# Telescoping Languages: A System for Automatic Generation of Domain Languages

KEN KENNEDY, FELLOW, IEEE, BRADLEY BROOM, MEMBER, IEEE, ARUN CHAUHAN, ROBERT J. FOWLER, JOHN GARVIN, CHARLES KOELBEL, CHERYL MCCOSH, AND JOHN MELLOR-CRUMMEY

*Invited Paper*

*The software gap—the discrepancy between the need for new software and the aggregate capacity of the workforce to produce it—is a serious problem for scientific software. Although users appreciate the convenience (and, thus, improved productivity) of using relatively high-level scripting languages, the slow execution speeds of these languages remain a problem. Lower level languages, such as C and Fortran, provide better performance for production applications, but at the cost of tedious programming and optimization by experts. If applications written in scripting languages could be routinely compiled into highly optimized machine code, a huge productivity advantage would be possible.*

*It is not enough, however, to simply develop excellent compiler technologies for scripting languages (as a number of projects have succeeded in doing for MATLAB). In practice, scientists typically extend these languages with their own domain-centric components, such as the MATLAB signal processing toolbox. Doing so effectively defines a new domain-specific language. If we are to address efficiency problems for such extended languages, we must develop a framework for automatically generating optimizing compilers for them.*

*To accomplish this goal, we have been pursuing an innovative strategy that we call telescoping languages. Our approach calls for using a library-preprocessing phase to extensively analyze and optimize collections of libraries that define an extended language. Results of this analysis are collected into annotated libraries and used to generate a library-aware optimizer. The generated library-aware optimizer uses the knowledge gathered during preprocessing to carry out fast and effective optimization of high-level scripts. This enables script optimization to benefit from the intense analysis performed during preprocessing without repaying its price. Since library preprocessing is performed only at infrequent "language-generation" times, its cost is amortized over many compilations of individual scripts that use the library. We call this strategy "telescoping languages" because it merges knowledge of a hierarchy of extended languages into a single library-aware optimizer.*

*In this paper, we present our vision and plans for compiler frameworks based on telescoping languages and report on the preliminary research that has established the effectiveness of this approach.*

***Keywords*—Compiler optimization, component integration system, domain-specific language implementation, high-performance computing, library generation, MATLAB compiler, type analysis.**

Manuscript received March 25, 2004; revised October 15, 2004. This work was supported in part by the Department of Energy under Contracts 03891-001-99-4G, 74837-001-03 49, and 86192-001-04 49 from the Los Alamos National Laboratory through the Los Alamos Computer Science Institute (LACSI); in part by the Texas Coordinating Board's Advanced Technology Program under Grant 003604-0061-2001; in part by the National Science Foundation through the National Partnership for Advanced Computational Infrastructure under Cooperative Agreement ACI-9619020; and in part by the Computer and Information Technology Institute, Rice University, under an Innovation Grant.

K. Kennedy, R. J. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey are with the Department of Computer Science, Rice University, Houston, TX 77251-1892 USA (e-mail: ken@rice.edu; rjf@rice.edu; garvin@rice.edu; chk@rice.edu; chom@rice.edu; johnmc@rice.edu).

B. Broom is with the Department of Biostatistics and Applied Mathematics, University of Texas M. D. Anderson Cancer Center, Houston, TX 77030-4009 USA (e-mail: broom@odin.mdacc.tmc.edu).

A. Chauhan is with the Department of Computer Science, Indiana University, Bloomington, IN 47405 USA (e-mail: achauhan@indiana.edu).

Digital Object Identifier 10.1109/JPROC.2004.840447

## I. INTRODUCTION

Over the last decade, the performance of computer systems has continued to advance at a rapid pace. However, much of the progress has come at the price of increasing architectural complexity: more parallelism within and among processors, longer instruction pipelines, and deeper memory hierarchies. As a result, the fraction of the science and engineering community that can use computer systems at the limits of their capabilities has become much smaller.

Over the same period, the drive to accelerate software productivity has led many scientists and engineers to turn to high-level scripting languages and problem-solving environments, such as MATLAB [33], Mathematica [69], EllPack [37], and the S family of languages [4], [10], [11]. MathWorks, Inc., reports that there are over 500 000 licenses for

MATLAB, making it the most widely used engineering language today.

Unfortunately, this usage is primarily for small-scale experiments and prototyping, rather than production code development, because these high-level languages often do not achieve acceptable performance for complex, computation-intensive applications, especially if they entail substantial programming in the package's scripting language. For example, a linear system solver written in MATLAB that runs for several minutes on a workstation may be acceptable for testing, but is prohibitively slow if the solver is embedded in the inner loop of a simulation running for millions of steps on full-scale problems. As a result, important applications that are prototyped in such languages are rewritten and optimized by professional programmers in lower level languages, such as C and Fortran. This labor-intensive step nullifies many of the advantages of programming in high-level scripting languages. Optimizing implementations of scripting languages would eliminate the need for this step. In other words, if applications written in scripting languages could be directly compiled into highly optimized machine code, the scientific community would experience a huge productivity gain—one that would accelerate progress on the science and engineering problems that are explored through computation.

Is this goal achievable? Even today, a number of projects have produced optimizing compilers for high-level scripting languages, particularly MATLAB [15], [24], [46], [52], [54]. However, single-language solutions are not sufficient, because current practice encourages the specialization of high-level languages to specific domains through the addition of domain libraries or "toolkits." For example, MATLAB and S include many toolkits for specific areas, such as digital signal processing and biomedical computation. Even small research groups can effectively define new domain-specific languages by adding their own specialized components. If each domain library effectively defines a new language, then there will be an enormous number of specialized high-level languages that will each require an optimizing compiler. Writing so many compilers by hand is clearly beyond the capabilities of the computer science community. If we are to address the efficiency problem for such languages in a general way, we must instead develop a framework for *automatically generating* optimizing compilers for them.

To that end, we advocate a new approach, called *telescoping languages*, as the basis for automatic generation of optimizing compilers for high-level domain-specific languages [41]. The fundamental idea underlying this strategy is to construct compiler systems via an extensive preliminary analysis and optimization of collections of libraries that define the functionality of a high-level programming environment. Since the analysis and optimization phase need be done only at infrequent "language-generation" times, its cost can be amortized over many compilations of programs that use the libraries. Because the libraries define the semantics of the language, they also embody much higher level information about its domain. We have designed the telescoping languages approach to exploit such knowledge for optimization of domain language programs.

## II. Telescoping Languages: Goals, Implementation, and Applications

In designing a system to support the generation of high-level domain-specific languages, we have been driven by three key principles.

First, the compilation system must generate code that achieves *high performance*. Although prototyping systems can be useful even with low performance, our goal is to make it possible to build production applications in the generated domain-specific languages without the need for tedious recoding in a lower level programming language once an application prototype has demonstrated its usefulness. To achieve this goal, our generated code must perform well enough so that the end user is not tempted to engage in such recoding exercises.

Second, the system must support *extensibility* in that it must be possible to create new domain-specific languages on top of existing languages. In our view, most of the power of a domain-specific language derives from the operations supported by such a language. Thus, one can define new domain-specific extensions to a given language by providing a library of components that implement those extensions. The difficulty with this approach is that it conflicts with our first principle because it treats components as black boxes, which can lead to substantial inefficiencies, particularly if these components are invoked many times during the execution of an application. Viewing components as immutable objects misses opportunities for performance improvements based on the context in which the components are invoked. To address this issue, our system will need compilation technology that understands and optimizes library-based extensions.

Third, the implementation must be *responsive*—the time for the pre-execution processing of applications written in the domain-specific language must not increase nonlinearly with the size of the program. In other words, the compilation time for a script should not be astonishingly long; rather it should be linear or near-linear in the size of the script. This precludes the use of standard interprocedural optimization schemes at script compilation time because these schemes would need to process not only the user's script but also the source code of all of the components invoked, directly or indirectly, from within that script.

To construct a system consistent with these principles, we must exploit powerful interprocedural compilation strategies without dramatically increasing script compilation times. To do this, we propose moving the costs of optimizing components to a new "language-generation" step, which will preprocess the library to produce a fast script optimizer. Fig. 1 provides a high-level overview of this strategy, which performs compilation in two phases.

- The *library analysis and preparation phase* depicted in the upper half of the figure is used to optimize li-

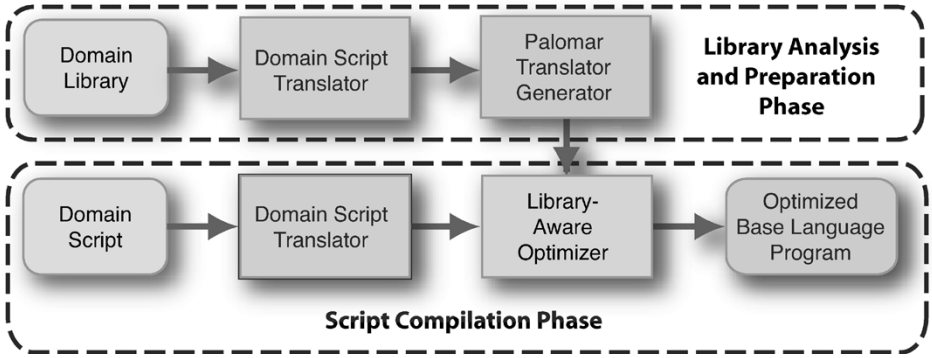



**Fig. 1.** Telescoping languages.

brary packages speculatively for use in high-performance scripts. Its principal component, the *Palomar translator generator* takes a library, along with programmer-specified relationships among the library entries, and generates a library-aware optimizer for use in script translation. Because this step is performed only when the library is defined or redefined, it can employ very expensive analyses and optimizations, incorporating the results into a library-aware optimizer that will be reused on many scripts.

- The *script compilation phase* depicted in the lower half of the diagram is used to compile and optimize scripts that invoke entries in the domain-specific libraries. The efficiency of the code generated in this phase depends on two components: the *script translator*, which produces an efficient version of the script in the system's "base language" and the *library-aware optimizer*, which incorporates numerous optimizing transformations generated by the library analysis and preparation phase.

In this scheme the domain-specific language is defined by two components: the scripting language, as implemented by the script translator, and the domain-specific library, which defines the component operations that can be invoked from the script. Because we expect the scripting language to change less frequently, the library analysis and preparation phase can be thought of as the language-generation step.

We note that these two compilation phases roughly correspond to two classes of programmers. Specialists will write the libraries that are input to the library analysis and preparation phase while ordinary users will feed their high-level applications to the script compilation phase. However, as code written in scripts matures, it can be fed back into the library analysis and preparation phase to optimize performance further. In this way, the compilation strategy can "telescope" hierarchies of libraries, thus constructing new optimized languages on top of the existing base.

To illustrate how telescoping languages could be used to create a high-performance, domain-specific programming environment, we present a simple scenario.[1] Initially, a user creates a new optimization method suitable for inversion problems such as seismic imaging. He develops and tests his method on small data sets by instantiating a new, domain-specific toolbox in a scripting language such as MATLAB. The seismic imaging application is an ordinary script calling the toolbox.

Once the method works for small data sizes, it must be tested on more realistic problems. If the performance is still disappointing using a conventional optimizing MATLAB compiler, the best current option is to turn over the application (including the toolbox) to a group of professional programmers who painstakingly reimplement it in Fortran or C. It is precisely this step that our scheme aims to make unnecessary.

Alternatively, using telescoping languages framework, the user would present the library (with annotations, as explained in Section IV) to Palomar. As appropriate, Palomar would synthesize specialized library routines tailored to expected call-site contexts and construct new library operations by decomposing and combining procedures. In addition, Palomar would generate a companion library-aware optimizer. This optimizer would then be used to compile unmodified seismic imaging applications into optimized programs that use the enhanced library. We expect a Palomar-generated optimizer to produce faster code by exploiting algebraic identities provided to it as library annotations. In addition, the optimizer will generate code more quickly than a whole-program compiler, thus shortening the program development cycle.

### A. Sample Applications of Telescoping Languages

Telescoping languages is a surprisingly flexible concept that can be used to unify a variety of emerging ideas on implementation of high-level languages and construction of powerful programming systems. To give the reader a taste of the possibilities, we now present a few examples of the application of telescoping language ideas to real problems.

- *MATLAB compilation:* MATLAB itself can be viewed as a scripting language in which the library of matrix operations from LAPACK and other sources defines the domain. Telescoping languages provides a mechanism for substituting calls to the most specialized entries to those libraries based on context. In addition, by avoiding extensive inlining, it reduces script compilation times. This application is described in Section III-B.

[1]This example is based loosely on the experiences of our colleague W. Symes of Computational and Applied Mathematics. Because he did not have access to telescoping languages, Symes had to abandon MATLAB for his seismic imaging application, primarily due to poor computational performance and lack of support for out-of-core arrays.

- *MATLAB for digital signal processing:* MATLAB is widely used for prototyping signal processing applications. Because successful prototypes are almost always rewritten in C to achieve high performance or compact memory usage on embedded processors, signal processing is also an ideal domain for telescoping languages. Our preliminary study, described in Sections III-A and V-A, has determined that reuse of standard libraries is common practice in these applications. In addition, the research has discovered a number of high-level transformations whose usefulness extends beyond the domain itself, including procedure vectorization and procedure strength reduction [13], which are described in Section IV-B.
- *Medical statistics:* Clinical trial design is a computationally intensive statistical calculation that must be repeated for each new trial. Currently, biostatisticians develop models of new experimental trial designs and new statistical methodologies in the language S, because it includes the high-level, complex statistical operations that greatly facilitate the rapid development of new models. Unfortunately, the performance of the current commercial and open-source S implementations is too slow for direct use in the simulations required, so that the applications must be laboriously recoded in C or Fortran after the prototyping stage. In a preliminary study, discussed in Section III-A, we have demonstrated that compilation techniques, combined with specialized high-level operations, can lead to hundredfold performance improvements in S, which would make the recoding step unnecessary. This application is discussed further in Section V-C.
- *Library prototyping:* Several library developers have confessed to us that they prototype their libraries in MATLAB or some other scripting language and then translate the prototype to several variants in C or Fortran to achieve higher performance. The latter process typically involves specializing the prototype by hand to different expected calling sequences (e.g., real versus complex, dense versus sparse, symmetric versus nonsymmetric matrices). In Section V-B, we show how telescoping languages can eliminate the need for this step by using our MATLAB compilation strategy and its type analysis system [44]. The result is a powerful tool that can be applied to libraries in many other domains.
- *Component integration systems for scientific computing:* Component integration systems, though acknowledged as a promising technology for improving software productivity, incur performance penalties that make the integrated applications inefficient for scientific use. The principal problem is that, for flexibility, components are constructed and compiled well in advance of the applications that incorporate them so optimization of components to context is precluded. In addition, the linking process for components is dynamic so that component-crossing invocations are more expensive than function calls in standard languages. Most of these problems can be overcome if the collection of relevant components is known in advance and preprocessed by a library analysis and preparation phase like the one in telescoping languages. Thus, telescoping languages could provide the basis for an efficient component integration system. This topic is explored in more detail in Section V-E.

### B. Implementation Overview

In the next few sections of this paper, we discuss our approach to implementation of the telescoping languages framework. To repeat, implementation is driven by the desire to achieve three general goals: the applications generated from scripts must have excellent performance, the scripting language must be extensible by adding new primitive operations into the library, and the script compile time should be reasonable—that is, script compile time should be roughly proportional to the length of the script. We divide our discussion of implementation into two main topics.

- *Scripting language translation*. Section III describes the strategies we are using to translate scripting languages into efficient code in a base language such as C or Fortran, so that source-to-source transformation strategies developed for such languages can be easily applied. Script language translation differs from script compilation as depicted in the lower half of Fig. 1 in that the latter also includes library-aware optimizations. In other words, scripting language translation is the first step in script compilation. In addition, if domain libraries may be written in scripting languages, scripting language translation must be part of the library analysis and preparation phase as well.[2]
- *Library analysis and preparation*. Section IV discusses the strategies for generating library-aware optimizers for scripting languages. This basically incorporates all of the facilities in the Palomar translator generator depicted in the upper half of Fig. 1. The key phases of the Palomar system include: 1) generation of type jump functions that can be used to propagate the generalized types supported in the domain language; 2) construction of a macro substitution system that replaces sequences of component invocations occurring in the script with other sequences that are more efficient in the application context; and 3) generation of components specialized to likely application contexts and a phase that substitutes these for more general, and less efficient, versions in any application context where it is legal to do so.

In the remainder of the paper, we describe the ambitious program of research that we have undertaken to validate and apply the telescoping languages concepts. We begin with a description of the approaches we plan to use for both script translation and library precompilation, completing this discussion with a description of the current state of our

[2]If one eliminates the script translation step, then the telescoping framework still works to optimize library calls in programs written in the base (C and Fortran) languages: it then becomes a powerful tool for optimizing programs, written in those languages, that make extensive use of libraries.

implementation. In Section V, we conclude the technical material in the paper with a more detailed description of the applications that are driving our research and we present some preliminary results with these applications.

## III. Scripting Language Translation

We now turn to a discussion of the technologies we are using to translate scripts to a base language such as C or Fortran. As shown in Fig. 1, the script translation step can be used either in the script compilation phase or the library analysis and preparation phase. Because the translation of libraries must be done without the knowledge of the calling programs, special requirements are imposed on script translation in the library analysis and preparation phase; these are discussed in more detail below.

Our research on script translation focuses on two languages that are widely used for prototyping applications in the science and engineering community: MATLAB and S.

- MATLAB is a dynamic, array-based language that is widely used in science and engineering. MATLAB is "untyped" in the sense that a variable may be used at different times for different data structures. The principal data structures available in the language are scalars and arrays of various dimensions. Arrays are dynamically allocated and can be expanded during the course of a program execution, often by simply storing into a row or column that has not yet been allocated. In addition, MATLAB supports both dense and sparse representation of arrays.
- The S language (its commercial implementation is called S-PLUS, while the open-source implementation is referred to as "R") is widely used for statistical calculations, particularly in biology and medicine. It is similar in flavor to MATLAB, in that it is interpreted and untyped and its execution involves dynamic allocation of large data structures, particularly arrays.

### A. Preliminary Studies

To determine the best strategies for scripting language translation, we conducted preliminary studies of MATLAB and S programs to determine the sources of performance problems in those languages. These studies, which focused on the kinds of optimizations that a translator should include, are described in the following paragraphs.

*MATLAB Optimization:* As a prelude to our work on the use of MATLAB in signal processing (see Section V-A), we studied the sources of inefficiency in MATLAB. The current commercial MATLAB implementation by MathWorks is interpretive, so that the operations on the major data structures in the languages—scalars, vectors, and arrays of various types and structures—can be dynamically selected during execution. Although the MATLAB 6.5 implementation includes a just-in-time (JIT) compiler, which produces good code locally, it does not perform global optimizations like type analysis over a whole function. MathWorks also offers MCC, a compiler that translates MATLAB to C. In our experiments, however, MCC does not markedly improve performance; instead, it provides a compatibility mechanism that allows merging of MATLAB and C programs. With MCC, the fundamental MATLAB operations are still embedded in the MATLAB runtime library routines, which are callable from C and perform the functions of the interpreter before carrying out an operation.

Clearly, a true compiler—one that determined the datatype, size, and shape of every variable in a MATLAB function and then generated C or Fortran code—should provide substantive performance benefits over the current interpreter, even with local JIT compilation. There are two things that must be done to produce such a compiler. First, the compiler must be able to determine, through a sophisticated type analysis such as the one described in Section III-C, a general type for every variable at every access point in the program being compiled. Fig. 2 shows the impact of inferring the correct types on the performance of a user-level DSP library procedure, called `jakes`. For each platform, the bars on the left show the running time of jakes on the MATLAB 6.5 implementation, which uses dynamic type testing, while the bars on the left show the running time of the Fortran version of `jakes` in which the types intended by the user are explicit and primitive operations specialized for those types are used. The figure clearly demonstrates that specializing on the right type configuration is very important to achieving high performance in a translated application.

Second, the compiler must take steps to make it possible to replace dynamic data structures with statically allocated structures or at least dynamic structures that are less frequently reallocated. In MATLAB, it is possible to increase the dimensions of an array by simply storing beyond the allocated bounds of that array. Our study showed that MATLAB programmers frequently do this in loops. However, if the aggregate size after all such allocations can be determined in advance, a single allocation at the beginning of a routine will suffice. Section III-C describes the strategy that our prototype MATLAB translator uses to accomplish this. Static allocation of this kind is well-understood strategy, described in MATLAB tutorials, for improving MATLAB performance by hand. Our contribution, driven by necessity (otherwise translation to C or Fortran would not work), is to automate the strategy in a translator.

In addition to these two essential functions, there are a number of transformations that are extremely effective for improving performance of MATLAB programs. For example, because array operations in MATLAB are implemented using calls to highly optimized array libraries, while element-wise operations on the same structures in loops are interpreted, it pays to convert loops to array operations. This optimization, known in the literature as "vectorization," can produce integer factor improvements in the running time of MATLAB programs.

*S Optimization:* In collaboration with biostatistical researchers in the M. D. Anderson Cancer Center, Houston, TX, we have conducted a study of a number of applications written in the S language. These applications include many that employ calls to standard toolbox routines written by the

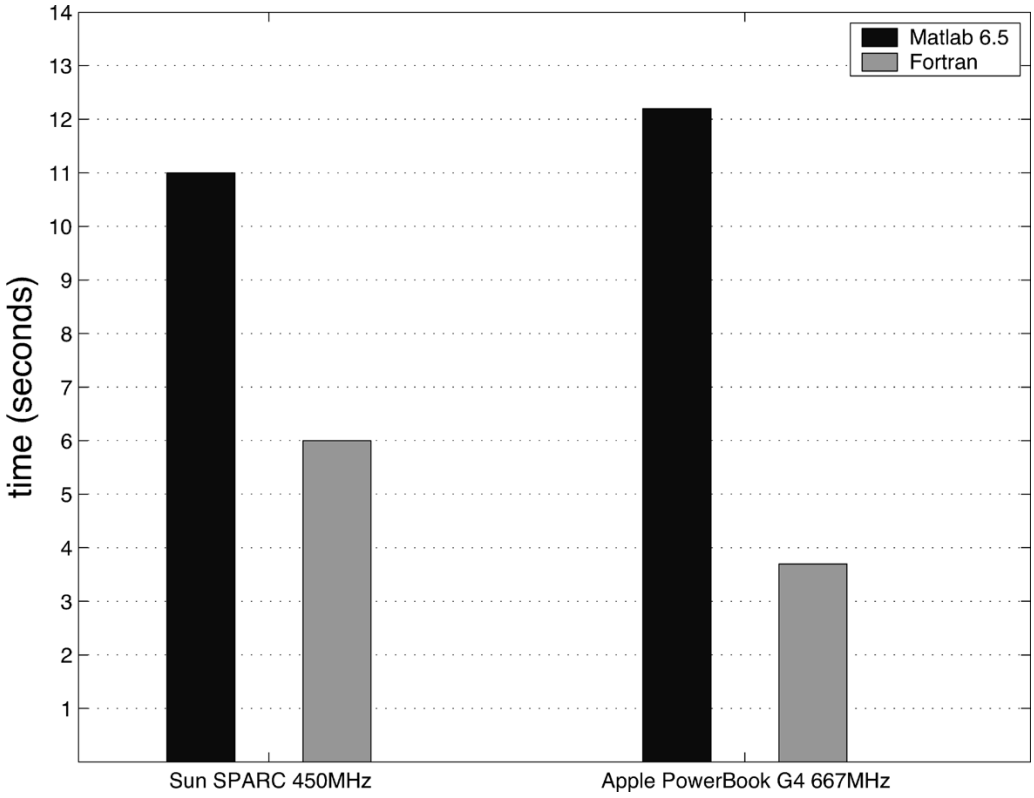



**Fig. 2.** Importance of type-based specialization.

M. D. Anderson researchers. The S programs we examined also included the use of many standard programming idioms.

Our study identified a collection of transformations and compilation strategies that improved various S programs by factors ranging from 10 to 143. The most important of these were: 1) avoidance and elimination of allocation of temporary arrays; 2) conversion of loops to array operations (e.g., vectorization); 3) hoisting array expressions out of loops; 4) translation to primitive types and native operations in C; and 5) folding temporary arrays into usage points. Using domain-specific information about the structure of a matrix, we were able to obtain an overall speedup of 23 000 for one program.

Although all but the domain-specific optimizations could be employed in a whole-program optimizing compiler, the compile time of such a strategy would be prohibitive. Scripting language users like being able to modify and run a script on a tight cycle and they would typically want to modify the program even after it goes into the production stage. This is the reason that we believe that the scripting community will greatly benefit from the telescoping languages strategy, which would move much of the optimization cost to the library analysis and preparation phase.

### *B. Generating Base Language Programs*

Although there is a significant body of work on compilation of MATLAB and S (especially MATLAB—see Section VI), to support telescoping languages, we need to avoid the common practice of extensive procedure inlining lest we make compile time too long to be practical. Thus, an important part of our work has been the development of interprocedural analysis and optimization strategies that leave procedures intact and only minimally expand the code size.

Our compiler strategy for these languages performs the following steps. First, it carries out a form of type analysis (discussed below) that reduces the number of types that must be considered for each variable at each point in the program. In most cases the number of such types can be reduced to one. Then, it uses the reduced set of types to generate a C or Fortran program that invokes an intrinsic operator or library routine implementing each operation or function invocation in the script. When the set of reduced types is greater than one, it uses runtime type checking to select the right operation.

In the case of the subset of MATLAB for linear algebra, the generated C or Fortran program invokes the appropriate routine from LAPACK once the types of the operations are known.

Our current work on S translation has focused on the closely related language "R," which differs from S in several significant ways. In particular, S is dynamically scoped, while R is lexically scoped. The principal reason for using R is that we can base our program generation on the R open-source runtime library. Our initial compiler, called RCC, simply converts the R code to C by using the components from the R runtime library to implement the intrinsic operations in the language. This approach does not produce the best possible performance because, as in the case of the MathWorks compiler for MATLAB, the component routines include parts of the interpreter to determine input and output types for the operation. We plan to improve this by using type analysis as in MATLAB. A more complete discus-

sion of the status and performance of RCC is contained in Section IV-D.

### C. Type Analysis

A key technology for compiling any untyped scripting language is type analysis, where the term "type" is used in the most general sense to indicate not only data types but also more abstract properties such as array rank, size, and matrix type (e.g., symmetric versus nonsymmetric). Type analysis, therefore, makes it possible to determine the correct declaration type and size of structures within the language so they can be statically allocated in a base languages such as C or Fortran, as well as providing information useful for specialization even in typed languages.

In our case, this problem is complicated by three factors. First, the dynamic nature of scripting languages such as MATLAB makes it difficult to statically capture the behavior of the functions. Second, because library routines might be specified in the scripting language, the type analysis must be carried out on functions in which the calling context is not yet known. Thus, the standard strategy of function inlining is not an option in our approach. Third, accurate type analysis for MATLAB requires the use of both forward and backward propagation of type information, which suggests a constraint-based approach.

In order to get the most accurate type information for specialization, type analysis must be able to infer which type configurations over the function variables could validly occur, not just types for each variable in isolation. To capture this information, we construct *type jump functions*, which can be quickly evaluated from the types of the inputs to determine variable types everywhere in the function. We represent a type jump function as a table where every entry represents a valid configuration of types over the variables in the routine. Each valid type configuration over the inputs results in a separate generated variant.

To address these issues, we have developed a new polynomial-time algorithm for constructing type jump functions for a MATLAB routine [44]. The long-term implication of this work is that it is possible to generate code for a given MATLAB routine during the library analysis and preparation phase and to use this information at script-compilation time to determine quickly and accurately the effects of library calls on its inputs and outputs.

Our type inference strategy first examines each statement in isolation and then combines the information from the statement over the whole function to determine the valid type configurations. Type inference is performed over a normalized version of the function in which each statement only involves a single operation or function call. Also, because variables can change types in MATLAB, we use static single assignment form so that each use refers to a single definition point and, therefore, a single type. The statement constraints are formed by looking up the type jump function for that statement's operation or function call. The statement constraints are written as a sequence of clauses, where each clause gives a possible assignment to the variable types.

The statement constraints are then combined over the entire function using a graph representation. The graph has $n$ levels, where each level corresponds to a different statement constraint. Each node in a level corresponds to a clause, and there is an edge from one node to another if the types given by the clauses do not contradict each other.

The problem of finding all valid type assignments then becomes one of finding $n$-cliques over the graph, where a clique is a complete subgraph. The intuition behind this is that we want to find type assignments that are valid over the entire function. That is, for each statement, they must satisfy a clause in the corresponding statement constraint. Thus, we want to find sets of clauses, one from each statement constraint, such that the clauses in each set do not contradict one another. On the graph, this exactly corresponds to finding $n$-cliques. Each resulting clique is solved to produce a type configuration which is included in the type-jump-function table.

While finding cliques in general is NP-complete, we are able to use the structure of the problem to prove that we are working on a subset of the clique-finding problem that is polynomial under practical conditions.

Array sizes present a special problem because they may depend on values or the outcome of control statements that are not determinable at library analysis time. The static type inference algorithm is able to describe the sizes of the arrays in terms of the size of the inputs and other variables defined within the program using metavariables to represent the statically undeterminable sizes (i.e., the sizes of the inputs). Unfortunately, this is not enough to capture the behavior of the library routines in some cases. For example, arrays can grow within MATLAB routines by assigning to a subscript outside of the array's initial bounds. This array growth in many cases requires costly reallocation of the array. To address this problem, we employ a code transformation technique called *slice hoisting* [14]. Slice hoisting moves the computation of these array sizes to just after function entry, so that allocation of the arrays may occur as early as possible.

Finally, type analysis can be substantively assisted by having types of the input parameters externally specified through annotations by library writers. While static type inference will account for every possible configuration of types over the variables, some of the inferred type configurations may involve input types that should never occur in practice. Annotations help the compiler generate only the variants that are used in practice. They also reduce the complexity of type inference, and in the case of size inference, can provide exact sizes that are undeterminable through static analysis.

Fig. 3 demonstrates how the three different ways to disambiguate types combine to precisely infer the sizes of all the variables in five different DSP library routines. Static type inference is able to determine a majority of the array sizes in terms of exact values or the sizes of the other variables. Slice hoisting is used to determine those variables that depend on the outcome of complex control flow or values resulting from complex computations. Finally, annotations were needed to determine exact sizes for some of the inputs.

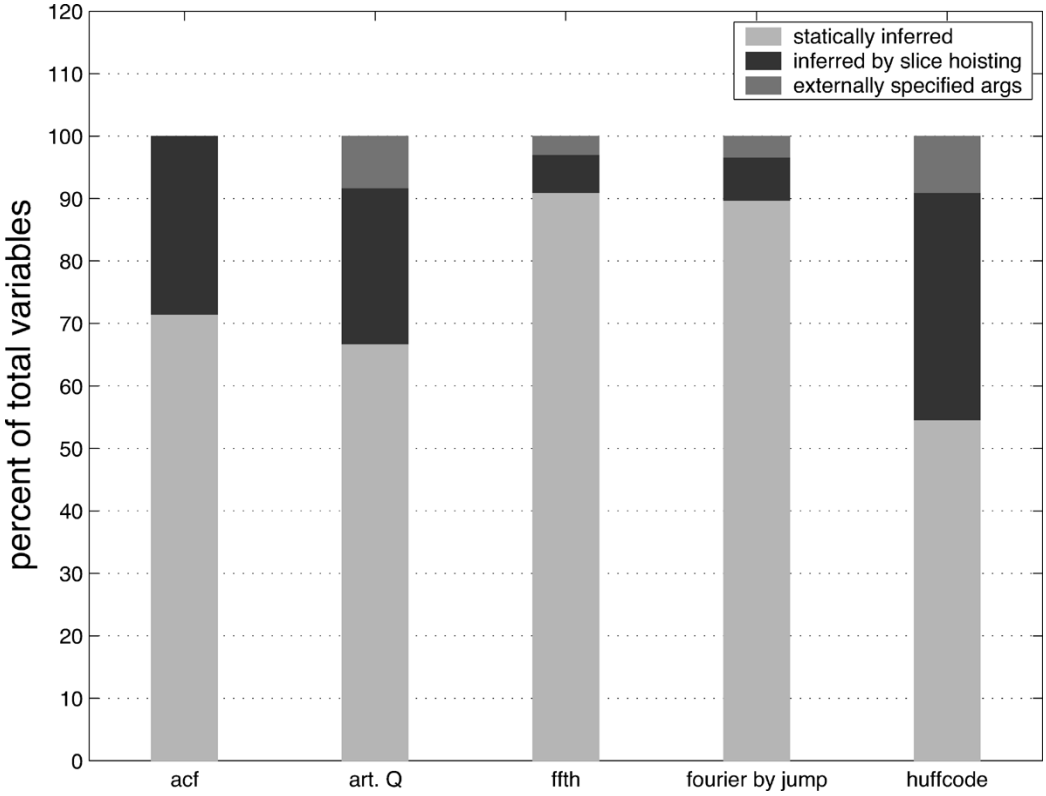



**Fig. 3.** Accuracy of type analysis.

Once the actual types of inputs are known at script-compile time (or during the generation of specialized variants during library analysis and preparation), the type jump functions are evaluated to make it possible to generate C or Fortran output code for the MATLAB routine. The operations and function calls are replaced with calls to the appropriate optimized variants. In our work, this code directly invokes the LAPACK routines for the primitive operations on the arrays, particularly the kernel functions that compose the BLAS.

In Section V, we present a special application of the MATLAB compilation technology to maintenance of scientific libraries, along with statistics establishing its effectiveness in improving performance.

## IV. Library Analysis and Preparation in Palomar

To create an optimizing compiler for a domain language, Palomar will be applied in the library analysis and preparation phase to generate an optimized runtime library and a library-aware optimizer for the domain language. Fig. 4 outlines the steps of this process. Palomar will take as inputs a set of procedures expressed in the base language that define domain language primitives and toolbox components (translated where necessary from the domain language). After extensive analysis, Palomar will create an enhanced library, which will include, in addition to the domain primitives and simple translations of procedures in the domain toolbox, versions of routines tailored to exploit common contexts in which they may be invoked. Palomar will also generate a library-aware base-language optimizer that will have knowledge of the semantics of both the original and enhanced library primitives.

To more completely understand what must be done by Palomar in this phase, we must consider the functions to be performed by the generated optimizer. These functions fall into three categories.

First, the optimizer must propagate generalized types (e.g., matrix base type, size, shape, density, and other user-defined properties) throughout an application script. To do this without making compile times excessively long, a violation of one of our goals, the optimizer must have precomputed *return type jump functions*, specialized versions of the type jump functions discussed in Section III-C, for each component in the library. For a given library component, the return type jump function quickly computes the types of output parameters from the types of the input parameters. This requirement means that the optimizer generator must both determine the relevant types that might be attached to component parameters and compute return type jump functions for the component on input parameters ranging over those types. This capability is discussed in Section IV-A.

Second, the optimizer must implement a substitution phase that replaces sequences of component invocations occurring in the script with other sequences that are more efficient in the application context. These substitutions are derived from annotations by the library designer that are part of the material presented to Palomar. In a sense, these annotations define an algebra of relations among the components along with information about cost that can be used as a basis for optimization decisions. For example, if a library

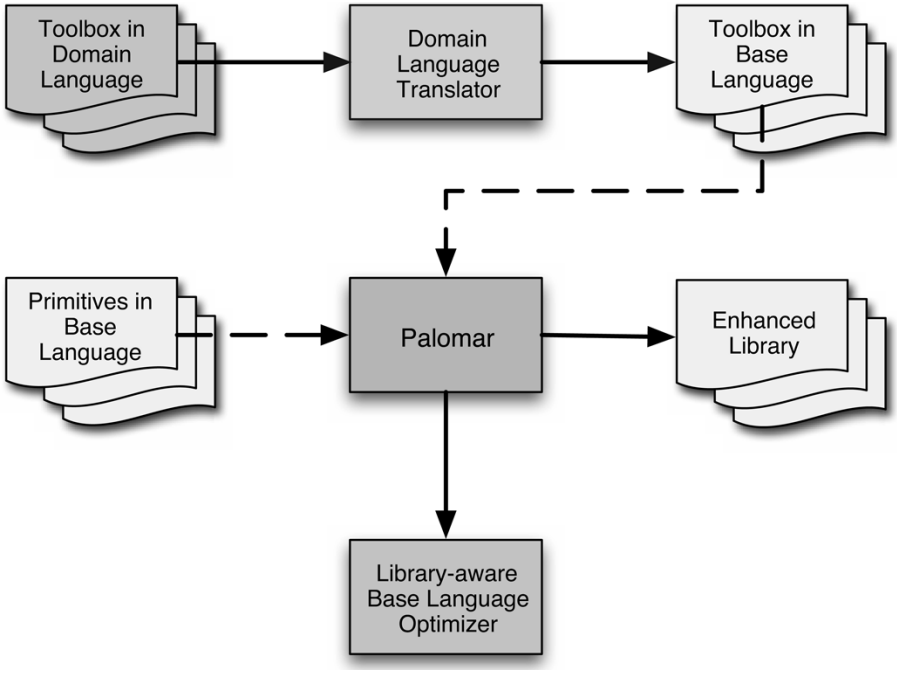



**Fig. 4.** Library analysis and preparation with Palomar.

implements a stack data structure, the designer knows that "push" and "pop" are in some sense inverses: the script compiler should be able to replace a push followed by a pop with a simple assignment. To generate this global substitution system, Palomar must be able to accept annotations and construct substitution transformations from them. Strategies by which to accomplish this are discussed in Section IV-B.

Third, once the calling context is known for each component invocation, the optimizer must substitute the specialized version of a component that is best suited to that context. In other words, if there are multiple versions of a library component, the optimizer should select the most efficient version consistent with the types that are actually presented when the application script is known. To support this process, the optimizer generator must produce a database of specialized versions of each library component, while limiting the number of such variants to a manageable size. The mechanisms for both specialization and selection are the subject of Section IV-C.

In the next few subsections, we discuss how elements of Palomar will support these script compilation functions. Section IV-D finishes the treatment by describing the current implementation status of the project and the software infrastructure that will serve as the foundation of Palomar.

### *A. Abstract Properties and Their Propagation*

If Palomar is to generate domain-language optimizers that can accurately predict the runtime types that apply at any point in the user's application, it must address two critical challenges. First, it must *identify* the types and properties that are to be propagated across the program and, second, it must produce type jump functions that propagate these types across calls to the underlying domain library used as the semantic basis for the language.

While the optimizers generated by Palomar should be able to propagate information on standard types (e.g., real or complex, scalar versus array), higher level information will often be the key to effective optimization. For instance, knowing whether a matrix is sparse, dense, symmetric, or triangular at some point in the program is an example of a key property that will not be obvious from low-level analysis.

The problem with abstract types such as these is that they can be large in number, which can cause an explosion in the number of specialized routines that need to be collected in the enhanced domain library. Therefore, Palomar will need to incorporate strategies for identifying the important types for use in the generated optimizer. There are three ways this may be done.

- Palomar can reason backward from particularly profitable optimization opportunities to determine what parameter types will make these optimizations possible. This will commonly involve reasoning about the value ranges of scalar variables or the sizes of arrays. For example, if an operation steps through an array using an unknown stride parameter, it may be worthwhile to have a specialized version for the case where the stride is greater than zero, because the operation can be unconditionally vectorized in this case.
- Palomar can examine sample input files, typically provided by library designers, to look for particularly interesting properties that might be exploited to produce better code. For example, it might be possible to determine that two array parameters to a given library routine are frequently just shifted versions of the same array, which could lead to better memory utilization within the body of a specialized version of that routine.
- Finally, important properties can be specified as annotations by the library designers. We expect that this

category will be the most useful because it provides a natural mechanism for the library designer to specify types that lead to the highest payoffs and the transformations and specializations that can be carried out based on those types. In this category, one might find the most abstract concepts such as sparsity and storage layout.

Our plan for the Palomar implementation is to support all three approaches with the same general mechanism. Specifically, each component of the domain library will be annotated with a collection of tuples, in which each tuple contains a set of parameter types that can be passed to that routine in some legal invocation for which a specialized version would be profitable. That is, each routine would have a table of input types for which specialization is desirable. Our preliminary implementation relies exclusively on library developer specification to define these tuples, but the mechanism should be completely general.

For this to work with abstract types specified by the library designer, the library itself would need to be augmented with the specification of the domain of types for that library. Our current MATLAB implementation supports types that include scalars and arrays of various base types, where arrays have additional specifications of rank, size in each dimension, and "pattern" (e.g., dense, triangular, symmetric, etc.). However, the library designer must be free to add new types as needed.

Finally, each callable library component must have a *return type jump function*, which computes the types for the output parameters as a function of the types for the input parameters. These functions make it possible to determine the side effects of procedure invocation without the overhead of a full analysis [9]. Thus, by precomputing return type jump functions during the library analysis and preparation phase, we can dramatically speed up the propagation of types when the application is presented during the script compilation phase. In Palomar we construct return type jump functions using the constraint-based type analysis strategy described in Section III-C.

### B. Constructing a High-Level Transformation System

To optimize code from a user application after script translation to a base language, the library-aware base-language optimizer generated by Palomar will transform it by rewriting fragments that match a database of transformation rules. This database, which is also produced by Palomar, will include rules synthesized automatically during analysis of the domain library as well as high-level transformation rules written by the language designer. In essence, Palomar will generate a transformation phase that can be thought of as a sophisticated macro substitution process.

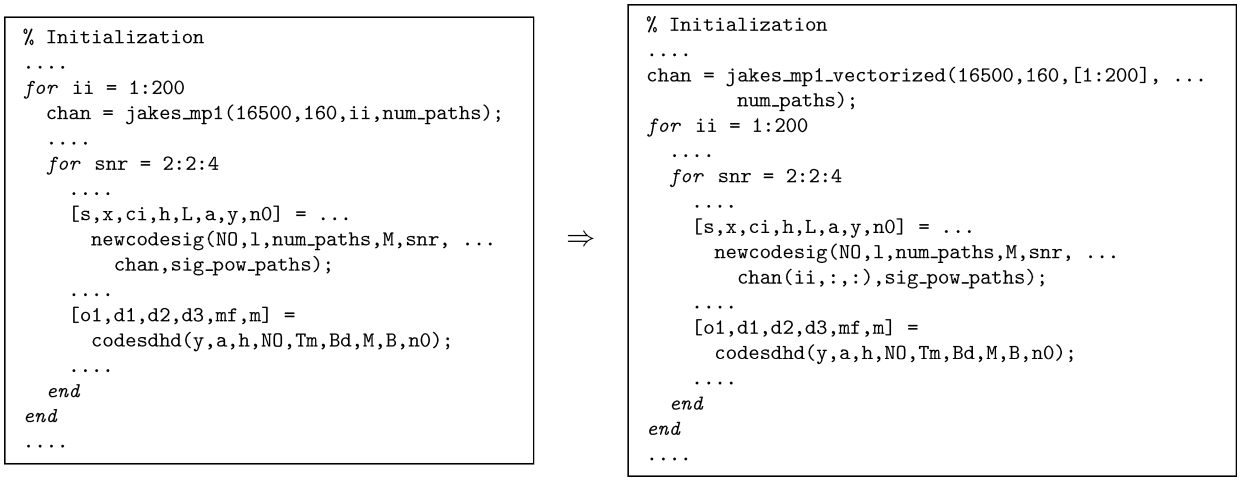
We plan to generate optimizers that include a transformation pass constructed both from library analysis and from designer specifications. In other words, some transformations will be automatically generated, while others will be completely specified by the library designer. In the first case, identification of useful transformations, along with the conditions under which they will be profitable, will lead Palomar to break up library components into more fundamental parts that can be recombined into efficient sequences by these transformations. In the second case, the library designer, based on a deep understanding of the underlying problem domain, should be able to specify an algebra relating sequences of library calls to other sequences that are equivalent in meaning within a given calling context. When coupled with an understanding of the relative costs of these sequences, the transformation pass can make substitutions that will improve performance, once the types of parameters and calling contexts are known. In what follows, we describe the mechanisms for each of these transformation strategies.

*Procedure Transformations:* The first high-level transformation strategy entails identification of a collection of common transformation patterns along with the conditions under which it is profitable to employ them. Our MATLAB and S experiments suggest several possibilities, primarily based on the observation that many procedures and functions are invoked within loops.

Our original study of MATLAB programs, described in Section III, established the importance of vectorization [12], which entails the conversion of loops into whole-array or vector operations that can be carried out by MATLAB's efficient array-handling primitives. At the procedure level, the analogous operation is *procedure vectorization* in which a procedure or function that might be applied to the elements of a vector or array is converted to one that can be applied to the entire vector or array. The transformation that makes this possible is the movement of a loop over the elements of the key array into the procedure or function. Fig. 5 shows the result of vectorization applied to a routine within a signal processing application from the Department of Electrical and Computer Engineering, Rice University, Houston, TX. Before the transformation is applied, the variable `chan` is a two-dimensional array that is computed each time through the loop on `ii`. After the transformation, an extra dimension of size 200 is added to `chan` which is now computed outside the loop on `ii`. The vectorized version of `jakes_mp1` is produced by moving a copy of the loop on `ii` into the function, which in turn permits vectorization of code within that routine.

In addition to procedure vectorization, we have designed and implemented *procedure strength reduction* [13], which can be applied to loop-enclosed procedures for which only a single parameter depends on the loop induction variable. Such procedures could be divided into two components—a loop invariant initialization section and a loop-varying section—enabling the initialization section to be moved outside the enclosing loop. Fig. 6 shows the result of applying this optimization to the same example that was used in Fig. 5. In this case, however, strength reduction is applied to three procedures: `jakes_mp1` with respect to the loop on induction variable `ii`, `newcodesig` with respect to the loops on both `ii` and `snr`, and `codesdhd` with respect to the loop on `snr`.

Our preliminary studies of MATLAB applications in signal processing have shown that procedure strength reduction alone can lead to substantive improvements in running

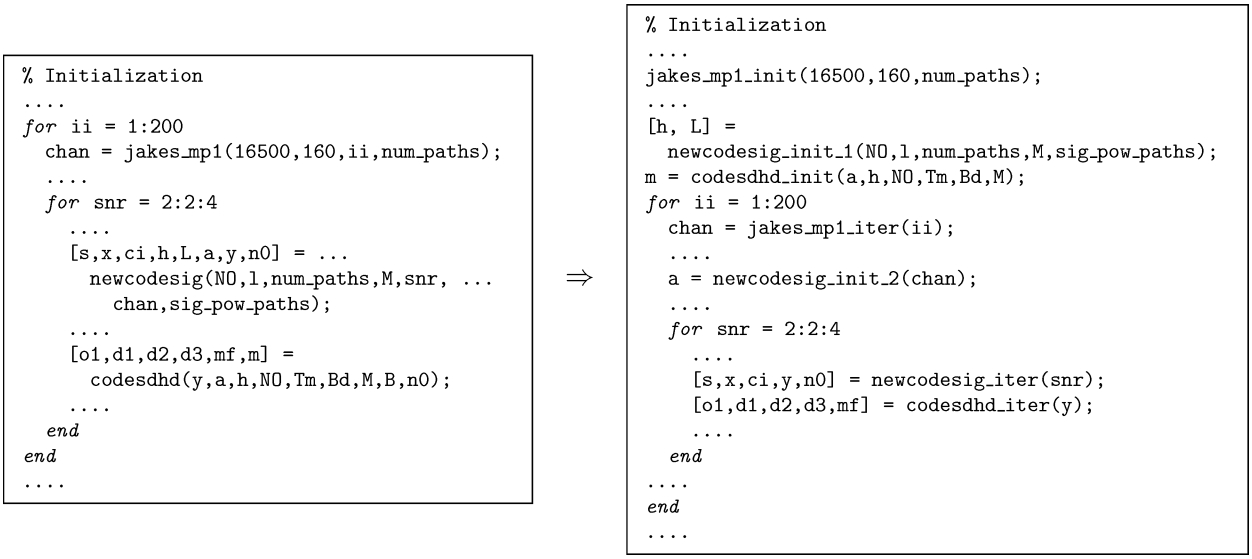


```
% Initialization
....
for ii = 1:200
  chan = jakes_mp1(16500,160,ii,num_paths);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = ...
      newcodesig(NO,l,num_paths,M,snr, ...
        chan,sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] =
      codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

⇒

```
% Initialization
....
chan = jakes_mp1_vectorized(16500,160,[1:200], ...
        num_paths);
for ii = 1:200
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = ...
      newcodesig(NO,l,num_paths,M,snr, ...
        chan(ii,:,:),sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] =
      codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

**Fig. 5.** Applying procedure vectorization to `jakes_mp1` called in `ctss`.



```
% Initialization
....
for ii = 1:200
  chan = jakes_mp1(16500,160,ii,num_paths);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,h,L,a,y,n0] = ...
      newcodesig(NO,l,num_paths,M,snr, ...
        chan,sig_pow_paths);
    ....
    [o1,d1,d2,d3,mf,m] =
      codesdhd(y,a,h,NO,Tm,Bd,M,B,n0);
    ....
  end
end
....
```

⇒

```
% Initialization
....
jakes_mp1_init(16500,160,num_paths);
....
[h, L] =
  newcodesig_init_1(NO,l,num_paths,M,sig_pow_paths);
m = codesdhd_init(a,h,NO,Tm,Bd,M);
for ii = 1:200
  chan = jakes_mp1_iter(ii);
  ....
  a = newcodesig_init_2(chan);
  ....
  for snr = 2:2:4
    ....
    [s,x,ci,y,n0] = newcodesig_iter(snr);
    [o1,d1,d2,d3,mf] = codesdhd_iter(y);
    ....
  end
....
end
....
```

**Fig. 6.** Applying procedure strength reduction to procedures called in `ctss`.

times. Fig. 7 shows the improvements resulting from procedure reduction in strength applied to three MATLAB routines and two full applications. On these codes, both the original and optimized version were run on the MATLAB interpreter.

These examples represent only some of the transformations that should be explored to fully generate optimized specializations of single procedure calls and common invocation sequences.

Both of the transformations described here involve generating one or more new routines from existing library procedures. As we shall see in Section IV-C, we must avoid generating too many specialized procedure variants if we wish to maintain performance of the script compiler. Therefore, we will use specifications by the library designer to indicate when to perform this generation. For example, the designer might indicate that a particular procedure will often benefit from vectorization or that it is likely to be called in a loop with the loop index passed in the third parameter position. Based on these "profitability specifications," Palomar will generate the appropriate specialized variants, such as `jakes_mp1_vectorized` or `newcodesig_iter` automatically and include them in the library of specialized variants.

*User-Specified Transformations:* User-specified high-level transformation rules are critical for achieving top performance with domain-specific languages because automatic analysis cannot discover all useful high-level transformations. Many useful transformations cannot be found because the enabling preconditions are obscured. For example, when using a library that implements access to an "out-of-core" array, it would be useful to replace a single-element fetch within a loop by a block fetch, provided that the loop can be distributed around the fetch operation. A compiler could not determine if this transformation is legal due to lack of knowledge about I/O system call side effects. However, annotations that would enable such an optimization would be fairly easy for a library designer to specify [6], [46], [68].

An important issue yet to be addressed in our work is how to express the high-level transformations and their preconditions. For example, the preconditions must be able to in-

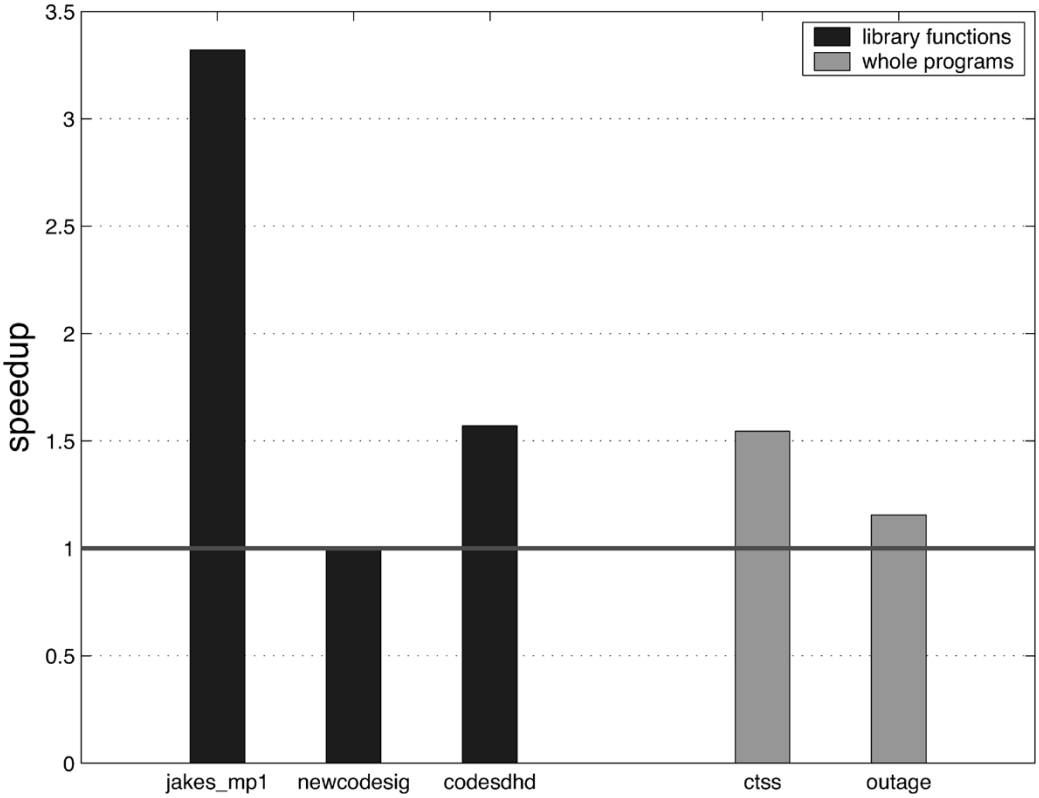



**Fig. 7.** Performance improvements due to procedure strength reduction.

clude specification of dataflow and dependence relations as well as loop containment. Our preliminary implementation uses an XML specification that indicates, for each transformation, the code pattern to be matched and a replacement code pattern. This suffices for the applications we have undertaken to date, but is unlikely to be adequate to meet all our goals in the long run.

To be effective, a specification language must make it easy to express transformations at a high level in a compiler independent fashion. However, we believe that it must also be possible to specify transformations with complex preconditions and effects. For example, how to express high-level transformations to achieve effects such as loop fusion is an open problem. In our research, we are exploring a variety of strategies for expressing transformations at a high level that enable complex semantic preconditions and postconditions to be specified and exploit semantic information in a significant way. One key issue is how to incorporate program dependence information, such as that required to express transformations like procedure vectorization and procedure reduction in strength, into our specification language. Our current XML-based specification strategy does not support the explicit use of dependence in the development of preconditions. We are currently exploring the use of predicates that explicitly query the results of dependence analysis in transformation preconditions.

Once the transformations are specified, Palomar must construct the substitution phase. There are two key issues here. First, how can we convert human-friendly transformations into a macro-substitution phase? One straw-man proposal is to take specifications, consisting of source code plus human-friendly annotations, and convert them to the XML specification for substitutions used in our current translator. Second, how can we prioritize transformations based on a reasonable cost model? Our current prototype uses an *ad hoc* approach in which transformations are applied in a specified order, but this will clearly be inadequate for the longer term. A better strategy may be found in the use of techniques from artificial intelligence, based on the profitability considerations described in the next few paragraphs.

Although transformation rules can indicate equivalence between two different sequences of operations, to use transformation specifications as the basis for optimization requires that we describe when the rule should be applied. To guide application of a rule, we will associate a quantitative cost model with its left- and right-hand sides to determine when it is profitable. In a compiler for tensor contraction expressions, Cociorva *et al.* [17] used dynamic programming to choose among implementation alternatives.

Database query optimizers provide a model for transformation strategies that can be potentially employed by the library-aware optimizer. Query optimizers use two-phase plans that involve first applying algebraic identities and axiomatic rewriting rules, and then using a cost-model-driven specialization of query operators [29]. The strategies employed in this specialization process are similar to those that our library-aware optimizer could use to explore the transformation space. These strategies include heuristic selection rules (with or without considering cost); naive exhaustive search; local search (e.g., hill climbing); branch and bound; dynamic programming (which records only the least cost alternative at each step in a bottom up enumeration); and

Selinger-style bottom-up search [3], which extends dynamic programming by keeping not only the least cost alternative at each point, but also a set of "interesting alternatives" that have properties that may be useful in other stages.

Heuristics for transforming database queries often have analogues in scientific computations. For example, greedy join ordering—combining the pair of relations that produces the smallest result first—can be likened to the heuristic for ordering a sequence of matrix multiplications—multiply the pair of adjacent matrices with the longest common dimension first.

A key choice when selecting physical query-operator implementations for performing a set of logical database operations is whether to use materialization (producing a value in its entirety), or pipelining (producing it incrementally). The alternatives available to our library-aware optimizer are whether to use a fully vectorized version of an operation, which materializes the largest possible vector result, a scalar version of a primitive, or some intermediate level of vectorization. The fully vectorized version may be the most efficient way of performing the operation in isolation, but at the cost of instantiating a large temporary array. The scalar version, possibly inlined, can be composed with other scalar operators in a pipeline that minimizes the size of temporaries. This improves memory hierarchy performance, but at the cost of increased pressure on registers and functional units. Vectorizing at intermediate granularities (for example, rows, columns, or tiles of arrays) will enable the optimizer to explore the space of tradeoffs.

In future work on generating an effective transformation phase, we plan to explore all these alternatives with the goal of producing truly effective optimizations based on the algebra of relations specified by the library designer.

### C. Context-Driven Specialization

As described in the introduction to Section IV, at script compilation time the generated optimizer will perform a preliminary analysis of abstract types; then, at each call site for a routine in the domain library, the optimizer will use the determined types of parameters to the called routine to select the most specialized version of the library routine that can be legally substituted at that call site. Given that we expect the number of specialized variants to be large, we will be using a fast selection algorithm that is a variant of unification from the domain of theorem proving.

In this section, we will describe the steps that the Palomar compiler generator must perform to support the specialization step and to make it more effective. Generally these tasks will fall into two categories. First, Palomar must determine the set of specialized variants to actually provide. The key issue here is how to avoid a combinatorial explosion of different variants. Second, once the type signatures for all variants to be included in the database are determined, Palomar must generate the specializations that are needed.

*Minimizing the Number of Variants:* Previous work on whole-program compilation has shown the value of interprocedural dataflow information (a form of context) applied to traditional languages [1], [7], [8], [21], [34]–[36], [48], [53], [64]. For example, procedure inlining [20], [22] is beneficial not only because it removes call/return overhead but also because it provides a context for optimizing the procedure's code. Unrestricted inlining can be detrimental, however, because it can place a significant burden on later compilation stages, particularly the register allocators. In addition, using inlining exclusively is an unsuitable approach for implementing telescoping languages because it may lead to excessively long compilation times even for short scripts.

An alternative to inlining, *procedure cloning* [18], [19], can generate specialized variants of procedures that are tailored to the calling context. Unfortunately, instantiating a clone for every potential calling sequence that might be encountered at script compilation time would be impractical, hence the need to prune the number of variants actually generated by Palomar.

Clearly there is a large space of alternative specializations to be explored, so limiting the number of specializations to those that are most likely to be useful is a major focus of our research. Generally there are two ways to do this.

- The first is to assess likelihood through the analysis of hints and specifications provided by the library developer. We have already seen in Section IV-A that annotations by the library designer will determine many of the required variants. However, it will undoubtedly prove useful to provide automatic ways of determining useful variants. To accomplish this, Palomar will perform some optimizations speculatively, "guessing" at the likely (and useful) contexts in which specific library components might be invoked. If we are to successfully limit the number of different variants generated speculatively, we must identify "interesting" contexts that can lead to substantive performance improvements. Sample calling sequences should provide a rich source of such hints. However, we plan to augment this information with feedback from the generated script optimizer. If a large number of programs that are actually presented to the optimizer could benefit from a specialization not present in the preprocessed domain library, then that specialization could be scheduled for generation offline. This generation step might be run at more frequent intervals than the entire language generation process because it is restricted to single routines.
- The second approach is to use information about profitability of optimizations to guide generation of specializations. For example, if for a given library component there are two potential specializations, one slightly more general than the other, and the more general version is only slightly less efficient than the specialized one, it may be prudent to include only the general version because it covers more cases. On the other hand, if the potential performance improvement is huge and there are indications that the more specialized case occurs frequently in practice, then both specializations should be generated.

In the first prototype implementation, we are focusing exclusively on designer-specified specialization strategies, although we intend to include context based specializations like procedure strength reduction. However, in the longer term, we expect to automate the generation of specialization annotations by having the Palomar interprocedural analysis system produce "suggested" annotations for the designer. These annotations can be augmented with those that are produced during actual script compilations. If the results of these processes are acceptable, it may significantly reduce the burden on library designers and developers.

*Variant Generation and Specialization:* We plan to base the generation of specialized versions of library routines on our type analysis strategy, described in Section III-C. Type analysis not only gives us a tool for constructing return type jump functions to be used for propagating types across library calls, it can also drive the generation of specialized variants through its specification of the abstract types used at each point within the original library routine. In addition, the algorithms we are producing will make it possible through backward analysis to limit the space of input parameter types to those that make sense.

In addition, as discussed in Section IV-B, we plan to go beyond simply specializing procedures for particular contexts—we aim to use information about potential calling contexts and procedure implementations to decompose and recombine library routines, creating a new library of streamlined primitives for high-performance programs. Thus, a library routine that performs multiple operations, some of which are not needed in every context, will be decomposed so that each operation can be invoked selectively.

As an example, the library operation invoked by the R interpreter to add together the values encapsulated in two data objects checks that they are numeric types and performs coercions as necessary, checks for conformable shapes of objects, allocates a result object, and then performs the operation. The checking and coercion operations should be explicitly exposed so that traditional compiler optimizations such as common subexpression elimination and hoisting invariants out of loops can then avoid unnecessary recomputations. Similarly, rather than allocating storage for a result inside the arithmetic routine, it would be better to expose the allocation and allow reuse of the storage using a combination of region analysis [62], [63] and explicit annotations.

Where profitable, composite routines will be constructed by combining two or more separate procedures to perform a computation more efficiently. For example, in S one might write an assignment statement $\mathtt{d} \leftarrow \mathtt{a} + \mathtt{b} * \mathtt{c}$, where a, b, c, and d are all objects coercible to vectors of the same size. S interpreters would compute this statement by first calling primitive routines to do type and size coercion if necessary, then calling a routine to perform the vector multiply into a temporary vector variable, and, finally, calling a separate primitive to perform the vector addition. If this represents a common idiom, it would be better to create a family of type-specific composite routines, each of which fuses these operations into a single loop. This avoids creation of multiple full-size temporary vectors and it potentially allows the use of a multiply–add machine instruction on the target platform.

Once such variants are identified, we will use various program transformation and optimization strategies to produce the actual code for specialized variants. Generally, we will take the combinations of routine bodies that are chosen for specialization and integrate them using well-known optimization strategies.

For example, in the case of procedure strength reduction [13], described in Section IV-B, Palomar would perform the partitioning of a selected function into an initialization and iterative version by using *program slicing* to select the code in the function that depends on the containing loop induction variable, leaving the code that is independent of the loop induction variable in the initialization function. (A better strategy, which we plan to use, is to construct the initialization routine by symbolically executing the function using the value of the induction variable on loop entry and to use program differencing to determine how to compute the iterative function value from the value for the previous iteration.)

In the case of procedure vectorization, Palomar would construct an initial procedure body by moving the vectorized loop into the procedure body, then applying loop distribution to see if further vectorizations could be achieved recursively.

### D. Implementation Status

To date, the implementation effort has produced preliminary prototypes of the script compilers for S and MATLAB and is well on the way to the first prototype of Palomar itself. These milestones are described in the paragraphs below.

*MATLAB Compiler:* We have developed a rudimentary compiler for MATLAB by coupling the type analysis routine described in Section III-C with a code generator that produces C with calls to the appropriate LAPACK routine. In addition, this compiler uses program slicing to move array allocations outside of loops. Because LAPACK is being used as the implementation library, the current compiler only supports linear algebra. The portion of the language for handling generalized objects is not yet implemented, although some of this will be required to extend the library generation to model reduction as described in Section V-B.

The performance of the code generated by this compiler is illustrated later (see Figs. 10 and 11, which appear in the discussion of LibGen, an application of the MATLAB compiler to the problem of library maintenance, in Section V-B).

*RCC:* The R language [38] is a modernized version of the award-winning S language for statistical computing. We have developed RCC [30], a prototype domain-language translator from R scripts to C code, by leveraging the open-source infrastructure for R.[3] At present, our translation of R into C is fairly straightforward. The C code generated by RCC constructs instances of data types compatible with the R runtime library and makes calls to the R runtime system to perform each language primitive operation. RCC translates each R function into a C function that can be called

[3][Online] Available: http://www.r-project.org

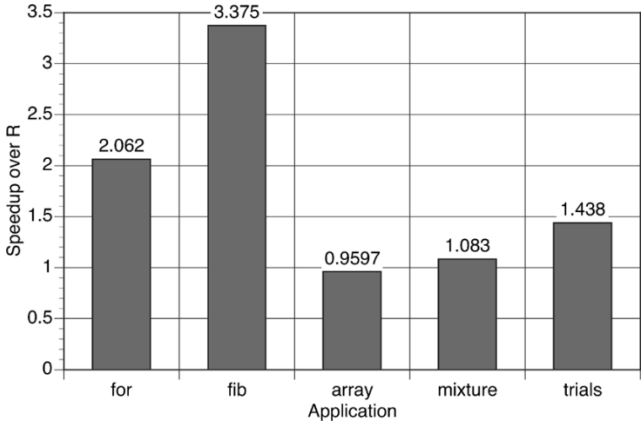



**Fig. 8.** Performance of RCC-generated code versus the R interpreter.

from other compiled code or R using the foreign code calling interface. RCC directly generates C control flow for R loops and conditionals. Construction of R data representations for invariant values is hoisted into an initialization routine to avoid unnecessary redundant synthesis at run time.

Fig. 8 shows the performance of five R programs compiled with RCC compared to their performance under the R interpreter. Each of the bars shows the ratio of the running time of the interpreted program to that of the same program compiled with RCC: numbers above one represent speedups, while numbers below one are slowdowns. The test programs are a combination of microbenchmarks (for, fib, array) and applications used by researchers at the M. D. Anderson Cancer Center (mixture, trials). The `for`, `fib`, and `array` microbenchmarks are designed to measure repeated function calls, deeply recursive functions, and array operations, respectively. `mixture` is a Bayesian mixture analysis code. `trials` performs calculations used in clinical trial design.

In general, the performance of code generated by RCC is similar to interpreted code, since it calls runtime library primitives much in the same fashion as the interpreter. RCC provides a building block that will enable us to apply a domain-language-independent analysis and optimization framework (developed to support analysis and optimization of library-based programs in C) for high-level optimization of R scripts. Compilation does eliminate much of the interpreter's overhead due to function calling, list allocation, and evaluation of code objects. RCC compilation improves performance significantly for `for` and `fib`, microbenchmarks with large numbers of function calls, and `trials`, a large application with significant evaluation overhead. In contrast, the programs `array` and `mixture`, which consist mostly of array operations, show compiled performance roughly equal to interpreted performance.

*Palomar:* Our preliminary research has led to the development of a number of the software technologies that are to be part of Palomar. For example, the type analysis mechanism described in Section III-C provides information needed about possible executions of a library routine to generate variants specialized for different possible calling contexts. Specifically, it is able to determine which variants are necessary so that every valid execution of the library routine is handled (e.g., one for each inferred type tuple over the inputs). It also provides the type information needed to generate specialized paths within the variants that handle different possible outcomes of control flow. Thus, the dynamic behavior of the routine is captured by the variants. Specialization of the user application or other library routines replaces the calls to the general library routines with calls to the appropriate specialized variants. We have also implemented some of the automatic specialization strategies, including procedure reduction in strength and procedure vectorization, that are planned for the library-aware optimization phase. These capabilities have been critical to the various application efforts we have been involved in (see Section V).

The first prototype of the Palomar system will be based on the type-based specialization strategy described above, which is elaborated in more detail in the discussion of LibGen in Section V-B. The organization of this system is presented in Fig. 9. In this prototype, the specializations are guided in part by annotations from the library designer that provide some of the type tuples to be expected for each library routine. The type analysis system is then used to generate both a database of *return type jump functions*, which describe the output types from a library function for a given set of input types, and *type jump functions*, which can be used to compute the specific type of each data element in a function given the input data types. The former would be used during type analysis for scripts and other library routines that invoke library routines while the latter make it possible to convert the dynamic types in MATLAB to static types in C or Fortran for each variant. Using these type jump functions, the code generation system simply generates a specialized version of each routine for each input type tuple inferred by type inference system, saving these specializations in the database.

Once the two databases are available, the script compiler can perform global type analysis on the script without tracing through library routines by employing the return type jump functions. This process determines the input types for the parameters at each invocation of a library routine in the script; the C code generator then selects the most specialized variant of each routine that is compatible with the given input types.

Currently, most of the top of Fig. 9 is in place. Although a C code generator exists, it does not yet incorporate type-based variant selection. Because this feature is fairly straightforward to implement, we expect that it will be incorporated into Palomar by the time this paper appears.

Building on this preliminary effort, we are designing a more sophisticated Palomar optimizer generation framework that will include the advanced analysis and optimization strategies discussed earlier in this section. For the first iteration of this new version, we are concentrating on automatic management of the number of variants produced for the specialization database and on construction of the high-level transformation system from annotations by the library developer.

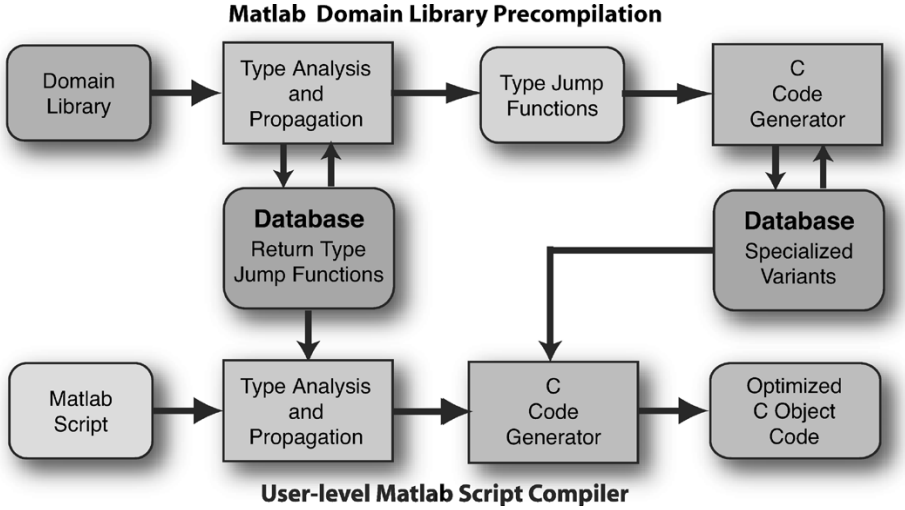



**Fig. 9.** Organization of the first Palomar prototype system.

## V. Applications

A major focus of our preliminary work has been the structure and function of the Palomar system itself: in particular, what strategies the Palomar generator should employ to support the speculative procedural optimizations that are needed in the library aware analysis and optimization phase.

Our work on the Palomar design, described in Section IV, has been driven by two preliminary application studies—MATLAB for signal processing applications and the LibGen library generation system— and three longer term application studies. Here we describe these applications in more detail.

### A. MATLAB for Signal Processing

MATLAB is hands-down the language of choice for prototyping signal processing applications. It is also ideal for telescoping languages because successful prototypes are almost always rewritten in C to achieve high performance, which in some cases means fitting into the memory of an embedded processor. With funding from the State of Texas, we carried out a small project in collaboration with researchers in Rice University's Electrical and Computer Engineering Department to improve the performance of digital signal processing applications written in MATLAB.

In addition to uncovering a number of important insights into the compilation of MATLAB, this work has validated the fundamental hypothesis of telescoping languages that substantive performance improvements can be achieved by transforming a program to use specialized versions of procedures in common contexts, such as loops. Two particularly useful transformations, whose applicability extends beyond the domain itself, are procedure vectorization and procedure strength reduction [13], introduced in Section IV-C. These optimizations can be effectively applied in the translator generation process.

Traditional compilation systems tend to use inlining to get whole-program optimizations. This is not the most desirable strategy for domain-specific languages because it increases the size of single procedures to the point where compile times, which can increase nonlinearly with procedure size, become intolerable. The lesson from our MATLAB study is that it may be possible to discover an algebra of procedure optimizations that are analogs of operation-level optimizations performed in conventional compilers (see Section IV-B). Such optimizations may make it possible to get the benefits of powerful optimization strategies without excessive inlining into the user scripts.

### B. Library Maintenance Using LibGen

Our colleague D. Sorensen of the Computational and Applied Math Department is the principal developer of the ARPACK library for large-scale eigenvalue problems. He currently prototypes these libraries in MATLAB and then translates them by hand to eight different variants (real and complex, dense and sparse, symmetric and nonsymmetric matrices) written in Fortran. His MATLAB prototype for the key routine *ArnoldiC* currently takes about a page of MATLAB to express.

We hypothesized that our basic MATLAB compilation strategy (described in Section III), which is based on a high-level type analysis system and is used to generate base language code from scripts written in MATLAB, might obviate the need for the hand translation step, the most time-consuming part of the development of ARPACK. As a demonstration, we undertook a project to construct a translator that could generate type variants for any MATLAB-specified library routine. We have come to call this library generation system "LibGen," for "Library Generator."

Using LibGen, we have been able to transform Sorensen's MATLAB program for *ArnoldiC* to produce Fortran versions in all eight variants with performance comparable to his hand-coded Fortran routines [44]. The performance results for two of the variants are shown in Figs. 10 and 11. These runs are on matrices of size 3200 by 3200 that are both sparse; however, the dense numbers employ a dense representation in MATLAB. Numbers for the MATLAB versions are from the MATLAB 6.1 interpreter. The bars labeled

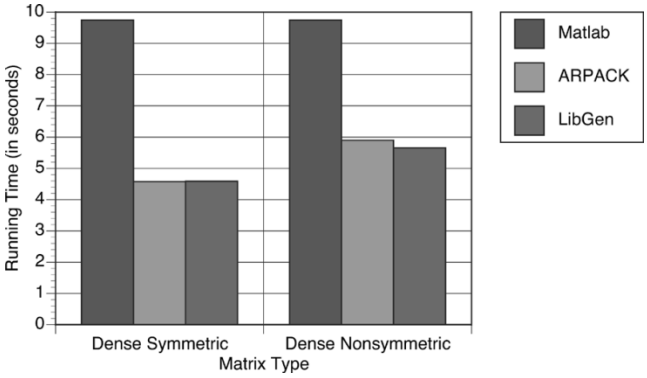



**Fig. 10.** LibGen versus hand-coded ARPACK on dense symmetric matrices.

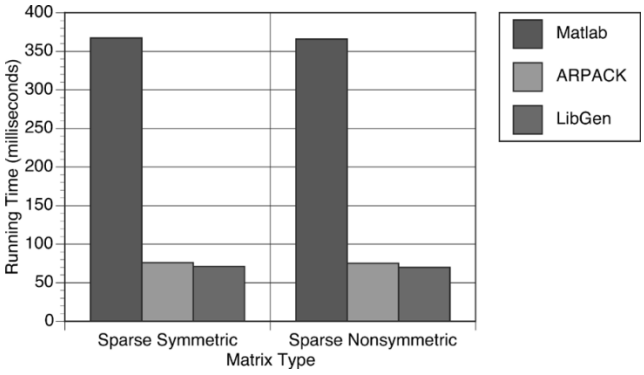



**Fig. 11.** LibGen versus hand-coded ARPACK on sparse symmetric matrices.

"ARPACK" represent the running times for the hand-coded Fortran, while the bars labeled "LibGen" are for the Fortran versions generated by our system. The LibGen-generated versions perform slightly better than ARPACK because the hand translation introduced overheads in the library interface that are not present in the original MATLAB version.

These results make it clear that, with the complete Palomar framework, we should be able to make hand translation completely unnecessary. However, Palomar would also make it possible to produce a script compiler that would serve as an integration system for the library: a kind of component integration system. The result would be a powerful tool that could be applied to libraries in many other domains. The use of Palomar as a component integration system will be discussed in Section V-E.

### *C. Computationally Intensive Statistics*

Computationally intensive statistical methods, such as Bayesian analysis, are required for many emerging problems; for instance, the analysis of very large data sets. For many such analyses, a new statistical application specifying the precise statistical model and computation required must be developed. S is the language preferred by many statisticians for developing new statistical models and methodologies, partly because it includes native support for vector and matrix data and operations, and intrinsic high-level, complex statistical operations. S is also popular because there is a very large, widely available collection of external libraries and packages, many written in S itself, for computing an extensive range of complex statistical functions. Consequently, S greatly facilitates the rapid development of new statistical models and methodologies.

For computation-intensive applications, however, S programs are far too slow for direct use. When presented with a computation-intensive problem, many statisticians who prefer S nevertheless abandon it for a lower level language, such as C or Fortran, that is known to be more efficient. A substantial drawback of this approach is that while a newly conceived statistical method is being transformed into a program, considerable time passes before even a preliminary numerical evaluation of the new method can occur. Another common approach is to develop a prototype in S and then rewrite, or employ professional programmers to rewrite, the S programs in an efficient, low-level language. As well as being extremely time consuming, this rewriting step introduces new possibilities for error. Both approaches require staff versed in statistics, programming, and often high-performance computing. Thus, there is no ideal language in which to rapidly design, prototype, and perform full-scale evaluation of computation-intensive statistical methods, creating a fundamental bottleneck retarding the ability of statisticians to solve many emerging statistical problems.

For many applications, time to solution is critical. For instance, before a trial of a new cancer drug can begin, cancer researchers must conduct an experiment to assess the efficacy of the treatment and to maximize the accuracy of estimation of confidence intervals for the relevant parameters associated with the trial's results. These experiments must be designed to minimize the risks and maximize the benefits to all individuals participating in the trial. Since each trial schema is different, a new experimental design must be created and evaluated for each trial. Usually the exact efficacy for specific cohorts undergoing the new treatment is unknown, so extensive simulations of the experimental design must be used to determine the design's operating characteristics under different assumptions about the new treatment's actual efficacy. These operating characteristics can then be used to select the optimal design. Similarly, new methodologies [25], [26], [59], [60], [61], such as the analysis of gene expression microarray data and the inclusion of such data into a clinical trial, must also often be evaluated by extensive simulation. Clearly, it is highly desirable to obtain the results of these computation-intensive simulations as quickly as possible, to enable new clinical trials to commence as soon as possible.

Currently, our collaborators at the M. D. Anderson Cancer Center—K.-A. Do, P. Mueller, and P. Thall—develop models of new experimental trial designs and new statistical methodologies in S and employ programmers to rewrite the S programs in a low-level language. As well as considerably delaying the commencement of new trials, this process also effectively excludes, for all except very small trials, the possibility of adaptive designs in which interim results can be used to guide subsequent redesigns in the latter part of a trial.

Time to solution is also paramount for many other applications, such as the analysis of massive quantities of intercepted communications for information relevant to national security.

We are applying the telescoping languages strategy to the construction of a script compiler for statistics. Our preliminary studies described in Section III-A have shown that such a compiler, generated by Palomar, could make the manual rewriting step unnecessary, which would dramatically speed up the development of new computationally intensive statistical methodologies.

### D. Image Processing

MATLAB is widely used to express many image processing and analysis algorithms. MathWorks, Inc., provides an image processing toolbox that efficiently "provides a comprehensive set of reference-standard algorithms and graphical tools."[4] However, these tools are not, by themselves, sufficient to address the needs of researchers who are adding new functionality at the forefront of the field. Our goal in this effort is to support such researchers.

The processing of large images requires enormous amounts of computation. To achieve reasonable performance on such problems, the algorithms must be coded to avoid particular language constructs, such as loops, wherever possible. Unfortunately, because of memory hierarchy issues, it is not practical to replace loops over an entire image with whole-array operations. In some cases, a special routine (blkproc) is used to apply a function to all the subblocks of an image without incurring MATLAB's loop overhead. These issues detract from MATLAB's suitability for computation-intensive image processing, requiring extensive manual optimization of the MATLAB program or its translation into a lower level language such as C. Our preliminary analysis of signal processing applications in MATLAB suggests that telescoping languages can help overcome these problems.

We have enlisted the collaboration of two Rice University faculty members, R. Baraniuk of Electrical and Computer Engineering and W. Symes of Computational and Applied Mathematics, to help us apply the telescoping languages technology to their problem domains. Both of these researchers would prefer to use MATLAB instead of low-level languages. Baraniuk currently uses MATLAB, but execution times are problematic. Symes abandoned MATLAB, primarily because of poor computational efficiency due to memory-hierarchy effects mentioned earlier (see the discussion of component integration below for another reason). Specific algorithms whose MATLAB implementations are inefficient include wavelet-based statistical signal processing [23] and image segmentation [16], [49].

In collaboration with these researchers, we plan to identify program transformations that improve the efficiency of these and other image processing algorithms and to discover strategies for automatically applying these transformations to MATLAB libraries and applications. Our goal is to create an environment in which image processing applications can be expressed in natural, high-level MATLAB, that is, without (significant) manual transformation aimed solely at increasing performance, and in which these programs will execute, after automatic transformation, with similar performance to the best manually optimized code.

[4]See http://www.mathworks.com/products/image/

### E. Component Integration and Parallel MATLAB

One of the major challenges to the goal of increasing the productivity of programming professionals is developing strategies to enhance software reuse. Although component integration frameworks (e.g., COM [56] and CORBA [50]) are widely employed in commercial software development, they are generally considered too inefficient for scientific software development. The Common Component Architecture (CCA) [2] initiative is trying to reduce the overheads to levels that are acceptable to computational scientists.

Nevertheless, CCA-compliant component frameworks still suffer from performance difficulties at the interfaces for two reasons. First, they support dynamic component selection which adds significant overhead to method invocation and compounds the standard performance penalties due to dynamic dispatch in object-oriented programs. Second, components are treated in most frameworks as black boxes to which cross-procedural optimizations cannot be applied. Thus, opportunities for great improvements—in some cases, by integer factors—are lost. These issues are particularly critical when a component implementing a data structure (e.g., sparse matrix) is to be integrated with a component implementing functionality (e.g., linear algebra). If component integration technology can be developed to make it possible to separate data representation from functional behavior without sacrificing performance, it will dramatically improve the effectiveness and applicability of component-based programming.

We plan to attack this problem using telescoping languages. Our initial strategy will be to abstract the matrix representation by requiring a matrix to provide certain methods, including those needed to get and put a single element and get a row or a column. These methods will be invoked by functional code in place of the usual matrix addressing and access operations. This approach is not unique—other research groups have developed such abstractions [43], [51]. The value added by our approach is that sequences of operations on the data structures can be optimized in context by the Palomar-generated script compiler. A facility like this would also make it possible to include out-of-core arrays in MATLAB—the absence of this capability is the second reason that our collaborator W. Symes (see Section V-D above) abandoned MATLAB.

The key to success is for Palomar to generate a compiler that can carry out analogs of standard addressing optimizations on sequences of accesses to the new access primitives. An example is reduction in strength, by which the multiplications required by standard matrix addressing are replaced by additions when successive elements are accessed in a loop. This could be accomplished in the new matrix representation

by keeping track of the currently referenced element and providing a fast *next* entry to access the next element along some dimension of the matrix. The code for this could be inlined in the loop to ensure that performance of stepping through an array is optimized.

A side benefit of this approach is that it leads to a strategy to incorporate parallelism in array scripting languages such as MATLAB. If we can extend MATLAB to deal with arrays written in terms of the more general representations described here, we can include the notion of *data distributions* for arrays, which could specify that an array be allocated across the nodes of a parallel machine. In addition to the standard distributions (e.g., block and cyclic), the developer could provide components implementing sparse or adaptive structures, such as those based on space-filling curves. The Palomar generator would speculatively apply both the standard optimization and others specific to parallelism, including distributed array allocation and communication scheduling. Our research team is in an ideal position to explore this strategy because we have already developed a rich infrastructure for compiling distribution-based parallelism in High Performance Fortran [45]. This strategy for parallelizing MATLAB, which generalizes the approach taken by Otter [52], is appealing because it preserves the simplicity of MATLAB while permitting sophisticated data-structure developers to exploit parallelism with help from the compiler and runtime system.

## VI. Related Work

Many projects have created optimizing MATLAB compilers [15], [24], [46], [52], [54], and we will use and extend their techniques as appropriate. Palomar, however, will attack a more general problem: using high-level semantics to automatically construct optimizers for domain-specific languages.

The telescoping languages approach shares much in common with other research [65] into simplifying the generation of high-performance domain-specific language translators. Palomar specifically requires new advances in speculative procedure specialization and high-level program transformation driven by annotations provided by library designers.

Recent work by Cociorva *et al.* [17] uses dynamic programming and (empirically derived) cost estimates to choose among alternate parallel implementations of tensor contraction expressions. We expect to use a similar approach to reason about implementation variants, although in the context of more arbitrary code fragments and a wider set of transformations.

Previous work on whole-program compilation has shown the value of interprocedural dataflow information (a form of context) applied to traditional languages [1], [7], [8], [18]–[22], [34]–[36], [48], [53], [64].

However, these techniques are not sufficient for telescoping languages because they require the libraries to be included with the script in analysis and optimization. Instead, our library preprocessing phase performs as much of this analysis as possible offline.

The problem of how to specify transformations has been studied by a number of other research projects. The code composition approach of Catacomb [58] and expression templates [66] enable optimizing transformations to be coded independent of the compiler's intermediate representation; however, while these approaches enable sophisticated expansion of individual operations tailored for their context, they do not support optimization across separate operations. Alternatively, Lacey and de Moor [42] describe a general method for specifying imperative program optimizations as rewriting rules. Their transformations specify predicates about what properties must match at certain nodes (e.g., definition or uses) along with a temporal logic notation that specifies the truth of predicates over paths between the nodes involved. They show that conventional optimization strategies such as constant propagation, dead code elimination, and operator strength reduction can be described conveniently using their notation. However, Lacey and de Moor analyze only basic properties of nodes; user-defined properties would also be useful. In our research on the specification of transformations, we plan to build on this work.

Procedure specialization has been widely researched in the literature on interprocedural analysis [19], [31] and in work on partial evaluation [5], [39]. Our strategy differs from the previous work in that we aim not only to produce specializations of individual procedures, but also to explore opportunities for decomposing and recombining procedures. Since we will perform these transformations during library preprocessing, when calling context is unknown, we must prepare specializations for any expected contexts that could lead to significant cost savings.

Previously, researchers have explored axiomatic approaches for optimizing domain-specific languages. Menon and Pingali [46], [47] describe a set of hand-coded axioms for matrix operations that are applied in a particular order to optimize MATLAB programs. Weaver *et al.* [67] describe a compiler representation that allows incorporation of user annotations concerning library properties such as associativity, commutativity, and identity. We plan to extend this work by deriving the optimizer automatically from the given axioms.

Guyer and Lin [31], [32] describe a compiler-based framework and an annotation language that allows a library developer to guide optimization of calls to library operations. Their notation language allows user-defined properties, but only of enumerated values in a two-level lattice. The annotations also can indicate when to remove an operation or replace it with a more specialized operation. We hope to extend their work in pursuit of our goal of optimizing sequences of library calls.

High-level rewriting systems for program optimization have been extensively studied for functional languages [57] and logic-based languages [55]. For instance, Jones *et al.* [40] describe a simple rewriting system that is included in the released Glasgow Haskell Compiler. However, implementing rewriting systems for imperative languages is much more difficult, requiring nontrivial program analyses. For

Palomar, the required analyses and their role in controlling program transformation must be described by a general, yet simple, notation.

The MAGIK system [27] gives advanced users the ability to include domain-specific semantics into compilation, enabling high-level interface optimizers and checkers. MAGIK is extended by dynamically linking C code with access to the system's intermediate representation. The KHEPERA system [28] is similar, but includes a little language for coding transformation rules. Both approaches require the language-extension designer to know considerable detail about the compiler's internal representation and make language extensions dependent upon a particular compiler implementation. In contrast, for Palomar we aim to use general annotations to portably describe domain-language transformations independent of details of the compiler generator's implementation.

## VII. CONCLUSION

Although users appreciate the convenience (and, thus, improved productivity) of using relatively high-level scripting languages, the slow execution speeds of these languages remain a problem. Lower level languages, such as C and Fortran, provide better performance for production applications, but at the cost of tedious programming and optimization by experts. If applications written in scripting languages could be routinely compiled into highly optimized machine code, a huge productivity gain would be realized.

If we are to achieve this goal, we must recognize that conventional compilation technology, by itself, may not be enough. In practice, scientists typically extend these languages with their own domain-centric components, such as the MATLAB signal processing toolbox. Doing so effectively defines a new domain-specific language. To address efficiency problems for such extended languages, we must develop a framework for automatically generating optimizing compilers for them.

Our group at Rice University has been exploring an innovative strategy called *telescoping languages* that uses a library preprocessing phase to extensively analyze and optimize collections of libraries that define an extended language. Results of this analysis are collected into annotated libraries and used to generate a library-aware optimizer. Since this preprocessing phase need be done only at infrequent "language-generation" times, its cost can be amortized over many compilations of individual scripts that use the library. The generated library-aware optimizer, which will be run much more frequently to translate individual scripts, uses the knowledge gathered during preprocessing to carry out fast and effective optimization of high-level scripts. This enables scripts to benefit from the intense analysis performed during preprocessing without repaying its price.

Research to date has followed two lines of investigation:

- compiler technology for MATLAB and S using constraint-based type analysis to specialized calls to the runtime libraries based on the operand types;
- compilation and optimization strategies within the Palomar framework for generation of library-aware optimizers.

This work has led to prototype compilers for MATLAB and S, along with a prototype of the Palomar system based on the MATLAB compilation system. We are in the process of developing a more sophisticated Palomar system based on the Open64 compiler infrastructure.

We are driving our research by exploring important applications of the telescoping languages strategy, including: 1) signal and image processing applications in MATLAB; 2) statistical calculations from scientific disciplines written in the language S; 3) library maintenance and generation systems, such as LibGen; and 4) component integration frameworks for scientific software. This work has already led to significant insights and preliminary results that demonstrate performance improvements that are significant enough to make recoding script-based applications in C or Fortran unnecessary.

## ACKNOWLEDGMENT

The authors would like to thank the referees for their numerous constructive suggestions, which substantially increased the quality of this paper.

## REFERENCES

[1] F. Allen, "Interprocedural Data Flow Analysis," IBM T. J. Watson Research Center, Yorktown Heights, NY, Comput. Sci. RC 4633 (#20 545), 1973.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. Mcinnes, S. Parker, and B. Smolinsky, "Toward a common component architecture for high performance scientific computing," in *Proc. High Performance Distributed Computing Conf.*, 1999, p. 13.

[3] P. G. Selinger, M. M. Astrahan, R. Chamberlain, R. A. Lorie, and T. Price, "Access path selection in a relational database management system," in *Proc. 1979 ACM SIGMOD Conf.*, 1979, pp. 23–34.

[4] R. A. Becker, J. M. Chambers, and A. R. Wilks, *The New S Language*. London, U.K.: Chapman & Hall, 1988.

[5] A. Berlin and D. Weise, "Compiling scientific code using partial evaluation," *IEEE Computer*, vol. 23, no. 12, pp. 25–37, Dec. 1990.

[6] B. Broom, R. Fowler, and K. Kennedy, "KelpIO: A telescope-ready domain-specific I/O library for irregular block-structured applications," in *Proc. 2001 IEEE Int. Symp. Cluster Computing and the Grid*, pp. 148–155. Joint best paper in the cluster computing category.

[7] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," presented at the SIGPLAN '86 Symp. Compiler Construction, Palo Alto, CA.

[8] D. Callahan, J. Cocke, and K. Kennedy, "Analysis of interprocedural side effects in a parallel programming environment," *J. Parallel Distrib. Comput.*, vol. 5, no. 5, pp. 517–550, Oct. 1988.

[9] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," presented at the ACM SIGPLAN '86 Symp. Compiler Construction, Palo Alto, CA.

[10] J. M. Chambers, *Programming with Data*. New York: Springer-Verlag, 1998.

[11] J. M. Chambers and T. J. Hastie, *Statistical Models in S*. London, U.K.: Chapman & Hall, 1992.

[12] A. Chauhan and K. Kennedy, "Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization," presented at the 15th ACM Int. Conf. Supercomputing, Sorrento, Italy, 2001.

[13] ——, "Reduction in strength of procedures: An optimizing strategy for telescoping languages," presented at the 2001 Int. Conf. Supercomputing, Sorrento, Italy.

[14] ——, "Slice-hoisting for array-size inference in MATLAB," presented at the 16th Workshop Languages and Compilers for Parallel Computing (LCPC'03), College Station, TX.
[15] S. Chauveau and F. Bodin, "Menhir: An environment for high performance MATLAB," *Sci. Program.*, vol. 7, pp. 303–312, 1999.
[16] H. Choi and R. Baraniuk, "Multiscale document segmentation using wavelet-domain hidden Markov models," presented at the IST/SPIE 12th Annu. Int. Symp. Electronic Imaging 2000, Science and Technology, San Jose, CA, 2000.
[17] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C.-C. Lam, P. Sadayappan, and J. Ramanujam, "Global communication optimization for tensor contraction expressions under memory constraints," presented at the International Parallel and Distributed Processing Symp., Nice, France, 2003.
[18] K. Cooper, M. W. Hall, and K. Kennedy, "Procedure cloning," in *Proc. 1992 IEEE Int. Conf. Computer Language*, pp. 96–105.
[19] ——, "A methodology for procedure cloning," *Comput. Lang.*, vol. 19, no. 2, pp. 105–117, Feb. 1993.
[20] K. Cooper, M. W. Hall, and L. Torczon, "An experiment with inline substitution," *Softw. Pract. Exper.*, vol. 21, no. 6, pp. 581–601, Jun. 1991.
[21] K. Cooper, K. Kennedy, and L. Torczon, "The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 4, pp. 491–523, Oct. 1986.
[22] K. D. Cooper, M. W. Hall, and L. Torczon, "Unexpected side effects of inline substitution," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 22–32, Mar. 1992.
[23] M. Crouse, R. Nowak, and R. Baraniuk, "Wavelet-based statistical signal processing using hidden Markov models," *IEEE Trans. Signal Process.*, vol. 46, no. 4, pp. 886–902, Apr. 1998.
[24] L. A. DeRose, "Compiler techniques for MATLAB programs," Ph.D. dissertation, Univ. Illinois, Urbana-Champaign, 1996.
[25] K.-A. Do, B. M. Broom, and S. Wen, "Geneclust," in The Analysis of Gene Expression Data: Methods and Software, G. Parmigiani, E. S. Garrett, R. A. Irizarry, and S. L. Zeger, Eds. New York: Springer-Verlag, 2003, to be published.
[26] K.-A. Do, B. M. Broom, and X. Wang, "Importance bootstrap resampling for proportional hazards regression," *Commun. Stat. Theory Methods*, vol. 30, no. 10, pp. 2173–2188, Aug. 2001.
[27] D. R. Engler, "Interface compilation: Steps toward compiling program interfaces as languages," *Softw. Eng.*, vol. 25, no. 3, pp. 387–400, 1999.
[28] R. E. Faith, L. S. Nyland, and J. F. Prins, "KHEPERA: A system for rapid implementation of domain specific languages," in *Proc. Conf. Domain-Specific Languages*, 1997, pp. 243–255.
[29] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database System Implementation*. Upper Saddle River, NJ: Prentice-Hall, 2000.
[30] J. Garvin, "RCC: A compiler for the R language for statistical computing," M.S. thesis, Rice Univ., Houston, TX, 2004.
[31] S. Guyer and C. Lin, "An annotation language for optimizing software libraries," in *Proc. 2nd Conf. Domain-Specific Languages*, 1999, pp. 39–52.
[32] ——, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proc. IEEE*, vol. 93, no. 2, pp. 342–357, Feb. 2005.
[33] B. Hahn, *Essential MATLAB for Scientists and Engineers*. London, U.K.: Arnold, 1997.
[34] M. W. Hall, "Managing interprocedural optimization," Ph.D. dissertation, Dept. Comput. Sci., Rice Univ., 1991.
[35] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. S. Lam, "Interprocedural parallelization analysis: A case study," in *Proc. 8th Workshop Languages and Compilers for Parallel Computing*, 1995, pp. 61–80.
[36] M. Hind, M. Burke, P. Carini, and S. Midkiff, "Interprocedural array analysis: How much precision do we need?," presented at the 3rd Workshop Compilers for Parallel Computers, Vienna, Austria, 1992.
[37] E. N. Houstis and J. R. Rice, "The engineering of modern interfaces for PDE solvers," in *Symbolic Computation: Applications to Scientific Computing*, E. N. Houstis, J. R. Rice, and R. Vichnevetsky, Eds. Amsterdam, The Netherlands: North-Holland, 1992, pp. 89–94.
[38] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *J. Comput. Graph. Stat.*, vol. 5, no. 3, pp. 299–314, 1996.
[39] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Greneration*. New York: Prentice-Hall, 1993.
[40] S. P. Jones, A. Tolmach, and T. Hoare, "Playing by the rules: Rewriting as a practical optimization technique in GHC," in *Preliminary Proc. 2001 ACM SIGPLAN Haskell Workshop*, pp. 203–233.
[41] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon, "Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries," *J. Parallel Distrib. Comput.*, vol. 61, pp. 1803–1826, Dec. 2001.
[42] D. Lacey and O. de Moor, "Imperative program transformation by rewriting," in *Lecture Notes in Computer Science, Compiler Construction*, R. Wilhelm, Ed. Heidelberg, Germany: Springer-Verlag, 2001, vol. 2027, pp. 52–68.
[43] N. Mateev, K. Pingali, P. Stodghill, and V. Kotlyar, "Next-generation generic programming and its application to sparse matrix computations," in *Proc. Int. Conf. Supercomputing*, 2000, pp. 88–99.
[44] C. McCosh, "Type-based specialization in a telescoping compiler for ARPACK," M.S. thesis, Rice Univ., Houston, TX, 2002.
[45] J. Mellor-Crummey, V. Adve, B. Broom, D. C. Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi, "Advanced optimization strategies in the rice dHPF compiler," *Concurrency: Practice and Experience*, vol. 14, no. 8–9, pp. 741–767, 2002.
[46] V. Menon and K. Pingali, "A case for source-level transformations in MATLAB," in *Proc. 2nd Conf. Domain-Specific Languages*, 1999, pp. 53–65.
[47] ——, "High-level semantic optimization of numerical codes," in *Proc. Int. Conf. Supercomputing 1999*, pp. 434–443.
[48] E. Myers, "A precise inter-procedural data flow algorithm," presented at the 8th Annu. ACM Symp. Principles of Programming Languages, Williamsburg, VA, 1981.
[49] R. Neelamani, J. Rombery, H. Choi, R. Riedi, and R. Baraniuk, "Multiscale image segmentation using joint texture and shape analysis," presented at the Wavelet Applications in Signal and Image Processing, San Diego, CA, 2000.
[50] "The Common Object Request Broker: Architecture and specification version 2.0," Object Management Group, Needham, MA, 1997.
[51] W. Pugh and T. Shpeisman, "SIPR: A new framework for generating efficient code for sparse matrix computations," in *Proc. 11th Int. Workshop Languages and Compilers for Parallel Computing (LCPC)*, 1999, pp. 213–229.
[52] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao, "Preliminary results from a parallel MATLAB compiler," in *Proc. Int. Parallel Processing Symp.*, 1998, pp. 81–87.
[53] S. Richardson and M. Ganapathi, "Interprocedural optimization: Experimental results," *Softw. Pract. Exper.*, vol. 19, no. 2, pp. 149–169, Feb. 1989.
[54] L. De Rose and D. Padua, "Techniques for the translation of MATLAB programs into Fortran 90," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 2, pp. 286–323, Mar. 1999.
[55] S. Seres and M. Spivey, "Higher-order transformation of logic programs," in *Proc. 10th Int. Workshop, Logic Based Program Systhesis and Transformation*, 2000, pp. 57–68.
[56] R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*. New York: Wiley, 1997.
[57] G. Sittampalam and O. de Moor, "Higher-order pattern matching for automatically applying fusion transformations," in *Proc. 2nd Symp. Programs as Data Objects (PADO-II)*, 2001, pp. 218–237.
[58] J. Stichnoth and T. Gross, "Code composition as an implementation language for compilers," in *Proc. Conf. Domain-Specific Languages*, 1997, pp. 119–131.
[59] P. F. Thall, J. J. Lee, C. H. Tseng, and E. H. Estey, "Accrual strategies for phase I trials with delayed patient outcome," *Stat. Med.*, vol. 18, pp. 1155–1169, 1999.
[60] P. F. Thall and K. E. Russell, "A strategy for dose-finding and safety monitoring based on efficacy and adverse outcomes in phase I/II clinical trials," *Biometrics*, vol. 54, no. 1, pp. 251–264, Mar. 1998.
[61] P. F. Thall and H. G. Sung, "Some extensions and applications of a Bayesian strategy for monitoring multiple outcomes in clinical trials," *Stat. Med.*, vol. 17, pp. 1563–1580, 1998.
[62] M. Tofte and L. Birkedal, "A region inference algorithm," *Trans. Program. Lang. Syst.*, vol. 20, no. 4, pp. 734–767, Jul. 1998.
[63] M. Tofte and J.-P. Talpin, "Region-based memory management," *Inf. Comput.*, vol. 132, no. 2, pp. 109–176, 1997.
[64] R. Triolet, "Interprocedural analysis for program restructuring with Parafrase," Dept. Comput. Sci., Univ. Illinois, Urbana-Champaign, CSRD Rep. 538, 1985.

[65] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.
[66] T. Veldhuizen, "Expression templates," *C++ Rep.*, vol. 7, no. 2, pp. 21–31, Jun. 1995.
[67] G. E. Weaver, K. S. McKinley, and C. C. Weems, "Score: A compiler representation for heterogeneous systems," presented at the 1996 Heterogeneous Computing Workshop, Honolulu, HI.
[68] C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," presented at the Int. Conf. Supercomputing'98, Orlando, FL.
[69] S. Wolfram, *The Mathematica Book*. Cambridge, U.K.: Cambridge Univ. Press, 1999.

**Ken Kennedy** (Fellow, IEEE) received the B.A. (*summa cum laude*) and M.S. degrees in mathematics and the Ph.D. degree in computer science from Rice University, Houston, TX, in 1967, 1969, and 1971, respectively.

He is the John and Ann Doerr University Professor of Computer Science and Director of the Center for High Performance Software Research (HiPerSoft) at Rice University, Houston, TX. He has supervised 36 Ph.D. dissertations and published two books and over 190 technical articles on compilers and programming support software for high-performance computer systems.

Prof. Kennedy is a Fellow of the Association for Computing Machinery (1995) and the American Association for the Advancement of Science (1994). He received the 1995 W. Wallace McDowell Award, the highest research award of the IEEE Computer Society, in recognition of his contributions to software for high-performance computation. In 1999, he was named the third recipient of the ACM SIGPLAN Programming Languages Achievement Award. He was elected to the National Academy of Engineering in 1990.

**Bradley Broom** (Member, IEEE) received the B.Sc. (with first-class honors) and Ph.D. degrees in computer science degree from the University of Queensland, Brisbane, Australia, in 1983 and 1988, respectively.

He is an Associate Professor in the Department of Biostatistics and Applied Mathematics at the University of Texas M. D. Anderson Cancer Center, Houston, TX. He is also the Associate Director of the Gulf Coast Center for Computational Cancer Research, a joint initiative between M. D. Anderson and Rice University, Houston, for promoting research into novel computational methods in cancer research.

**Arun Chauhan** received the B.Tech. degree in electrical engineering and the M.Tech. degree in computer science from the Indian Institute of Technology (IIT), New Delhi, in 1991 and 1993, respectively, and the Ph.D. degree in computer science from Rice University, Houston, TX, in 2003.

Between his M.S and Ph.D. degrees, he worked with HCL-Hewlett Packard Ltd., India, in the Technical Consultancy Group and as a Senior Scientific Officer at IIT on a project to parallelize a medium-range weather-forecasting model. He is currently a Visiting Assistant Professor at Indiana University, Bloomington. His research interests are in compilers, high-level programming systems, high-performance computing, and grid computing.

Dr. Chauhan is a Professional Member of the Association for Computing Machinery.

**Robert J. Fowler** received the A.B. degree in physics from Harvard University, Cambridge, MA in 1971 and the M.S. and Ph.D. degrees from the University of Washington, Seattle, in 1981 and 1985, respectively.

He is a Senior Research Scientist and Associate Director of the Center for High Performance Software Research at Rice University. His research interests are in the area of high-performance distributed and parallel computing. Specific interests include compilers and programming environments, architectures, operating systems, performance evaluation, and simulation.

**John Garvin** received the B.S. degree from Yale University, New Haven, CT, in 2001 and the M.S. degree from Rice University, Houston, TX, in 2004. He is currently working toward the Ph.D. degree in the Department of Computer Science at Rice University.

His research interests include compilation of high-level programming languages, high-performance compiler optimizations for scientific computation, and performance issues in biostatistics.

**Charles Koelbel** received the B.A. degree from Augustana College, Rock Island, IL, in 1983 and the M.S. and Ph.D. degrees from Purdue University, West Lafayette, IN, in 1985 and 1990, respectively.

From 1998 to 2001, he served as a Program Director at the National Science Foundation, where he helped coordinate the Information Technology Research program. He is currently a Research Scientist in the Computer Science Department at Rice University, Houston, TX. He has contributed to many research projects while at Rice, including the High Performance FORTRAN Forum. He just completed serving on the National Academies of Science committee studying the Future of Supercomputing. He is coauthor of *The High Performance Fortran Handbook* (Cambridge, MA: MIT Press, 1993) and many papers and technical reports.

**Cheryl McCosh** received the B.S. degree in mathematics from the University of North Carolina, Chapel Hill, in 2000 and the M.S. degree in computer science from Rice University, Houston, TX, in 2003. She is currently working toward the Ph.D. degree in computer science at Rice University under the direction of Prof. K. Kennedy.

Her current research interests include high-performance computing, component integration, and domain-specific languages.

**John Mellor-Crummey** received the B.S.E. degree *magna cum laude* in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1984 and the M.S. and Ph.D. degrees in computer science from the University of Rochester, Rochester, NY, in 1986 and 1989, respectively.

In 1989, he joined Rice University, Houston, TX, where he holds the rank of Associate Professor and Senior Faculty Fellow in both the Department of Computer Science and the Department of Electrical and Computer Engineering. Since 2002, he has been Deputy Director of the Center for High Performance Software Research. His research focuses on compiler, tool and runtime library support for high-performance computing.

Dr. Mellor-Crummey is a Member of the Association for Computing Machinery, the IEEE Computer Society, Tau Beta Pi, and Phi Beta Kappa.