the code at compile time, but the static removal of the code can expose additional optimization opportunities, such as dead code elimination.

Lower layer: MPI communication

The lower layer contains explicit communication using MPI, the Message Passing Interface. We have identified several optimizations at this level. For example, we could analyze the matrix splitting pattern in an application to determine where a point-to-point broadcast might yield software pipelining. However, these experiments require additional annotations that

we leave as future work.

C. Object type analysis

We are now ready to describe how we encode specific PLAPACK optimizations using the annotation language. In PLAPACK the `PLA_Obj` data type represents all linear algebra objects. However, the library can create and manipulate many different kinds of objects, such as matrices, vectors, and scalars (which are called *multi-scalars* when they are replicated across processors). The internal library data structures maintain this type information at runtime so that the various library routines can handle these objects in the appropriate manner. The `PLA_Copy` routine, in particular, needs to know the type of the objects to decide how to perform data copying.

We use the annotation language to track this information at compile time. Since object types are explicit in the creation routines, this analysis often succeeds at accurately determining their types statically. We use this information for two purposes. First, we can make sure that the types passed into a computation match the expected types. For example, the `PLA_Gemv` routine expects a matrix and a vector as input. We use the object type analysis to validate this requirement at compile time. If the compile-time check succeeds, we can improve performance by eliminating the runtime check. If the compile-time check fails, we issue an informative message describing the nature and location of the error, which allows the programmer to fix it without having to execute and debug the program.

The second use of the object type information is to perform algorithm selection at compile-time. In combination with the distribution analysis described below, we can often avoid the cost of the runtime switches that ordinarily make these choices. By itself, this optimization does not yield significant performance improvements. With runtime switches removed, however, the compiler can often inline and further optimize the implementation of the chosen algorithm.

The `ObjType` property, shown in Figure 11, provides names for the different kinds of linear algebra objects. The base types are matrix, vector, projected vector (Pvector), and multi-scalar (Mscalar). An ordinary vector is distributed over the processor grid in a manner that improves matrix-vector operations [13]. A projected vector is a vector that is distributed like a column or row of a matrix. Multi-vectors consist of several vectors stored together. A duplicated projected multi-vector is a projected multi-vector that is replicated across the rows or columns of the processor grid. Figure 8 shows graphically two examples of projected multi-vectors being copied to duplicated projected multi-vectors.

```
property ObjType : { Matrix,
                     Vector, Mvector,
                     Pvector, Pmvector, Dpmvector,
                     Mscalar }
```

Fig. 11. The ObjType property captures the different kinds of linear algebra objects supported by PLAPACK.

Figure 12 shows the annotations for the routine that creates matrices. Note that we associate the type with the view structure, which will allow us to change the type of an object when it suits the computation better. For example, we can treat a panel of a matrix as a projected multi-vector, which helps reduce the amount of work in the copy routine.

```
procedure PLA_Matrix_create(datatype, length, width,
                            template_ptr,
                            v_align, h_align, matrix_out )
{
  on_entry { matrix_out --> the_matrix }
  on_exit  { the_matrix -->
                  new the_view { length, width,
                                 data --> new data } }

  analyze ObjType { the_view <- Matrix }
}
```

Fig. 12. The object creation routines set the type of the object.

D. Distribution analysis

The most significant PLAPACK optimizations result from recognizing and exploiting special-case object distributions. Figure 13 shows the property annotations for tracking distribution. We define two separate properties, one for the rows of an object and one for the columns of an object, because the distribution of rows and columns can vary independently.

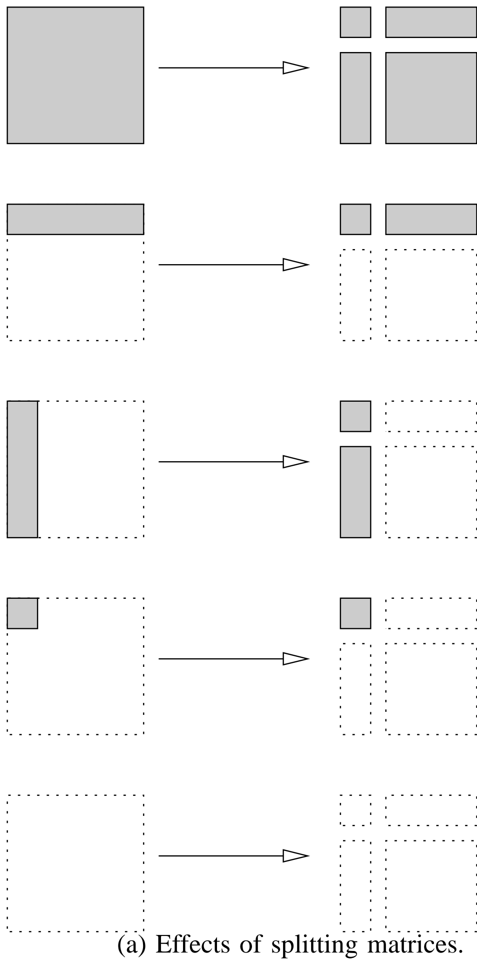
```
property RowDistribution :
            { Unknown { NonEmpty { Distributed,
                                   Local { Duplicated },
                                   Vector },
                        Empty } }

property ColDistribution :
            { Unknown { NonEmpty { Distributed,
                                   Local { Duplicated } },
                        Empty } }
```

Fig. 13. These two annotations describe the different ways that the rows and columns of a matrix can be distributed.

The distribution of an object is determined initially by the routine that creates it and subsequently by any splitting operations applied to it. Figure 14 (a) graphically depicts the effects of the `PLA_Obj_split_4` routine on the possible shapes of the input matrix. Figure 14 (b) shows representative analysis annotations for this routine, which codify the effects as a set of rules. The actual annotations contain all of the cases, and they model the ability of the routine to split a matrix relative to any of the sides of the matrix, not just the top left corner.

In many instances the split routine produces empty views, as often happens when a general-purpose routine, such as `PLA_Trsm()`, is specialized for a context where the input matrices are not fully distributed. The compiler can eliminate subsequent operations on these empty views. Figure 15 shows an example of this optimization. We see that if *any* of the dimensions of the inputs is empty, then we remove the call. Furthermore, consider a loop that repeatedly splits a matrix: if the matrix is already in the desired form, then the first iteration of the loop consumes all of the data and all other views are empty, so the loop can be removed.



(a) Effects of splitting matrices.

```
analyze RowDistribution {

  if (view_A is-exactly Distributed)
    { view_A11 <- Local
      view_A12 <- Local
      view_A21 <- Distributed
      view_A22 <- Distributed }

  if (view_A is-atleast Local)
    { view_A11 <- Local
      view_A12 <- Local
      view_A21 <- Empty
      view_A22 <- Empty }

  if (view_A is-exactly Empty)
    { view_A11 <- Empty
      view_A12 <- Empty
      view_A21 <- Empty
      view_A22 <- Empty }
}

analyze ColDistribution {

  if (view_A is-exactly Distributed)
    { view_A11 <- Local
      view_A12 <- Distributed
      view_A21 <- Local
      view_A22 <- Distributed }

  if (view_A is-atleast Local)
    { view_A11 <- Local
      view_A12 <- Empty
      view_A21 <- Local
      view_A22 <- Empty }

  if (view_A is-exactly Empty)
    { view_A11 <- Empty
      view_A12 <- Empty
      view_A21 <- Empty
      view_A22 <- Empty }
}
```

(b) Annotations codifying these effects.

Fig. 14. Analysis annotations for the `PLA_Obj_Split_4()` routine. Depending on the distribution of the input matrix, the split routine can create special case views or even empty views.

```
procedure PLA_Trsm(side, uplo, transa, diag, alpha, a, b)
{
  on_entry { alpha --> view_alpha
             a --> view_a
             b --> view_b }
  when (RowDistribution : view_a is-exactly Empty ||
        ColDistribution : view_a is-exactly Empty ||
        RowDistribution : view_b is-exactly Empty ||
        ColDistribution : view_b is-exactly Empty)
    replace-with %{ ; }%
}
```

Fig. 15. This annotation states that operations on empty views can be removed.

E. Special-case inlining

The first step towards generating customized routines is to expose the implementations of the global layer routines. We use library-specific analysis to decide when to perform inlining, so inlining is only performed where it is likely to be useful. For most of the level 3 BLAS routines, we use the following policy: if either the row or the column distributions of input objects is local, then inline the implementation. This policy exposes operations on local objects, which tend to yield the most benefit. Figure 16 shows the annotations for inlining the `PLA_Trsm()` routine, which performs a triangular solve with multiple right-hand sides.

```
procedure PLA_Trsm(side, uplo, transa, diag, alpha, a, b)
{
  on_entry { alpha --> view_alpha
             a --> view_a
             b --> view_b }
  when (RowDistribution : view_a is-atleast Local ||
        ColDistribution : view_a is-atleast Local ||
        RowDistribution : view_b is-atleast Local ||
        ColDistribution : view_b is-atleast Local)
    inline;
}
```

Fig. 16. This annotation uses dataflow analysis information to define a library-specific inlining policy.

F. Algebraic simplifications

At both the global layer and the middle layer, we define optimizations that take advantage of algebraic identities. Figure 17 shows two examples for the `PLA_Scal()` routine, which applies a scalar multiplier to all elements of a matrix. When the scalar is equal to one, the multiplication has no effect, and we can remove it. When the scalar is zero, we can avoid the multiplication operations and just set the matrix to zero.

```
procedure PLA_Scal(alpha, a)
{
  on_entry { alpha --> view_alpha { length, width,
                                    data --> data_alpha }
             a --> view_a  }
  when (data_alpha == 1.0)
    replace-with %{ ; }%
  when (data_alpha == 0.0)
    replace-with %{ PLA_Obj_set_to_zero( $a ); }%
}
```

Fig. 17. This annotation exploits domain-specific algebraic identities.

Such opportunities might seem to be rare, but they often appear after inlining. For example, the `PLA_Scal()` routine is called inside the implementation of `PLA_Gemm()` to handle the coefficients alpha and beta. In almost all cases these values are zero, one, or minus one. However, we cannot exploit this information until the routine is inlined.

G. Redundant copy removal idiom

PLAPACK programs use the copy routine to redistribute data so that they are in a suitable form for subsequent computations. If, however, the input submatrices are already suitably distributed for a given call site, no copying is necessary. This situation occurs in the specialization of the triangular solve routine `PLA_Trsm()`. Unfortunately, the current annotation language cannot express this optimization because it requires the compiler to recognize and replace a *sequence* of library calls. (We have defined the syntax for such an optimization and we anticipate having this capability in the future [21].) For the experiments in Section IV we show results for both the fully-automated system and results that include the copy remove optimization applied by hand.

IV. RESULTS

This section presents performance results obtained by applying our system to a set of PLAPACK applications and kernels. We find that the annotations effectively specify library-level optimizations and that these optimizations produce significant performance gains across layers of abstraction.

A. Methodology

We start with well-written versions of three PLAPACK programs, which serve as a baseline and represent our ideal programming style: the code clearly expresses the algorithm and is unobscured with hand-coded optimizations. The programs generally use the highest layer of PLAPACK, but they are by no means poor implementations. They perform competitively with similar programs written using other parallel programming technologies. We apply library-level optimization to all three programs using a single set of PLAPACK annotations.

We apply a series of optimization passes to each program. Each pass first performs the library-specific analysis, followed by the library-specific code transformations. We then apply a set of "cleanup" optimizations, including constant propagation, control-flow simplification, and dead-code elimination. We repeat this process until no new code transformations occur. We find that there is considerable synergy between these optimizations, so the process typically requires four or five iterations.

B. Programs

We use three programs for these experiments: (1) **Cholesky factorization**, (2) **LU factorization**, and (3) the **Kernel of a Lyapunov equation solver.** The baseline version of the Cholesky factorization is shown in Figure 18. The Lyapunov equation [4] arises in control theory applications. It is more complex than the other two and poses a more challenging optimization problem for our approach. The PLAPACK authors provided our baseline implementation [30].

```
while ( 1 ) {
   PLA_Obj_split_size( a_next, PLA_SIDE_TOP,
                       & size_top, & owner_top );
   PLA_Obj_split_size( a_next, PLA_SIDE_LEFT,
                       & size_left, & owner_left );

   if ( size = min(size_top, size_left )) break;

   PLA_Obj_split_4(a_next, size, size, & a_cur, PLA_DUMMY,
                                       & a_col, & a_next );

   PLA_Local_chol( uplo,  a_cur );

   PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
             PLA_TRANS, PLA_NONUNIT_DIAG,
             one, a_cur, a_col );

   PLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
             min_one, a_col, one, a_next );
}
```

Fig. 18. The main loop of the baseline Cholesky factorization.

C. Annotations

We have shown several examples of the PLAPACK annotations, but due to space limitations we do not include the entire annotation file. The following summary characterizes the annotation file and the annotation effort.

- The PLAPACK library consists of about 45K lines of C code. The PLAPACK annotation file consists of about 3400 lines of annotations.
- We annotated 85 PLAPACK routines. Each routine averages about 40 lines of annotations. While most routines require about 20 lines to annotate, several routines, such as the view splitting routines, require as many as 200 lines to handle all of the analysis cases.
- About 30% of the annotation file is devoted to the pointer and dependence information. In our current language this information must be repeated in each routine.
- The annotations define seven library-specific program analyses (property annotations). Only one of them, the ViewUsed property, is a backward analysis.
- There are 48 error reporting and debugging annotations.
- There are 70 code transformation annotations. Of these, the majority remove useless computations—e.g., computing on an empty view. Many others describe the conditions for inlining the implementation of a routine. This emphasis reflects our goal of generating customized code from general-purpose routines.

D. Platform

For these experiments we use Broadway as a cross compiler: we compile the programs locally on a Pentium 4 workstation running Linux, and then copy the source to the parallel environment, an IBM Power4-based multiprocessor. This multiprocessor consists of a tightly-bound network of three 16-way symmetric multiprocessors (SMP), one 32-way SMP, and 32 4-way SMPs. Each processor runs at 1.3Ghz. We compile using the vendor-supplied tools, and we link against the vendor supplied Message Passing Interface (MPI), which handles the non-uniform memory architecture.

E. Performance results

For each of the three programs, we measure the execution time of the baseline version and two Broadway optimized versions: one with the redundant copy idiom and one without (see Subsection III-G). For Cholesky factorization, we also time a version that is hand-optimized by the PLAPACK implementation team. We run each program on a range of input matrix sizes, from 1000×1000 to 8000×8000, and on a range of processor grids, from 2×2 processors to 10×10 processors.

We find the following general results:

- Our PLAPACK annotations consistently improve performance. Depending on the program, the problem size, and the number of processors the improvement ranges from just a few percent to 30 percent.
- Overall, the per-processor performance improvement increases as we increase the number of processors and decreases as we increase the problem size. This suggests that our annotations are effectively eliminating the software overhead associated with the library layers.
- While the redundant copy idiom noticeably improves performance, the rest of the annotations also contribute significantly.

For each program, we show the performance improvement obtained by using Broadway. The three program-specific graphs show the percent improvement in execution time over the baseline version for 64 processors (an 8×8 grid) over a range of problem sizes.

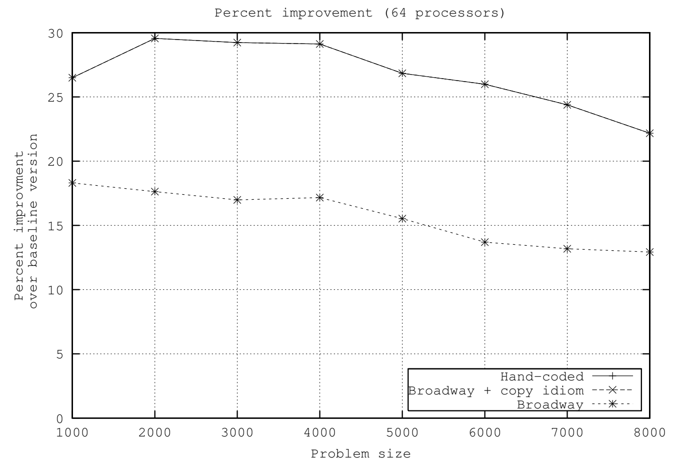


Fig. 19. Percent improvement for Cholesky on 64 processors. The curves for the hand-coded and Broadway+copy-idiom versions sit on top of one another.

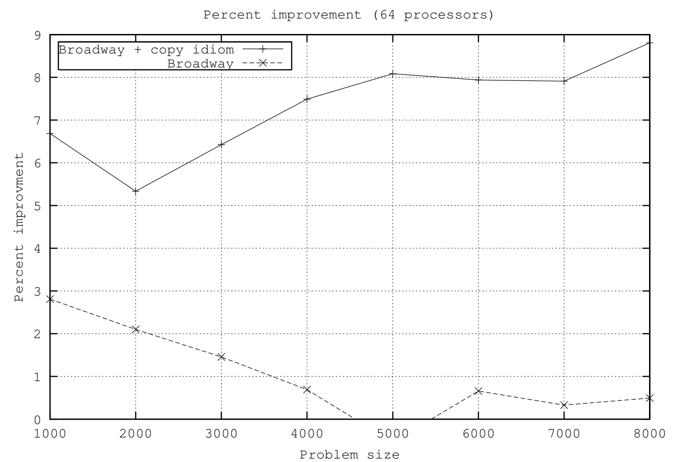


Fig. 20. Percent improvement for LU on 64 processors.

Figure 19 shows the results for the Cholesky factorization program. The code generated by the specialization strategy alone runs 13 to 18 percent faster than the baseline version. When we include the redundant copy idiom, the improvement jumps to between 22 and 29 percent. In this case, our Broadway-generated version runs as fast as the hand-coded Cholesky factorization written by the library authors, which serves as an upper bound for our approach. In fact, many of the optimizations codified in our annotations come from insights into this hand-coding process [3]. In annotation form, however, we can easily apply the same optimizations to other programs, including the other two test programs.

Figure 20 shows results for LU factorization. This program is dominated by calls the triangular solve routine, so the redundant copy idiom makes a significant difference. Manual inspection of the Broadway-generated code indicates that there are few additional optimization opportunities at the PLAPACK level of abstraction. (For problems larger than 4000 by 4000, the improvement obtained without the copy idiom is negligible, and even falls slightly below the baseline for 5000 by 5000.)

Figure 21 shows the result for the Lyapunov equation solver. These results represent a more significant test of our approach because of the program's complexity. The specialization strategy improves performance by 5 to 10 percent. The addition of the redundant copy idiom improves performance by 9 to 15 percent.

Figure 22 shows the results for all three programs on a large fixed-size problem, plotted against the number of processors. For Cholesky factorization and the Lyapunov solver, the library-level optimizations provide consistent and scalable performance improvement. The LU factorization appears to scale more poorly beyond 36 processors, but still maintains a consistent improvement. Figure 23 shows the execution times of the three programs for varying problem sizes.

F. Discussion

The experiments described above lead us to believe that library-level optimization is an effective way to optimize

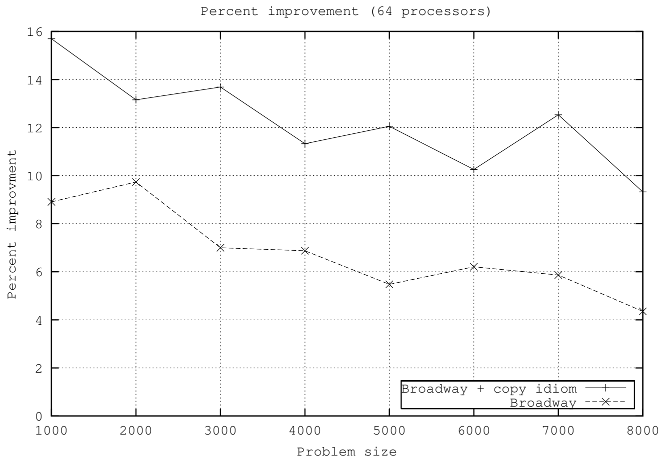


Fig. 21. Percent improvement for Lyapunov on 64 processors.

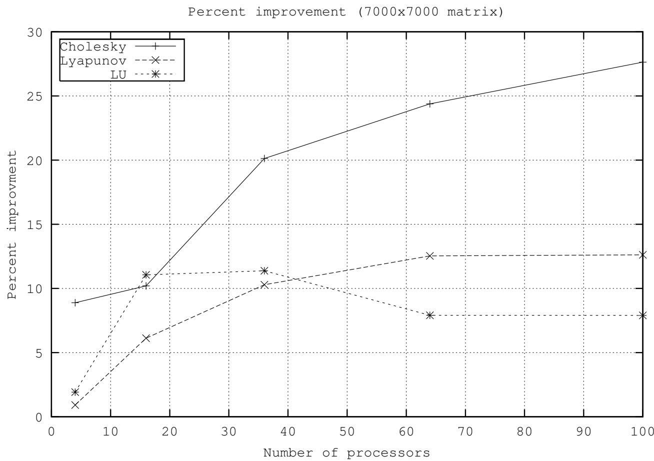


Fig. 22. Percent improvement for all three problems, across different numbers of processors.

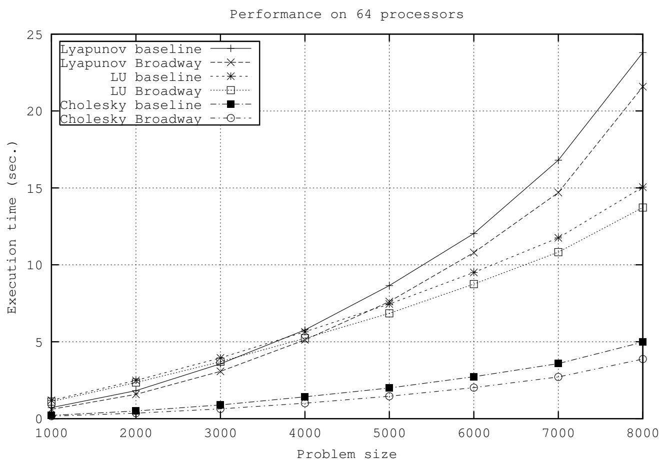


Fig. 23. Execution time for the three programs, with and without Broadway optimization.

layered scientific systems. Several observations about the experiments contribute to this conclusion:

- The technique works because it exploits domain-specific semantics that would otherwise be ignored by conventional compilers. Without a notion of matrices and data distributions, none of the optimizations we applied to PLAPACK are possible.
- The technique is effective because it crosses software layers, optimizing each layer in the context of the application and the layers above. Our design allows the compiler to shift from one domain to the next, systematically processing each layer.
- Even with limited configurability, the annotations capture useful and interesting properties of the layer abstractions. We find only a few optimizations that we could not adequately express in the language; these optimizations work on MPI routines and require an accurate model of communication.
- The annotations can be difficult to develop, but this difficulty is mitigated by two factors. First, we can develop annotations incrementally, adding new optimizations as we discover them. Second, the cost of the annotations can be amortized across a large number of applications that use the library.
- The manual application of the PLAPACK optimizations is infeasible because it is tedious and because the resulting code is incomprehensible and unmaintainable.

V. Conclusions and future work

In order to provide better optimization and error detection services, programming tools such as optimizing compilers and software checkers need improved information about program behavior. Existing systems have focused almost entirely on obtaining this information directly from application programmers. We believe that by using software libraries, programmers are already providing a wealth of domain-specific information. By capturing and codifying this information, we can significantly improve the quality of compilation without requiring any changes to existing programs or existing programming practices.

While this foundational work has produced promising results, we believe that it only scratches the surface of a large untapped source of optimization. We have identified a number of potential improvements and future directions:

- **Richer types of dataflow analysis.** Our annotation language currently supports a relatively simple class of program analysis problems. More generalized dataflow analyses would allow our compiler to construct more complex models of the library's domain.
- **Code patterns.** The current compiler only allows the annotations to replace individual library calls with other code. We can expand our range of optimizations by supporting annotations that recognize stylized patterns of library routine usage and can replace or alter the entire sequence.
- **Domain-specific traditional optimizations.** In the current compiler implementation, the traditional optimiza-

tions, such as constant propagation and dead-code elimination, work on library routines in exactly the same way that they work on primitive operations. For other traditional optimizations, however, we can formulate optimizations that work on library routines by analogy to their primitive counterparts. For example, if we tell the compiler that a particular library routine effectively creates a copy of an object, then it can apply a domain-specific version of copy propagation. Other traditional optimizations lend themselves to this technique: common subexpression elimination, management of resources, and scheduling. By exploiting existing algorithms, we can continue to keep the annotations simple.

Our work also suggests a new approach to designing software libraries that takes advantage of compiler support. In the future, such a library might consist of two distinct interfaces, one for the programmer to use and one for the compiler to target. The programmer's interface would focus on providing straightforward and intuitive access to the library's domain without exposing implementation and performance details. This high-level interface serves two purposes: first, it makes the programmer's job easier, and second, it provides domain-specific information for the compiler. The compiler interface consists of low-level library routines that serve as the compiler target and that give the compiler fine-grained control over the implementation. At this level, the routines implement the basic building blocks of the domain. The compiler analyzes the high-level interface and generates an appropriate implementation by assembling these building blocks.

Our technique is not strictly limited to libraries: it can exploit module boundaries—wherever they occur in software—to convey domain-specific information to the compiler. Our research is part of a wider trend in programming language research towards using software modularity to improve the capabilities and the performance of software engineering tools. We hope that by providing tools that are practical as well as powerful, we can help to move some of the valuable advances in compiler research into everyday programming practice.

Acknowledgments

We thank Robert van de Geijn for many useful discussions about PLAPACK and Teck Bok Tok for his recent improvements to the Broadway compiler. This work is supported by NSF grants CCR-0085792, EIA-0303609, ACI-0313263, and ACI-9984660, and by DARPA Contract #F30602-97-1-0150.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Englewood Cliffs, NJ, 1986.

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

[3] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: High performance through high-level abstraction. In *Proceedings of the International Conference on Parallel Processing*, pages 414–423, 1998.

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

[5] Arun Chauhan. Telescoping Matlab for DSP applications. Technical Report Thesis Proposal, Dept. of Computer Sciences, Rice University, June 2002.

[6] Arun Chauhan, Cheryl McCosh, and Ken Kennedy. Automatic type-driven library generation for telescoping languages. In *Proceedings of SC: High-performance Computing and Networking Conference*, November 2003.

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

[8] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it right. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[9] Timothy Scott Collins. *Efficient Matrix Computations through Hierarchical Type Specifications*. PhD thesis, The University of Texas at Austin, 1996.

[10] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[11] Luiz A. DeRose. *Compiler techniques for MATLAB programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

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

[13] Carter Edwards, Po Geng, Abani Patra, and Robert van de Geijn. Parallel matrix distributions: Have we been doing it all wrong? Technical Report CS-TR-95-39, University of Texas, Austin, 1995.

[14] Dawson R. Engler. Incorporating application semantics and control into compilation. In *USENIX Conference on Domain-Specific Languages*, pages 103–118, October 1997.

[15] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[16] Samuel Z. Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, University of Texas, Department of Computer Sciences, 2003.

[17] Samuel Z. Guyer, Emery Berger, and Calvin Lin. Detecting errors with configurable whole-program dataflow analysis. Technical Report TR 02-04, Dept. of Computer Sciences, University of Texas at Austin, February 2002.

[18] Samuel Z. Guyer, Daniel A. Jiménez, and Calvin Lin. The C-Breeze compiler infrastructure. Technical Report TR 01-43, Dept. of Computer Sciences, University of Texas at Austin, November 2001.

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

[20] Samuel Z. Guyer and Calvin Lin. Optimizing the use of high performance software libraries. In *Workshop on Languages and Compilers for Parallel Computing*, pages 221–238, August 2000.

[21] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *International Static Analysis Symposium*, pages 214–236, June 2003.

[22] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[23] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[24] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), 1991.

[25] Gary A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[26] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–181, 1986.

[27] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel code for sparse matrix applications. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–18, 1997.

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

[29] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[30] Enrique S. Quintana and Robert van de Geijn. Specialized parallel algorithms for solving linear matrix equations in control theory. *Journal of Parallel and Distributed Computing*, 61:1489–1504, 2001.

[31] Luiz De Rose and David Padua. Techniques for the translation of MATLAB programs into Fortran 90. *ACM Transactions on Programming Languages and Systems*, 21(2):286–323, March 1999.

[32] Rob Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliabiity. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[33] Steven W. K. Tjiang and John L. Hennessy. Sharlit—A tool for building optimizers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 82–93, 1992.

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

[35] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.

[36] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[37] Daniel Weise and Roger Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.

[38] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. *ACM SIGPLAN Notices*, 26(6):120–129, June 1991.