# Telescoping Languages: A Compiler Strategy for Implementation of High-Level Domain-Specific Programming Systems

Ken Kennedy
Center for High Performance Software
Rice University
ken@rice.edu

## Abstract

*As both machines and programs have become more complex, the programming process has become correspondingly more labor-intensive. This has created a* software gap *between the need for new software and the aggregate capacity of the current workforce to produce it. This problem has been compounded by the slow growth of programming productivity over the past two decades. One way to bridge this gap is to make it possible for end users to develop programs in high-level domain-specific programming systems. The principal impediment to the success of these systems in the past has been the poor performance of the resulting applications. To address this problem, we are developing a new compiler technology that supports script-based* telescoping languages*, which can be built from base languages and domain-specific libraries. By exhaustively compiling the libraries in advance, we can ensure that the performance and portability of the applications produced by such systems are high, while the compile times for scripts are acceptable to the end user. These qualities are essential if script-based systems are to be practical for development of production applications.*

## Introduction

As high performance computing platforms have become more complex and performance has become more sensitive to programming style, application developers have been burdened with an increasingly complex programming task. This has severely limited the productivity of scientific programmers and inhibited the rate at which high-end computing systems are accepted by the scientific rank and file. The productivity of application developers would be greatly enhanced if they could use high-level programming systems to rapidly develop prototype and production applications in languages based on the standard notations of their problem domains.

It should be observed that many high-level problem solving environments exist today. Matlab [11] and Mathematica [21] are standard tools for matrix and symbolic analysis. Octave [9] is a high level language originally designed for chemical reactor design. EllPack [12] is a language for describing and solving elliptic partial differential equations. The problem is that most of these do not achieve acceptable performance for computation-intensive scientific problems. Therefore, if we are to make high-level problem-solving systems effective for building production applications, the performance of the resulting object code must be high enough to obviate the need for recoding them in more conventional programming languages such as Fortran, C, and C++.

To address the performance problem, we are conducting research on compiler technology and library design to support construction of practical domain-specific development environments for high-performance applications. Our approach is to build these environments from domain-specific libraries that are invoked from flexible scripting languages. This is similar to the way such systems are constructed today. However, because existing scripting languages are interpreted and they treat libraries as black boxes, these systems fail to achieve acceptable performance levels for compute-intensive applications. Previous research has shown that this problem can be ameliorated by translating the script to a conventional programming language and using whole-program analysis and optimization to improve performance [8, 4]. Alternatively, some specialization can be built into the library instantiation process, if the library is written in a language that uses some form of dynamic macro expansion. This is the approach used to optimize the POOMA library [2].

Unfortunately, these approaches suffer from the disadvantage that program compilation times, either for scripts or expansion of data structures, can be long. To avoid this, we have developed a new approach called *telescoping languages*, in which the libraries that provide component operations accessible from scripts are extensively precompiled

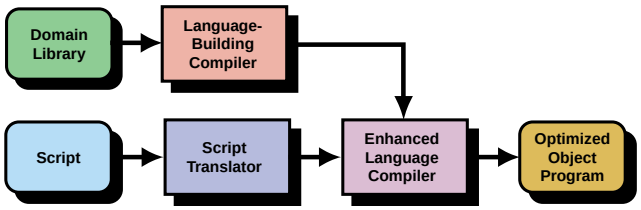


**Figure 1. Telescoping Languages**

and optimized to permit fast generation of efficient code and local optimization of scripts when they are processed. The goal of precompilation is to specialize different versions of each library routine for sets of conditions that may hold when the routine is invoked. The entire set of specialized routines is collected in a database that permits efficient code selection and inlining when the scripts are compiled.

Because large-scale applications in the future are likely to be executed on computational "grids"—collections of interconnected heterogeneous computing platforms—a secondary goal of our research is to support programming for grid environments. Domain-specific problem-solving environments are attractive for this purpose because language processors for problem solving environments can exploit domain knowledge to tailor the implementation to the execution environment. To support this goal, we must address the problem of *portability* of performance across different architectures and hardware configurations—in other words, a useful high-level programming system cannot afford to achieve high performance at the expense of portability.

To provide high degrees of portability, we will generate *self-tuning* applications by designing and precompiling libraries to produce alternative implementations tailored to different target platforms. Specific implementations can then be selected and integrated in a dynamic compilation pass just prior to execution, once the actual computing environment is known. In addition, library code could reconfigure itself dynamically for better performance once the size and shape of the data are known. This strategy has been adopted by the GrADS project [15, 14], which is building a framework for Grid application development and execution.

The unifying theme in these approaches is to reduce script compilation time and improve portability by investing extra time in library compilation, on the assumption that the domain libraries will be recompiled far less often than scripts that invoke them. Under this assumption, it is possible to use extremely powerful optimizations that exhaustively explore the implementation space for library modules. Such optimizations might take hours, but they would be worth the cost if the libraries are reused in many different scripts.

## Telescoping Languages

The basic idea behind *telescoping languages* is to support the construction of high-level problem-solving languages based on sophisticated domain-specific libraries connected via scripting languages. This is similar to the way such systems are constructed today. However, because scripting languages are usually interpreted and the libraries are treated as black boxes, performance falls short of its potential. Telescoping languages will address this performance shortfall by exploiting powerful interprocedural compilation strategies developed over the past two decades.

One way to do this is to translate scripts and libraries to the same intermediate representation then compile the entire collection into highly efficient code using an whole-program optimizing compiler [4, 7, 6]. The effectiveness of this approach was established for Matlab programs by De Rose and Padua [8]. However, it suffers from the disadvantage of excessively long compilation times because the whole program, including all invoked libraries, must be compiled each time a new script is processed. A user is likely to be surprised if a short script compiles for a very long time. Although this might be ameliorated by optimizing only when the application is in production, it has proved difficult in the past to maintain the distinction between development and production.

To deal with these issues, the telescoping-languages strategy, depicted in Figure 1, shifts the cost of compilation from script compilation time to library compilation time. The domain-specific library is provided as input to a specialized version of the base language compiler that produces an enhanced base-language compiler in which library entry points and their execution properties are known as language primitives. In essence, this defines a new language consisting of the original base language plus the domain library primitives. This strategy can be applied iteratively to several different levels of libraries—hence the term "telescoping languages." Once the extended compiler is available, scripts can be translated into calls to the extended library and compiled to efficient object code.

A key to success of the telescoping languages approach is to compile libraries into a format that permits efficient local optimization of scripts when the scripts are themselves processed. Because the library compilation step will not happen very often, it is practical to expend enormous amounts of compile time to ensure that the script compilation times can be kept short without sacrificing performance of the object code. This approach goes far beyond the conventional use of precompiled headers because the library *implementations* are precompiled and optimized, so that the locally optimal implementation for a given invocation can be quickly selected at script compilation time based on knowledge of the types and values of parameters gained from data flow analysis.

The cost of this strategy is an extensive, and time-consuming, compilation and optimization process for the libraries. A fundamental aspect of this compilation is *reverse program analysis*, which reasons back from points of potential optimization to library interfaces, with the goal of specializing different versions of the code to different conditions that may apply when the script invokes the library. These versions are saved in a database that permits efficient code selection and inline substitution when the scripts are compiled. Thus, the script can be translated into efficient code based solely on information that is local to the script.

In the case of a distributed application, this optimization capability is of great importance. If the script program is making a large number of invocations to a library object on a remote host, it may be possible to transform the script to use a single, block-request interface on the library. This can only happen if the library can be compiled so that it implements a mutable set of interfaces and this fact is known to the script transformation engine.

Three types of compilation technologies are critical to making this strategy successful:

- *Reverse program analysis*, which identifies promising optimization points in the library and reasons backward from those points to the preconditions at the call that are needed to make the optimizations feasible.

- *Construction of jump functions* that can quickly determine the side-effects to parameters passed from the script to the library, making it possible to instantly propagate parameter properties (e.g., type and value) through procedure calls at script compilation time.

- *Recognition and exploitation of identities* involving the library interface that specify when it is safe to replace composite library calls with efficient specialized code at script compilation time.

It should be noted that some identities will be provided by the library developer based on domain knowledge—these identities should be included in the optimization table for use along with those that are constructed from examination of the library.

The telescoping languages approach has some other major advantages over traditional ways of dealing with libraries. Three of them are worth mentioning here:

- *Reliability*. Because the telescoping languages script compiler is effectively an interprocedural compiler, it can check that a script adheres to all of the conventions for calling library components. In addition, the compiler can check special correctness conditions defined by the library designer at compile time or generate code to check them at run time if required. Finally, since the script deals in macro operations, the programmer will not be tempted to insert the kind of shortcuts that are typical of hand-produced code. These shortcuts, which cause so many problems in normal programming, are instead taken automatically by the compiler when it can determine that doing so is correct. Thus, once the library is free of bugs, we would expect high reliability in scripts that use it.

- *Client-server Communications.* It is common in distributed computing environments to have a standard library interface to broker all invocations of remote services. In many cases, however, a black-box approach to this problem is not the most efficient strategy. Often a library invocation for remote services consists of two components: code that translates the specific call into a general remote services invocation and the remote service invocation itself. The interface-specific portion of the code typically resides on the requesting processor and can often be optimized in the context of the point of call. For example, if remote data lookups are invoked on a series of keywords, the keywords could be saved in an array and a request for lookup for all of them could be submitted in a single remote invocation—effecting a database analog of the classical reduction-in-strength optimization [1]. This coalesced remote request is likely to perform much better in most distributed environments.

- *Protection of Source Code.* The telescoping-languages strategy also answers another common question, namely: How can proprietary library source code be protected? A library developer should have the option of declaring that particularly sensitive portions of the code are not to be inlined, thus preventing the source for those pieces of code from being examined by the end user. However, by making these library sources available to the language-building compiler, specialized versions of these library routines can be constructed as necessary to exploit optimization opportunities for configurations of conditions that may arise. Other parts of the code could then be freely inlined to enhance performance.

It should be clear to the reader that if the technology is feasible, telescoping languages provide an attractive way to extend existing languages and build problem-solving systems. The next few subsections explore the issue of technical feasibility.

## Reverse Program Analysis

The goal of reverse program analysis is to identify the points where profitable optimizations might be applied if there were complete information about parameters were

available and to propagate information about desirable values backward from those points to procedure interfaces. In the case of a complete library, this propagation must proceed across the boundaries of inter-library interfaces where one library routine might call another. There are five significant components to this analysis:

- *Interface Analysis.* First the library structure must be analyzed to determine which interfaces are external and which are internal—i.e., may be called from within. The definition of external interfaces may be part of an annotation by the library designer and builder, or it may be evident from the structure of the library and language considerations.
- *Forward Parameter Analysis.* Next the analysis must forward propagate information about which values of parameters are involved in calculations of inputs to each statement in the library routine. Again this must take place across procedure boundaries.
- *Optimization Analysis.* The code in each library routine is analyzed to determine which optimizations are possible and which desirable optimizations are precluded but might be applicable if certain inputs that are affected by parameter values are in certain ranges. Breaking conditions in dependence analysis, which are predicates attached to dependences that specify when the dependence can be safely ignored [10], are the kinds of conditions that might be determined.
- *Reverse Condition Propagation.* The optimization conditions are propagated backward in an attempt to develop conditions at a library call interface that may yield significant simplifications.
- *Code specialization.* When all optimization conditions are known, a set of specialized versions of the code are generated and saved for each set of specialized conditions. In some cases, sets of conditions may be combined to reduce the number of alternative implementations.

Perhaps the best way to illustrate this idea is by an example. Consider the following simple Fortran routine:

```
SUBROUTINE VMP(C, A, B, M, N, S)
   REAL A(N), B(N), C(M), S
   I = 1
   DO J = 1, N
      C(I) = C(I) + A(J)*B(J)
      I = I + S
   ENDDO
END
```

In this code, which is modeled after code from LAPACK, the product of arrays A and B is added to array C. The problem with this example is that the stride variable S might be 0, in which case the statement is not vectorizable, and must be replaced by a reduction.

When the process above is applied to this routine, the interface analysis discovers that the increment to the variable I is a function of the input S. When induction variable substitution is carried out during Optimization Analysis, the routine becomes

```
SUBROUTINE VMP(C, A, B, M, N, S)
   REAL A(N), B(N), C(M), S
   DO J = 1, N
      C(S*J-S+1) = C(S*J-S+1) &
         + A(J)*B(J)
   ENDDO
END
```

The DO loop is vectorizable if $S \neq 0$. When this condition is propagated back to the interface and code is specialized, we would get two versions of the code. For $S \neq 0$, we have:

```
SUBROUTINE VMP(C, A, B, M, N, S)
   REAL A(N), B(N), C(M), S
   C(1:S*N-S+1:S) = C(1:S*N-S+1:S) &
      + A(1:N)*B(1:N)
END
```

But for the case where $S = 0$, we get:

```
SUBROUTINE VMP(C, A, B, M, N, S)
   REAL A(N), B(N), C(M), S
   C(1) = SUM(A(1:N)*B(1:N))
END
```

These two variants would be saved in a code selection database for the library in a form suitable for in-line expansion.

## Jump Function Construction

If the compiler is to quickly and accurately determine the properties of variables passed to library components by the script, it must be able to instantaneously determine the effect on those properties of other library calls that are on a path from the start of the script to the call being analyzed. *Jump functions* provide a way to do this. A jump function for procedure *P* summarizes the value of an output parameter from *P* in terms of the values of input parameters. In those cases where no precise summary can be computed, the jump function will return a special value indicating that the output parameter is undefined. Where jump functions are well defined, they can make it possible to propagate properties of variables through a call site without redoing extensive analysis of the source of the called program. Thus, if telescoping languages are to be an effective strategy, the library compilation phase must compute jump functions for every public interface.

Jump functions have been a subject of substantial research on interprocedural analysis [3]. Within the library,

they can be constructed by composition if partial jump functions are computed to determine the translation of parameters from a procedure entry to other parameters at call sites within each procedure. In the case of libraries within the telescoping languages framework, the challenge is to compute jump functions for specialized versions of the procedures that may be selected due to the values of parameters at scripts. This may also require some iteration. A general jump function may be used to determine that certain parameters are always constant at a given call site. These constant parameters may in turn be used to select a specialized version of the entry point with simpler jump functions, which may allow selection of an even more specialized version of the code.

As an example consider the following simple procedure

```
 PROGRAM MAIN
   V = 1
   N = 0
   DO WHILE(V>0)
      V = FUNC(V)
      N = N + 1
   ENDDO
   PRINT N
END MAIN

REAL FUNCTION FUNC(X) ! Part of library
   IF (X>0) THEN
      RETURN (X-1)
   ELSE
      RETURN X
   ENDIF
END FUNC
```

Compilation of the routine FUNC in the library will produce three implementations in the code database:

```
! X known > 0
   inline(X-1)

! X known <= 0
   inline(X)

! X unknown
   inline( (X>0 ? X-1 : X) )
```

Note that in each case the implementation of the routine is to be inlined. Note also that the C conditional expression syntax is used to denote a conditional expression because Fortran has no corresponding expression.

In each of these cases, the jump function will be the same as the body of the inline directive. At code generation time the compiler will perform the following analysis steps:

1. First V=1 will be propagated to the FUNC call. Note that this is not the final value because the compiler must determine whether the loop back produces a different constant value for V.
2. The general jump function (third in the list above) is invoked and produces the constant value 0 for V.
3. In propagating that value around the loop, the compiler discovers that the loop condition will cause the loop to be exited. Thus it concludes that only one iteration of the loop is performed and the constant value 1 always reaches the function.
4. The compiler now selects the refined jump function for the function and eliminates the loop to produce the code below.

```
PROGRAM MAIN
  V = 1
  N = 0
! DO eliminated
     V = V - 1
     N = N + 1
! ENDDO eliminated
  PRINT N
END MAIN
```

which would simplify to

```
PROGRAM MAIN
  PRINT 1
END MAIN
```

Although this is clearly an oversimplified example, it should give some indication of the power of the approach. The main point is that practically no time is spent during script compilation on analysis of the function itself, since that analysis is done in advance.

## Interface Identities

The third key property of the telescoping languages strategy is the recognition of identities involving different interfaces. This recognition can be performed automatically or through the use of annotations provided by the library developer. An example of such an identity comes from a stack manipulation package. In Fortran, this package might look like the following:

```
MODULE STACK
   PRIVATE, ALLOCATABLE, &
      INTEGER :: VALS(:)
   PRIVATE, INTEGER :: SIZE, TOP

CONTAINS
   SUBROUTINE INIT(N)
      INTEGER N
      SIZE = N
      ALLOCATE ( VALS(SIZE) )
      TOP = 0
   END INIT
```

```
   SUBROUTINE PUSH(X)
      INTEGER X
      IF (TOP == N) THEN
         CALL OVFLO(X)
      ELSE
         TOP = TOP + 1
         VALS(TOP) = X
      ENDIF
   END PUSH

   INTEGER FUNCTION POP()
      IF(TOP > 0) THEN
         TOP = TOP - 1
         RETURN ( VALS(TOP+1) )
      ELSE
         CALL EMPTY
      ENDIF
   END POP
   . . .
END MODULE STACK
```

The library designer, and in some cases the compiler, may wish to observe that a push followed immediately by a pop returns the value pushed (except in the case of overflow, but overflow may not need to be preserved).

Encoding these identities in libraries, when combined with powerful common subexpression elimination and value propagation phases can lead to substantive simplifications of the code.

## Script Compilation

The general script compilation procedure for the telescoping languages strategy would fall into 3 phases:

1. First the propagation of variable properties (e.g., value and type) would be analyzed with the help of jump functions. This would provide a "most precise estimate" of the properties of variables that are passed as parameters to each of the routines in the library. This phase would also recognize sequences of library invocations that formed identities and replace them with the appropriate specialized code.

2. At each invocation point for the library, the collection of input variable properties would be used to select the code to be substituted for the call in a process similar to *unification* from theorem proving [18]. This permits the selection of the most specialized code given the set of parameter properties. The database would be organized to facilitate this process.

3. Finally, the selected code would be substituted inline for the library invocation.

Note that through the use of the *noinline* directive in the substituted code, the library designer can preserve procedure invocations where there is no point in performing inlining, the code needs to be kept private, or the invocation is a remote procedure call to a server process on another machine.

## Portable Performance

An essential property of a useful script-based application development system is that it be possible to port the applications developed in the system to a variety of target platforms with the expectation of reasonable performance. The problem is that the final target of a script-based application may not become known until script compilation time—long after the libraries that form the basis for the telescoping language are precompiled. In a naive system, this would mean that the entire application, including the library code, would need to be optimized for a specific target platform just prior to execution time. Once again, long compilation times might make this approach impractical.

Telescoping languages provide an opportunity to overcome this problem. Because library compilation times can be long, it is possible to generate highly-optimized variants of every implementation in the database for each of a potential collection of target machines. Because the library precompilation process is primarily a high-level interprocedural phase to be followed by a low-level compilation for the target machine, in many cases the same high-level version will be effective for each of the potential target systems. Given that, all that is required is to compile that version for each potential machine using that machine's low-level compiler. Exceptions will occur in cases where high-level transformations, such as blocking for cache, must be tuned to a specific machine. In those cases, the specialized implementation is tuned during the exhaustive compilation phase for each of the target machines and then passed to the low-level compiler.

Once preoptimized and compiled versions of each database routine are available, the correct implementation can be selected at link time. In fact, there is no requirement that the same target machine be used for every invocation—in the GrADS project [15], the link phase is replaced by a dynamic optimization phase that distributes work among a heterogeneous collection of processors and tailors the code and communication to each target machine in the collection.

To summarize, the process of preparing a library for use in a system based on telescoping languages is as follows:

- Using the compilation process described earlier, compute specialized implementations, along with associated jump functions, for each public interface in the library. These are typically machine-independent.

- For each potential target machine, specialize the code for that target machine, if necessary, and invoke the

target machine's compiler to produce a linkable object module for each target.

At script compile time, the specialized implementations are chosen for each library call for the target machine to which that call has been assigned.

The work on optimizing for specific machines builds on previous efforts that use extensive preprocessing to produce near-optimal code for specific machines. These efforts include the CMSSL library [13], Atlas at the University of Tennessee and Oak Ridge [20], and UHFFT at the University of Houston [16]. All these efforts used precompiled and optimized kernels that are selected dynamically.

Another obstacle to achieving high performance is the fact that the shape and size of many of the application data structures will not be known until run time. This is particularly true in applications that use adaptive or dynamic data structures, which are increasingly common in serious scientific applications. Within the telescoping languages framework, this problem can be addressed, at the cost of initial code space, by including alternative implementations in the object code for the application and selecting the correct one dynamically at run time in a preprocessing step that takes place right after the parameters in question are known. This strategy is similar to the dynamic compilation strategies in Java [19] and the inspector-executor method for compiling irregular applications for parallel execution [5].

Although these strategies are not unique to telescoping languages, the framework makes it possible to automate many of the steps in tailoring libraries so that the library developer need only identify shared compute-intensive kernels along with test drivers—leaving the specific optimizations to the library precompilation system.

## Application: User-Defined HPF Distributions

The telescoping languages strategy described in this paper is designed to support high-level programming systems based on libraries that add functionality particular to a problem-solving domain. However, the technology has many other useful applications that demonstrate its advantages over traditional interprocedural compilation systems. Here we will describe an application to the compilation of High Performance Fortran (HPF).

One reason that HPF has not been as successful as its designers envisioned is its lack of flexibility. HPF offers a fixed set of distribution strategies that are suitable for many scientific applications, particularly those that are defined on regular meshes. However, it provides no way for the defined collection of distributions to be extended. This means that there is no way for a user to add support for a specific distribution strategy that is not built in to the language. Many users who would otherwise use HPF, are unable to do so because their problem is defined on an irregular mesh or requires some sort of custom distribution.

The telescoping languages strategy could help overcome this disadvantage if HPF permitted user-defined distributions. In reality, a distribution can be thought of as any library that provides a certain interface, with a required set of entries. For example, a distribution needs to be able to store or load a particular element, it must be able to produce the number of the processor that owns a particular element, and it should be able to load or store all or parts of array columns or rows.

Once a distribution is defined, it could be exhaustively precompiled to be integrated into the base language. It could then be used by the HPF compiler to perform its normal optimizations, invoking (and possibly inlining) calls to the distribution library. Such a strategy would permit a much easier way of experimenting with new distribution mechanisms for sparse matrices and out-of-core arrays.

As an example of an interesting optimization that could be performed with any distribution, consider a loop that iteratively accesses the elements of a single array column:

```
DO I = 1, N
   B(I) = A(I,J) * C
ENDDO
```

If the array `A` is distributed via a distribution library, the naive way to implement the accesses to `A` is to repeatedly call the "load single element" entry point. However, a compiler could recognize that the loop is accessing successive elements in a column of A and call the (presumably faster) entry that provides contiguous portions of a single column. This is a direct analog of the conventional optimization known as *reduction in strength*[1].

Such a mechanism could be used to add two different new distributions that have been proposed since the final version of the HPF standard was adopted:

- *Space-Filling Curves*. A number of researchers have experimented with Hilbert curves—also known as space-filling curves—for distribution of adaptive problems [17]. Space-filling curves provide a natural way to lay out irregular data so that items that are close to one another in physical space are close to one another on the curve. Thus, the Hilbert curve provides a linear layout of the data items of relevance to the problem. This linear representation can be subdivided into equal parts for distribution without generating enormous amounts of communication.

- *Out-of-Core Arrays*. User-defined distributions could also be used to develop support for out-of-core arrays, because disk could be just an extension to local memory from the perspective of a distribution library. In this model, all the transformations used to enhance locality in a problem would translate to minimization of I/O operations in the out-of-core distribution.

## Summary

The telescoping languages strategy provides a new approach to the implementation of high-level, domain-specific problem-solving environments. By expending extensive compilation time on preprocessing the domain specific libraries that provide functionality to the system, it may be possible to use a fast compilation step to produce code from scripts that is efficient enough for production use on scientific problems. The basic idea is to have the library compilation process produce an extensive database of specialized implementations that can be selected at script compilation time. In addition a similar strategy can be used to select from among implementations optimized for different computing platforms, ensuring that good performance is portable across multiple platforms.

**Acknowledgments** Many people have contributed to the development of the ideas described in this paper. I would particularly acknowledge the contributions of Keith Cooper, Jack Dongarra, Dennis Gannon, Rob Fowler, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon, who helped develop the telescoping languages project. In addition, John Reynders' lucid descriptions of the POOMA project deeply influenced the ideas in this effort. Finally, I would like to acknowledge the NSF Center for Research on Parallel Computation, the Los Alamos Computer Science Institute, supported by the Department of Energy ASCI program, and the GrADS Project supported by NSF Next Generation Software Program, under the program management of Frederica Darema, for the support they provided and for their influence on the vision behind this effort.

## References

[1] F. E. Allen, J. Cocke, and K. Kennedy. Reduction of operator strength. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Application*, pages 79–101. Prentice-Hall, New Jersey, 1981.

[2] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. In *Proceedings of Supercomputing 95*, San Diego, CA, Dec. 1995.

[3] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[4] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S. K. Warren. A practical environment for Fortran programming. *IEEE Computer*, 20(11):75–89, Nov. 1987.

[5] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. Software support for irregular and loosely synchronous problems. *International Journal of Computing Systems in Engineering*, 3(2):43–52, 1993.

[6] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, Seattle, WA, June 1985.

[7] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, Feb. 1993.

[8] L. DeRose and D. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of the 10th International Conference on Supercomputing*, May 1996.

[9] J. Eaton. Octave. http://www.che.wisc.edu/octave, 1999.

[10] G. Goff. *Practical Techniques to Augment Dependence Analysis in the Presence of Symbolic Terms*. PhD thesis, Department of Computer Science, Rice University, 1997.

[11] B. Hahn. *Essential MATLAB for Scientists and Engineers*. Arnold, 1997.

[12] E. N. Houstis and J. Rice. The engineering of modern interfaces for pde solvers. In E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, *Symbolic Computation: Applications to Scientific Computing*, pages 89–94. North-Holland, 1992.

[13] L. Johnssson et al. CMSSL: A scalable scientific software library. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1994.

[14] K. Kennedy. Compilers, Languages, and Libraries. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 181–204. Morgan Kaufmann, 1998.

[15] K. Kennedy et al. The GrADS Project. http://nhse2.cs.rice.edu/grads, 1999.

[16] R. Mahasoom. An adaptive software library for fast fourier transforms. Master's thesis, Department of Computer Science, University of Houston, Houston, TX, December 1999.

[17] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski. A common computational infrastructure for adaptive algorithms for pde solutions. In *Proceedings of Supercomputing '97*, 1997.

[18] M. S. Patterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[19] Sun Microsystems Inc. Java HOTSPOT performance engine architecture: A white paper about Sun's second generation performance technology, Apr. 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[20] C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC '98*, Orlando, FL, November 1998. IEEE Publications.

[21] S. Wolfram. *The Mathematica Book*. Cambridge Univ. Press, 1999.