# Prospective Procedure Cloning with Linear Algebra Routines

William Marshall Hilands
B.S.C.S., Portland State University 1990
B.A. Accounting, Portland State University, 1990
B.A. History, Princeton University, 1978

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

April 1992

The thesis "Prospective Procedure Cloning with Linear Algebra Routines" by William Marshall Hilands has been examined and approved by the following Examination Committee:

[illegible]

Michael Wolfe
Associate Professor
Thesis Research Adviser

Robert Babb II
Associate Professor

Steve W. Otto
Assistant Professor

# Contents

# List of Figures

# Abstract

## Prospective Procedure Cloning with Linear Algebra Routines

**William Marshall Hilands, M.S.**
**Oregon Graduate Institute of Science & Technology, 1992**

**Supervising Professor: Michael Wolfe**

With procedure cloning, the linker tailors a routine to suit the call site. With *prospective* procedure cloning, the compiler identifies routine characteristics (without knowledge of call sites) so that the linker can make effective and efficient optimization choices. Conventional language tools take compiler-generated objects into an executable program image directly. ParaScope, a programming environment, uses interprocedural methods to generate a more optimal executable. With a proposal for prospective procedure cloning, this project examines the costs and benefits of ParaScope usage on Fortran linear algebra routines. We investigate the requirements of conventional language tools and of ParaScope. These experiments use the Basic Linear Algebra Subroutines (BLAS) routines and the LINPACK linear algebra subroutine package.

# Chapter 1

# Introduction

With procedure cloning, the linker tailors a routine to suit the call site. With *prospective* procedure cloning, the compiler identifies routine characteristics (without knowledge of call sites) so that the linker can make effective and efficient optimization choices. The compiler analyzes procedures to determine whether parameters can be used to create alternative implementations that will execute more efficiently. The linker uses the actual parameters to generate a procedure version. Also, prospective procedure cloning makes possible reduction of program code size, resulting in lower memory demands at execution time. Constant parameters, stride, and array size provide opportunities to improve efficiency. This project uses the ParaScope programming environment to evaluate how linear algebra routines respond to prospective procedure cloning. Additionally, the time and space required by ParaScope are compared to those required by conventional language tools.

Prospective procedure cloning's major opportunity examined in this thesis comes in serial code. As parallel operations are set up to execute more efficiently, the relative importance of serial code increases. As more parameter checking and link-time constant conditional expressions occur, there are more opportunities to improve that serial code.

## 1.1 Background

Typically, a source file compiles into an object module with some relocation information. A linker then combines these modules into an executable image. This approach was satisfactory in a slow single processor environment with expensive memory. Separate compilation speeds program development; only modified files need to be recompiled so that the link can create an executable program.

Parallel and vector processors now offer increased capabilities that demand better tools in a world of inexpensive storage. Taking advantage of today's supercomputing capability requires use of more complex tools. A change in method is a prerequisite to improving support.

## 1.2 Motivation

There is consensus that the traditional compilation scheme needs improvement [KDLS86]. ParaScope is one tool that supports a new approach [CCH+87]. This programming environment offers many possibilities by maintaining more information – interprocedural data – that can be used to make optimization choices. Fortran is the traditional language for scientific computation and is the language handled by ParaScope. A module editor provides for program entry and does module analysis. The program compiler performs the linking function; in addition, it achieves even more through interprocedural optimization. Initial phases of ParaScope determine interprocedural data for later phases. Thus, the programming environment maintains many advantages of separate compilation so that a program's entire text need not be processed at one time; indeed, ParaScope's design attempts to apply its additional efforts so that the extra time required is not noticeable. If analysis occurs once in the module compiler, it need never be a concern of the program compiler. Avoiding redundant effort – always a good idea

– is especially desirable at link time when the entire program comes together.

## 1.3 Linear Algebra Routines

The linear algebra routines in the BLAS (basic linear algebra subroutines) and LINPACK linear algebra subroutine package are used to evaluate the optimizations examined here. Linear algebra is only part of a large body of mathematical software that is the subject of much current optimization study. "[S]ince scientific calculations are extremely computation-intensive, efficiency of the code generated by the compiler is critical to the Fortran programmer" [CCH+87]. Human factors such as turnaround time are often more important criteria than execution efficiency; however, sophisticated linkage analysis may determine that interprocedural optimization results in less total effort. Because mathematical kernel routines are called a great many times in engineering applications, any improvement in execution efficiency will accumulate. Procedure development can be simplified if *intraprocedural* issues receive automatic handling because interprocedural information is available. Interprocedural optimization thus permits the programmer to focus on human factors, letting language tools do more work.

## 1.4 Interprocedural Optimization

Procedure or function inlining is an obvious form of interprocedural optimization. However, using inlining, code size can grow exponentially as can the demands on the optimizer. It is difficult to tell how sophisticated a scheme to choose for inlining. A simple, naive approach can result in unproductive work.

Mary Wolcott Hall's dissertation *Managing Interprocedural Optimization* investigates interprocedural compilation [Hal91]. After examining benchmarks using inlining and cloning separately, she moves to a goal-directed strategy; this strategy "clone[s] to expose additional interprocedural constants" so the program compiler can "[i]nline to expose loop nests". In one benchmark, execution speed doubles because of efficient memory access.

This thesis exploits simpler situations that provide modest savings. Prospective procedure cloning (outlined in Section 2.3.4) may not produce any improvements; however, if parameter checking and other conditional expressions use constant values, the suggested optimizations provide significant advantages. With prospective procedure cloning, much of the optimization effort comes during module compilation where the time can be best afforded. Hall's strategy provides a higher payoff, but it applies in narrower circumstances and depends on greater linkage effort.

## 1.5 Goals

Performance – in terms of lessened execution time – is the central goal. The question is whether ParaScope's benefits provide adequate reward for an increased investment in compilation time and memory utilization. Little published evaluation has been done on interprocedural optimization; Conradi did a mainly theoretical treatment in 1983 [Con83]. Performance must provide a compelling reason for the use of a new program development paradigm. This project looks at linear algebra routines using the ParaScope environment to determine the potential of interprocedural optimization, and in particular, of prospective procedure cloning.

## 1.6 Overview

The second chapter provides background on interprocedural optimization and its implementation in the ParaScope programming environment; at the chapter's end, we propose prospective procedure cloning. The third chapter surveys a variety of linear algebra routines. The fourth chapter discusses the different opportunities with serial and parallel execution. The fifth chapter gives the experimental results of prospective procedure cloning. The last chapter summarizes the results and suggests future work.

# Chapter 2

# The Programming Environment

The ParaScope programming environment, a project at Rice University, is the base tool this thesis uses to explore how *interprocedural optimization* improves mathematical software's efficiency. ParaScope supports Fortran, the language traditionally used by scientific programmers. ParaScope's achievements come from development and storage of interprocedural information in an intelligently designed database. The database facilitates efficient updating, recompilation and computation; however, handling the additional information requires greatly increased memory.

This chapter discusses the potential advantages in the executable program image made possible by interprocedural optimizations. At the end of the chapter, we propose prospective procedure cloning as an additional ParaScope capability; description and evaluation of experiments with it will follow in later chapters.

ParaScope is the successor to $R^n$ [CCH+87]. It gains a dependence analysis and a parallel optimization capability [Wol89] that $R^n$ lacked. ParaScope is a huge program. In total, over 100,000 lines of 'C' code make up this Fortran programming environment (before the integration of parallel analysis). The graphical window interface requires more code than a purely textual one, but it adds an important capability. The interprocedural aspects of ParaScope require significant time and result from considerable algorithmic effort. The fifth chapter will present experiments showing the resources required by ParaScope in comparison to those required with more conventional language tools.

ParaScope is difficult to use casually, but moderate expertise provides computational scientists with a powerful tool. In the course of many executions, more efficient operation provides the user with worthwhile savings. A quick discussion of ParaScope's components follows.

An interactive module editor provides *structure* and text editing; it produces an abstract syntax tree. This abstract syntax tree is stored to facilitate efficient access and analysis. At any time, module compilation is possible to verify correctness. If a module is syntactically correct, interprocedural information is generated for all its routines.

The screen-oriented composition editor provides the environment with detailed instructions on what modules to use in creating a program. A new capability adds a textual interface to develop a composition. This editor uses module information to build "a hierarchical description ... a list of modules and other compositions" [CCH+87] that will make up the executable program image. The program compiler knows if the composition presented to it is consistent and complete.

Finally, using interprocedural analysis to guide optimization, the program compiler constructs an executable program image. ParaScope does not itself generate object code; it depends on the host compiler to pull together the generated Fortran source. Execution monitoring is available to aid in debugging the program.

## 2.1 Interprocedural Analysis

Interprocedural analysis augments and connects with data dependence analysis. An intraprocedural scope restricts dependence analysis so that optimizing transformations cannot occur. The availability of procedural summary information makes possible interprocedural analysis and thus improved optimization choices. Thus, more exact analysis of data dependence replaces unnecessarily conservative assumptions. This thesis does not address dependence analysis nor parallel transformations.

Researchers at Rice University, the developers of ParaScope, developed low-cost interprocedural analysis for FORTRAN[CK87a]. Side-effect analysis computes many sets: MOD, USE, DMOD, GMOD, LOCAL, LMOD, IMOD, RMOD, and ALIAS [CK87b]; a time-bound linear in the size of the program is made possible through this intelligent choice of set construction. Constant propagation "is linear in the size of the call graph" [CCKT86] with three reasonable assumptions.

A call multigraph details all the call sites in the program. Hall's dissertation provides the algorithm of choice to build the call multigraph; it works in the presence of procedure-valued parameters. Multigraph construction takes time $O(N + EP)$ "where $N$ and $E$ are the number of nodes and edges in the final call multigraph, respectively" [Hal91] and $P \leq N$ ($P$ is the number of procedure-valued parameters).

With low-cost interprocedural analysis, there is significant potential for comprehensive optimization. Once a complete view of the program is available, the program compiler can determine how to guide program creation.

## 2.2 Interprocedural Optimization

Reidar Conradi surveyed the advantages to be gained by interprocedural optimization [Con83]. In his "Introduction - the Compiler Crisis", Conradi says "to produce good object code, we must at least perform conventional (i.e. intra-procedural) flow analysis, common sub-expression elimination, code motion, global/local register allocation, etc. In addition, *inter-procedural* flow analysis may improve the code both at the calling sites and in the called procedures." To be more precise, flow analysis only provides an *opportunity* for code improvement.

In Conradi's work, primarily a theoretical treatment, he presents "seven applications of inter-procedural optimization":

1. Optimization of the procedure call mechanism,
2. Optimization of the calling context's code,
3. Optimization with complete alias information,
4. Optimization of global data allocation,
5. Optimization across subprogram boundaries,
6. Optimization for data layout,
7. Optimizing transformations of parallel programs.

These applications provide a framework for discussing the potential benefits of interprocedural optimization. Conradi also briefly reviews many projects. He states that such optimizations produce "a potential saving of 8% in code size and 20% in execution time for normal ... programs on the NORD-10 computer [Con83]". In the next few pages, there are brief descriptions of these seven applications.

### 2.2.1 Optimization of the Procedure Call Mechanism

Allen and Cocke describe four types of linkage [AC72]: *open, closed, semi-open* and *semi-closed.* Open linkage is just inline substitution and receives extended discussion in *Managing Interprocedural Optimization* [Hal91]. Closed linkage is that found in standard separate compilation.

Semi-open linkage puts the called procedure into the calling context, with repeated jumps between them. Code size grows more slowly than with inlining, but we find that execution time may increase.

"Semi-closed linkage requires the called procedure to be compiled before the calling procedure. In this way, the compiler can optimize the passing of parameters at the call site... " [Hal91].

### 2.2.2 Optimization of the Calling Routine's Code

Procedure call side-effects limit optimization. Interprocedural information facilitates optimizations across calls. Also, improved register usage becomes possible with coordination between caller and callee.

### 2.2.3 Optimization with Complete Alias Information

When ref-parameters and global variable accesses within procedure bodies are analyzed, optimization occurs with complete alias information. "If we know all possible *aliases* of a procedure's ref-parameters, we may eliminate compile-time 'cascading of kill actions' when such entities are changed" [Con83].

### 2.2.4 Optimization of Global Data Allocation

Frequently used data can be put in globally reserved registers. Life-time analysis of variables permits improved space utilization.

### 2.2.5 Optimization across Subprogram Boundaries

With interprocedural information, the linker can eliminate code that has no effect nor possibility of execution. In many cases, constant parameters are the basis for tailoring routines.

Prospective procedure cloning (described in section 2.3.4) optimizes across subprogram boundaries. The literature contains little discussion of creating multiple versions, each version built to suit sets of call sites having similar characteristics. This project develops an approach that facilitates cloning, an approach discussed more fully at the end of this chapter.

### 2.2.6 Optimization for Data Layout

Analysis of global data access patterns determines optimal space allocation. Composite data objects and local stack-frames are of interest here.

### 2.2.7 Optimizing Transformations of Parallel Programs

Conradi focuses his attention on parallel processes. However, with interprocedural information, there are improved opportunities to manipulate loops. Huson uses procedure inlining to allow parallel transformations [Hus82]; he investigates calls inside loops using the Parafrase environment. With his efforts, Parafrase obtains more information about the loop's actions thus finding more potential parallelism with a speedup of 1.1 to 6 times (an average speedup of 2).

## 2.3 Current Interprocedural Optimization Research

Hall's dissertation *Managing Interprocedural Optimization* is the major recent contribution in this field. Hall first presents an inlining study that proves unfruitful. She then offers a goal-based approach that achieves significant success.

We propose prospective procedure cloning in the last part of this section. In contrast to Hall's improvement of parallel operations, we suggest an approach that improves serial code.

### 2.3.1 Inline Substitution Research

Recently, researchers at Rice University have enhanced ParaScope to do inline substitution. Their experiences provide a base for discussing interprocedural code manipulation. This study [CHT90a] used inlining to restructure eight benchmarks with the resultant code given to five different compilers; program execution times indicate inlining's success or failure.

In *An Experiment with Inline Substitution* [CHT90a], three types of call sites were inlined:

1. "a procedure of fewer than 25 source code lines",
2. "the sole call to a procedure of less than 100 source code lines",
3. "in a loop and invoked a procedure of less than 175 source code lines".

These three heuristics rely mostly on syntactic measures, not on semantic analysis. The inlining choices resulted in elimination of 75% of static calls and 89% of dynamic calls. On average, procedure length increased from 66 lines to 297 lines. Object code growth "averaged less than ten percent" even though average total source lines grew from 1887 to 3101.

Relative to source text size, compile time no more than doubled – except for four programs on one computer. Execution time changes showed "no real trend, either by compiler or by program. Overall, wins outnumber losses. But, the vast majority of the wins fall in the range between the infinitesimal improvement and five percent". The study suggests that register pressure and data interlocks are second-order costs that reduce speedups gained from reduced call overhead.

*The Perils of Interprocedural Knowledge* [CHT90b] documents unexpected behavior when inlining the LINPACK benchmark. "The Fortran 77 standard allows the compiler to assume that no aliasing occurs at call sites" [CHT90b]. However, inlining breaks this assumption so 72% more nops, 76% more data interlocks, 100% more add interlocks, and 45,503% (!) more floating point interlocks cause running time to increase by 8.5%. They then demonstrate that use of dependence analysis can restore the speed lost by inlining.

Another report, *Improving Parallelism After Inline Substitution*, details methods to reap inlining's benefits. "After inlining, some loops exhibited properties inhibiting optimization that would not likely appear in human-generated code. These properties can be categorized in the following way:

1. Unreachable code.
2. Loop-invariant code.
3. Bounds checking.
4. Partial parallelism." [Hal90]

Unreachable code elimination is an obvious step and can result in fewer dependences. Loop-invariant code at times causes problems that disappear when such code is placed outside the loop. Bounds checking often improves when given a larger context to work with. Within this larger context, partial parallelism can occur, and loop distribution takes advantage of this parallelism.

However, only the one set of inlining heuristics receives consideration in *An Experiment with Inline Substitution* [CHT90a]. Different heuristics would likely produce different results. Limited improvement might occur in a particular case with less additional source text and thus less additional compile time. The success of heuristics is a function of the source text and its constructs. The use of the number of source code lines as a predictor of inlining success is too simplistic: one procedure may be all unconditional code and therefore offer no optimization opportunity whereas another procedure could have high customization potential, allowing inlining to result in the selection of only a few statements. An example of customization appears in section 3.3.

The appeal of inlining is obvious, but as the above study suggests, second-order effects can mask or even outweigh the primary savings. The SGBCO routine makes many identical calls to the *level 1* BLAS routines SSCAL and SASUM. In my study of this LINPACK routine, open linkage never improves execution speed with any subset of *level 1* BLAS inlined. Also, semi-open linkage fails to improve execution speed although less additional code occurs, and some optimizations become possible.

## 2.3.2 Goal-Directed Interprocedural Optimization

Hall moves to a goal-directed cloning and inlining approach that looks at the benchmark *matrix300*. Execution speedup comes from cloning a *level 1* BLAS routine, DAXPY. As her study demonstrates, six versions of DAXPY, inlined into different contexts, are necessary to resolve dependence relations and allow "*loop interchange, loop fusion, unroll and jam,* and *scalar replacement*" [Hal91]. Program execution is faster by a factor of two or more. Savings result from "avoid[ing] nearly 100% of the stores and 50% of the loads. In *matrix300*, we save about 216 million stores and nearly as many loads". Also, "multiply-accumulate statements can potentially be executed in parallel" [Hal91]. Inlining and cloning are now the tools to permit other semantically-based optimizations.

Because Hall's dissertation focuses on benchmarks, her use of LINPACK uses just the *level 1* BLAS routines: IDAMAX, DAXPY, DDOT, and DSCAL. Our project examines all the BLAS routines. There are ten *level 1* BLAS routines, twenty-five *level 2* BLAS routines and six *level 3* BLAS routines; their characteristics differ. The next chapter will examine all the BLAS routines.

### 2.3.3 Reasons for Prospective Procedure Cloning

In a sense, prospective procedure cloning *is* Goal-Based Interprocedural Optimization; the technique uses program semantics to achieve particular results. Whereas Hall focuses on improving parallel operations, we work to improve serial code. The changes are always profitable, although often to a limited degree.

First, we review Hall's description of cloning. She states "[c]loning is useful when calls to a procedure can be partitioned into groups, with each group having distinctly different interprocedural information." [Hal91] But what is to be distinctly different? Our proposal is to examine routines at compilation time to determine customization potential as exemplified in section 3.3. In Hall's general strategy, she develops estimates for how many times each procedure is invoked. With this information available, prospective procedure cloning can determine what distinct differences to take advantage of. If a procedure with high customization potential is called many times, cloning that procedure is a good choice. If program size is a concern and there is a large procedure with high customization potential, cloning that procedure can reduce that program size.

"Improved constant propagation has been suggested as the most important effect of inlining [Bal79] [WZ89]" [Hal91]. In our project also, constant propagation is the primary reason for cloning's success. With ParaScope, once the program compiler finishes interprocedural analysis, optimization study begins. Through examinination of linear algebra routines, we determined a strategy for optimization. Early versions of ParaScope create only one version of a procedure; our proposal is to create multiple versions tailored to sets of callsites.

ParaScope has an inlining capability, but some drawbacks to the technique quickly appear. Programmers often find it advantageous to produce a routine that does many related functions; therefore, overhead is high when executing a small operation contained within a large routine. To solve this problem, inlining must be repeated *each* time the involved procedures change; this and other requisite actions are much more complicated than is the case with cloning; therefore, inlining should be used only when it provides greater advantages.

### 2.3.4 Prospective Procedure Cloning

Prospective procedure cloning determines *at compile time* what procedure characteristics can be taken advantage of. Scalar parameters which are invariant in the procedure body are assumed to be constants. Conditional expressions that remain constant given the above assumption are found and reclassified. More easily made choices occur because important decisions within a procedure can be made at link time – not execution time. More rewarding choices result from knowledge of execution frequency so that optimization can focus on time-saving opportunities. In addition, with cloning, there are fewer secondary effects that reduce efficiencies gained. Procedure cloning is a simpler means of interprocedural optimization than inlining; however, the link stage must be capable of using call site information to provide interprocedural context.

In our study, prospective procedure cloning decides whether it is possible to divide a subroutine into two parts: a parameter checking phase and the remaining computation phase. As a module is compiled, summary information is generated by walking the abstract syntax tree; at this time, conditional expressions are reclassified and a parameter checking phase, if any, is noted.

The integrity of the *parameter checking phase* remains intact; instead of many individual code fragments, the code is split off as a whole so that if any exception is detectable at link time, any prior parameter checking will still correctly occur at execution time. For example, if the SGBMV routine (shown in Appendix B) is invoked with M=P and N= -1, INFO is set to 3 at link time; however, at execution time, M=P= -1 will cause INFO to be correctly set to 2. The parameter checking phase often contains little data flow into the following computation phase; however, conditional assignments from the parameter checking phase must remain in the computation phase if data flow requires it. The checking portion necessarily has conditional return(s); in our implementation, it must occur at a single level of nesting. Also, parameter checking can contain no loops or labels. Appendix A.1 shows the SAXPY routine with two statements labeled 101 and 102; these statements are the parameter checking phase. Appendix B shows the SGBMV routine with the parameter checking phase on page 61.

The *computation phase* is the remainder of the routine. With knowledge of the call site, ParaScope's program compiler selects the applicable portion of the routine. This conditional cloning splits routines, at program link time, into versions with no constant conditional expressions. The decision to conditionally clone is made with high probability of achieving success because the opportunities are explicitly known from the module compiler's earlier work. Versions come into being when required and remain available; it is not advisable to generate large numbers of unused versions. A bit vector with boolean values of the constant conditional expressions encountered enables easy discovery of the same clone's prior generation. Appendix A.2 shows the results with SAXPY. Appendix B has the code selection (of a typical SGBMV invocation) highlighted by use of boldface amidst the underlying routine.

The strategy to divide a procedure into parameter checking and computation phases is a result of examining many linear algebra routines. In the next chapter, the linear algebra routines receive further attention.

# Chapter 3

# Linear Algebra Routines

Many linear algebra routines receive extensive use in both academic and industrial applications. Mathematical software is heavily used, portable, and standardized; thus, it is worthy of optimization. Improved performance at execution time yields repeated payback. Mathematical software libraries are available, but a link is required to use their functionality; prospective procedure cloning, by expanding the link's role, provides significant opportunities for optimization.

In this project, interprocedural optimization offers a technique for improving performance, and linear algebra problems are the context for evaluating gains produced by the technique. Dongarra and Grosse describe mathematical software available through the mail network [DG87], including the linear algebra routines used in this project. The following four papers provide background on these routines:

1. *Basic Linear Algebra Subprograms for Fortran Usage* [LHKK79],
2. *An Extended Set of Fortran Linear Algebra Subprograms* [DCHH86],
3. *A Set of Level 3 Basic Linear Algebra Subroutines* [DCDH88],
4. *LINPACK Users' Guide* [DBMS79].

## 3.1 Benefits of Cloning

It seems appropriate here to focus on code features of the linear algebra routines that allow successful interprocedural optimization. These features include: the degree of modularity, the ratio of serial to parallel code, and the complexity of the parameters involved.

The degree of modularity is an idea presented by Hall [Hal90]. Modularity refers to the proportion of call overhead in relation to the work done by the procedure. When high modularity is present, inlining becomes an optimization offering efficiency gains (such as in a call to copy a string). However, we point out that the call mechanism's speed is important; a slower mechanism provides more significant opportunities. Schiefler's paper finds that inlining reduces execution time by 5 to 28%, given a bulky and slow call mechanism [Sch77]; he presents no separate tests for "a significantly faster call mechanism".

Although parallel code has been the focus of interprocedurally-based optimization, we observe that few writers have studied interprocedurally-based serial code optimization. The next chapter more fully examines the potential of serial code optimization; it also discusses the distinction between serial and parallel operations. Interprocedural optimization appears to be an effective method to improve serial code. Of course, serial code optimization may increase the opportunities for parallel code optimization.

An example of serial code is the parameter checking phase, occuring frequently in linear algebra routines. The number of parameters and their complexity determine the time spent checking them. Lengthy, time-consuming and complicated checks are a significant part of code in the BLAS. Interprocedural optimization opens the possibility of eliminating this serial code. In particular, constant propagation allows many conditional expressions to be evaluated by the program linker.

The computation phase performs the work done by the BLAS. The parameters select one of the execution pathways. If constant propagation allows linker evaluation of conditional expressions, that execution pathway becomes a procedure version. Each version will have a greater proportion of parallel activity than the original routine. Routines differ. Many linear algebra routines will be examined below to determine what advantages can be gained.

## 3.2 BLAS: Basic Linear Algebra Subroutines

The Basic Linear Algebra Subroutines are the basis for more extended analysis. Properly invoked, these routines need make no calls of their own; they will be leaf nodes on a call multigraph. There are three *Levels* – corresponding to their complexity – $\underline{O(n^1), O(n^2) \text{ and } O(n^3)}$.

LINPACK and many other applications use BLAS functionality. BLAS subroutines make no use of global data. All data is received through parameters. The LINPACK routines call only corresponding LINPACK factor routines or *level 1* BLAS.

The BLAS routines use naming conventions to identify their functionality [DG87]. The first consonant stands for the data precision of the BLAS routine. The S is for real or single precision, D stands for double precision real, C stands for complex operations, and Z is for COMPLEX*16 or, if available, double complex precision. Only the single precision or S version is discussed in this thesis.

### 3.2.1 *Level 1* BLAS

The table on the next page presents the ten *Level 1* basic linear algebra routines with the number of calls to each in the LINPACK linear algebra subroutine library:

| Name | Action | LINPACK calls |
|---|---|---|
| ISAMAX | index of max abs value | 6 |
| SASUM | sum of absolute values | 38 |
| SAXPY | y = a*x + y | 68 |
| SCOPY | copy x into y | 11 |
| SDOT | dot product | 52 |
| SNRM2 | Euclidean norm | 6 |
| SROT | apply Givens' rotation | 4 |
| SROTG | setup Givens' rotation | 7 |
| SSCAL | x = a*x | 70 |
| SSWAP | swap x and y | 15 |

ISAMAX, SASUM, SDOT, SNRM2 return values; SROTG modifies its parameters; the other routines update vector(s). All routines take time O(n).

The Euclidean norm computation, SNRM2, has a different structure than most other routines; the function examines vector elements, handles unusual values and calculates a result.

Givens' plane rotation setup, SROTG, is all serial code. It prepares its parameters for use in calls to SROT that apply the rotation(s).

For all but SNRM2 and SROTG, prospective procedure cloning picks out the proper half of the computation phase, also eliminating one to three parameter checking statements. Each of these routines have a special case for increment(s) equal to 1. For these unit increments, loop unrolling is typically done with a modulus of 6, 4, 7, 5, 5 and 3 in SASUM, SAXPY, SCOPY, SDOT, SSCAL, and SSWAP. However, loop unrolling could be under linker control; often, constant array size parameters would allow the loop size to be made a factor of the bound. Thus, the modulus effort would become unnecessary (see its use in Appendix A). As the code is now, it is difficult for a compiler or linker to make a good choice. This choice is of most importance when significant parallel capability can be stalled in the serial modulus handling statements.

The increment(s) for arrays are formal parameters, but all applications in LINPACK use unit stride. The stride could be detected with any special cases generated automatically; this capability would eliminate now redundant code and programmer effort.

The best example is the SAXPY routine because, with its less powerful cousin SSCAL, it makes up 138 of LINPACK's 277 calls to BLAS *level 1* routines. Also, it has separate code for unit increments. Appendix A shows the SAXPY code.

## 3.2.2 *Level 2* BLAS

The basic linear algebra *level 2* modules perform sixteen matrix-vector operations:

| | | | |
|---|---|---|---|
| SGBMV | SGEMV | SGER | SSBMV |
| SSPMV | SSPR | SSPR2 | SSYMV |
| SSYR | SSYR2 | STBMV | STBSV |
| STPMV | STPSV | STRMV | STRSV |

These routines use many more parameters than BLAS *level 1* routines do. Twelve execution pathways are available through use of the parameters TRANS, UPLO and DIAG [DCHH86]. Also, the actions of these routines are more complicated than the *Level 1* routines. Of course, routines for the other data types are available.

There are nine complex number routines that have no corresponding real versions. CGERC and CGERU perform general rank-1 updates. CHBMV, CHEMV, CHER, CHER2, CHPMV, CHPR, and CHPR2 operate on hermitian matrices.

Dongarra puts it well: "The Level 2 BLAS involve O($mn$) scalar operations where $m$ and $n$ are the dimensions of the matrix involved. These could be programmed by a series of calls to the Level 1 BLAS, though we do not recommend that they be implemented in that way. Hence, in a natural sense, the Level 2 BLAS are performing basic operations at one level higher than the Level 1 BLAS"

[DCHH86]. Inlining and cloning *could* remove any inefficiencies caused by *level 2* routines calling *level 1*, but there is no particular reason to make this choice other than simplicity.

Naming conventions provide the type in the first consonant as for *Level 1* BLAS. For *Level 2* BLAS, the second and third letters correspond to matrix type:

| | |
|---|---|
| GB | General band matrix |
| GE | General matrix |
| HB | Hermitian band matrix |
| HE | Hermitian matrix |
| HP | Hermitian matrix stored in packed form |
| SB | Symmetric band matrix |
| SP | Symmetric matrix stored in packed form |
| SY | Symmetric matrix |
| TB | Triangular band matrix |
| TP | Triangular matrix in packed form |
| TR | Triangular matrix |

The remaining letters correspond to the operation to be performed on the previously given data and matrix type:

| | |
|---|---|
| MV | Matrix-vector product |
| R | Rank-one update |
| R2 | Rank-two update |
| SV | Solve a system of linear equations |

### 3.2.3 *Level 3* BLAS

The basic linear algebra *Level 3* modules perform six functions:

| | | |
|---|---|---|
| SGEMM | SSYMM | SSYRK |
| SSYR2K | STRMM | STRSM |

These routines solve $O(n^3)$ problems that are matrix-matrix operations [DCDH88]. Twelve execution pathways are available through use of the parameters TRANS, UPLO and DIAG.

Naming conventions provide the type as the first consonant as for *Level 1* BLAS. The second and third letters correspond to matrix type:

| | |
|---|---|
| GE | All matrices are general rectangular |
| HE | One of the matrices is Hermitian |
| SY | One of the matrices is symmetric |
| TR | One of the matrices is triangular |

The remaining letters correspond to the operation to be performed on the previous data and matrix type:

| | |
|---|---|
| MM | Matrix-matrix product |
| RK | Rank-k update of a symmetric or Hermitian matrix |
| R2K | Rank-2k update of a symmetric or Hermitian matrix |
| SM | Solve a system of linear equations for a matrix of right-hand sides |

## 3.3 An Example of Cloning's Benefits

The BLAS *Level 2* and *Level 3* routines are quite similar in structure. Appendix B has code for the SGBMV routine (with some comments omitted). It is a long routine, but only a small subset of the code applies to any given invocation. We say that this routine has high customization potential.

For the invocation CALL SGBMV('N', 1, 1, 1, 1, 0, ARRAYA, ARRAYLDA, ARRAYX, 1, 0, ARRAYY, 1), the resultant cloned code follows on the next page. Admittedly, this example is somewhat exaggerated but shows optimization's potential; the main effort of SGBMV is not performed because the parameter ALPHA equals 0. It is much easier to inline the resultant code fragment, which of course takes up much less memory. Indeed, a simple vector zeroing instruction is the only action. One hundred thirty-one executable lines of SGBMV is reduced to these four executable lines in figure 3.1.

```
************************************************************
      SUBROUTINE SGBMV ( TRANS, M, N, KL, KU, ALPHA, A,
     $                   LDA, X, INCX, BETA, Y, INCY )
*     .. Scalar Arguments ..
      REAL               ALPHA, BETA
      INTEGER            INCX, INCY, KL, KU, LDA, M, N
      CHARACTER*1        TRANS
*     .. Array Arguments ..
      REAL               A( LDA, * ), X( * ), Y( * )
*
*     .. Parameters ..
      REAL               ZERO
      PARAMETER        ( ZERO = 0.0E+0 )
*     ..
*     .. Executable Statements ..

      DO 10, I = 1, M
         Y( I ) = ZERO
   10 CONTINUE
      RETURN
*
*     End of SGBMV .
*
      END
*
************************************************************
```

Figure 3.1: A Cloned Version of SGBMV

## 3.4 The LINPACK Subroutine Library

LINPACK offers solutions to linear algebra problems. As with the BLAS, there are different routines for different data types. The package makes 277 calls to *Level 1* BLAS routines; there are further details regarding these invocations in the section on *Level 1* BLAS. The only other calls are from seven matrix condition routines to corresponding LINPACK matrix factor routines. These are the 40 single precision routines in LINPACK:

| | | | | |
|---|---|---|---|---|
| SCHDC | SCHDD | SCHEX | SCHUD | SGBCO |
| SGBDI | SGBFA | SGBSL | SGECO | SGEDI |
| SGEFA | SGESL | SGTSL | SPBCO | SPBDI |
| SPBFA | SPBSL | SPOCO | SPODI | SPOFA |
| SPOSL | SPPCO | SPPDI | SPPFA | SPPSL |
| SPTSL | SQRDC | SQRSL | SSICO | SSIDI |
| SSIFA | SSISL | SSPCO | SSPDI | SSPFA |
| SSPSL | SSVDC | STRCO | STRDI | STRSL |

LINPACK uses no global data. LINPACK has little customization potential. The routines vary tremendously in the number and nature of calls that they make. One routine may make many identical calls to *Level 1* BLAS with unvarying parameters. Another routine makes no calls. Some routines make many entirely different calls with rapidly varying parameters.

LINPACK routines do not have a parameter checking phase. There is little opportunity to clone the computation phase. The main opportunity comes in how LINPACK interfaces with the *Level 1* BLAS.

# Chapter 4

# Amdahl's Law: Serial Execution and 1/s

Amdahl's law (in its simplist form) states that "execution gains are limited to 1/s where s is the serial portion of a program" [Amd67]. Since the computation inside mathematical routines such as the BLAS can be easily parallelized, the serial portion of these routines is of greater interest. The serial portion is usually made up of call overhead, parameter checking and variable initialization. The next chapter details serial program portions in the linear algebra domain.

The increase in hardware's vector and parallel capability can make the serial portion much more important *relatively*. For example, the Stardent Titan has 4 processors each with a vector capability. It is not always possible to take advantage of all this computational power; dependence relations may restrict any advantage. However, if the serial portion is reduced by interprocedural analysis, the potential speed of execution increases substantially, barring other difficulties. Hardware capability has thus put new demands on software to improve performance. With mathematical software, the expense of loop execution often dominates a procedure, and dependence analysis determines the major constraints on parallel improvement. However, dependence analysis can improve with serial code optimization as will be shown at the end of the chapter.

## 4.1 An Example

Suppose a fictitious routine spends a tenth of its execution time in an inherently serial portion and the remainder in a loop allowing parallel operations. The parallel portion gets a speedup equal to the number of processors. As the following table shows, serial execution takes a rapidly increasing part of the total as the number of processors increases.

| Processors | Serial Portion |
|---|---|
| 1 | 10% |
| 2 | 18% |
| 4 | 31% |
| 8 | 47% |
| 16 | 64% |
| 32 | 78% |
| n | 100% |

Routine complexity is important when considering optimization of serial code; for instance, $O(n)$ tasks see more improvement than $O(n^3)$ ones as $n$ grows.

## 4.2 Gustafson's Work

John Gustafson points out that increased use of parallel and vector computers allows problem dimensionality to be increased; better answers in the same time are his desired result [Gus88]. He assumes that the serial portion is independent of problem size, thus is fixed; parallelism is used to improve the quality of the result by increasing array size. His example has a minute serial portion "for three very practical applications (s=0.4-0.8 percent)" [Gus88]. With such a minor serial portion, improvement by its diminution appears difficult and unlikely anyway. However, this and the next chapter provide practical examples where serial code optimization can work well.

## 4.3 Implications

With increasing performance in loop execution, the focus of optimization may change. ParaScope's creators use inline substitution [CHT90a] to study the four vector processor Convex C240 and Stardent Titan machines. On average for eight programs, Convex vector and parallel execution sped up to 2.5 and 2.25 times that of unoptimized uniprocessor code; Stardent vector and parallel execution went 1.47 and 1.03 faster than that of unoptimized uniprocessor code. If inlining had been performed previous to compilation, for "both compilers the difference [in reduced execution time] was just under seven percent" [Hal91]. Of course, this reduction is divided between savings on serial and parallel execution.

However, the averages conceal some significant improvement. In particular, with the LINPACK benchmark in double precision, the execution speedup improves significantly. And with inlining performed, the speedups are even better. Figure 4.1 shows the speedups from [CHT90a].

The next chapter examines linear algebra routines' time demands in both serial and parallel operations. As stated at this chapter's start, increasing vector and parallel capability make the serial portion more important.

**LINPACK Speedups**

(Relative to unoptimized serial execution)

| | | Inlining | Performed |
|---|---|---|---|
| Machine | Mode | No | Yes |
| Convex | Parallel | 4.8 | 5.6 |
| | Vector | 3.8 | 7.1 |
| Stardent | Parallel | 5.0 | 5.0 |
| | Vector | 4.8 | 7.1 |

Figure 4.1: LINPACK Speedups

# Chapter 5

# Results of Prospective Procedure Cloning

This chapter presents experiments using ParaScope and prospective procedure cloning. The compile time, link time, execution time, and object size of linear algebra routines were measured; we evaluate ParaScope both with and without prospective procedure cloning. Small vectors and arrays allow improvement in serial execution to be more significant than with larger spaces. Appendix C provides details on the computation environment.

More specifically, this chapter will discuss the costs of using ParaScope with its interprocedural analysis, elimination of parameter checking, and conditional cloning. The chapter then shows the resultant benefits: execution efficiency, object size reduction, and programmer feedback. Chapter 6 provides conclusions on how the costs compare to the benefits.

In a concluding note, we offer improvements to Hall's Goal-Directed Interprocedural Optimization. We review the merging phase of her algorithm suggesting instead a simpler selection phase. Also, we recommend a space management capability that allows greater flexibility in space utilization.

Our experiments focus on four software packages: the *Level 1*, *2*, *3* BLAS and the LINPACK subroutine package.

| Routine Class | Sun Time | Sun Size | ParaScope Time | ParaScope Size |
|---|---|---|---|---|
| BLAS*Level 1* | 0.2:0.2 | 1140 | 0.3:0.16 | 10375 |
| BLAS*Level 2* | 0.7:0.2 | 4809 | 0.9:0.15 | 33213 |
| BLAS*Level 3* | 0.9:0.2 | 7440 | 1.1:0.15 | 45965 |
| LINPACK | 0.5:0.2 | 3811 | 0.8:0.22 | 30007 |

Sun Time – user:system seconds on SUN 4

Sun Size – average size of object file (.o)

Parascope Time – user:system seconds on SUN 4

Parascope Size – average size of abstract syntax tree

Figure 5.1: Experiment One – Average Compile Time and Sizes

## 5.1 Costs of Prospective Procedure Cloning

Our discussion will touch on Fortran compilation and linkage, using ParaScope and conventional language tools. Then we consider two features of prospective procedure cloning: parameter checking elimination and conditional cloning.

### 5.1.1 ParaScope and its Costs

ParaScope performs interprocedural analysis in addition to other compiler and linker duties. As most compilers do, ParaScope takes O(n) time to perform its basic job, where $n$ is the size of the source file [CK87a]. In figure 5.1, a comparison is made between compilation on the Sun to that done by ParaScope. Of course, the Sun columns reflect creation of an object module while the ParaScope columns reflect creation of an abstract syntax tree with interprocedural information. A time allowance for object code generation needs to be made for ParaScope.

In user time, ParaScope compilation takes from 20% to 60% longer than that of the Sun compiler, because of the time required for interprocedural analysis and

| Routine Class | Sun Linker | ParaScope Program Compiler |
|---|---|---|
| BLAS *Level 1* | 0.5:0.3 | 0.1:0.3 |
| BLAS *Level 2* | 0.7:0.3 | 0.1:0.3 |
| BLAS *Level 3* | 0.5:0.3 | 0.1:0.2 |
| LINPACK | 0.9:0.3 | 0.2:0.4 |

user:system seconds on SUN 4

Figure 5.2: Experiment Two – Total Link Times

other capabilities. However, in system time, Parascope requires from 25% less to 10% more to perform its work. With our implementation of prospective procedure cloning, the user and system times increase no more than 30% and 100% over those of regular ParaScope; our inexpert implementation requires an additional scan of the abstract syntax tree which could be much more efficiently performed as interprocedural information generation occurs.

Next, we evaluate the link phase in figure 5.2; here, the operations differ more than for compilation. ParaScope's program compiler *manages* the link done by the host software while the Sun linker *integrates* objects into an executable. The textual interface to the composition editor appears to take a tenth of a second of user time and two tenths of a second in system time for these applications. The program compiler was invoked with the -n flag so no compilation (or linking) occurred.

For the entries above, all routines in each routine class become part of the executable program image since a main program calls them all. For now, the ParaScope link times do not include interprocedural analysis while the programming environment is being improved. At the link phase, ParaScope takes comparable time to perform its task. The overall task includes generation of the program summary graph, computation of the interprocedural sets, side-effect analysis, and constant propagation. With our implementation of prospective procedure cloning, the user and system times should increase significantly; however, a more expert implementation would require modest time demands. The implementation is grossly inefficient in that a merging of the two routines involved is used to generate the bit vector described on page 17; a better method would use the constant propagation to set up the constant parameters for analysis of the called routine. The next step chooses optimizations and performs them.

It is obvious that ParaScope requires up to ten times the investment in memory, if the software generates a version of the original source for compilation. However, ParaScope produces an abstract syntax tree with added interprocedural data; this information remains available to the user and the programming environment. Indeed, the interprocedural information is used by the program compiler. In terms of time, the module and program compiler require no more than twice that of conventional language tools.

### 5.1.2 Elimination of Parameter Checking and its Costs

As computation of interprocedural information proceeds, analysis of scalar parameters that are constant on invocation and invariant in the procedure body determines those conditional expressions that are thus constant. These conditional expressions receive a new classification so they can be found quickly. The O(n) cost for interprocedural information remains the same, though a negligible increase in compilation time can be expected as more work occurs during interprocedural data generation. Also, a few extra entries in the abstract syntax tree need to be kept; these entries indicate optimization potential and important locations in the tree.

At program link time, constant conditional expressions determine whether the parameter checking portion of the called routine detects if a return statement will be performed. If no conditional return necessarily occurs, the parameter checking code (such as on page 61 in Appendix B) need not be called and ParaScope then does conditional cloning on the computation phase of the routine.

### 5.1.3 Conditional Cloning and its Costs

At compile time, the costs of conditional cloning are the same as for elimination of parameter checking. At program link time, any constant conditional expressions determine whether cloning of the computation phase is possible and advantageous. An example of such cloning appears in Section 3.3.

As previously stated, our implementation of prospective procedure cloning, the user and system times increase up to 30% and 100% for the module compiler. For the program compiler, the user and system times increase up to TBD% and TBD%.

## 5.2 Benefits of Prospective Procedure Cloning

We first examine the BLAS procedures and functions followed by LINPACK. Our focus is on execution efficiency, object size reduction and programmer feedback.

Gains in execution efficiency can come in code of a serial nature: call overhead, parameter checking, variable setups, and memory demands. An obvious gain in execution efficiency appears to be offered by inlining : elimination of procedure call overhead. Many matrix computation routines have many constant parameters and thus many other optimizations are possible. However, both in *An Experiment with Inline Substitution* [CHT90a] and in our evaluation of LINPACK, it appears that inlining by itself provides no consistent benefit; rather it can enable other worthwhile optimizations.

The greatest gain in execution efficiency usually comes in using parallel capability in loops. Code simplification, accomplished by prospective procedure cloning, can ease the job of parallel compilers. Cloning, as well as inlining, can produce the opportunities discussed in section 2.3.1.

In each execution environment, different optimizations are more desirable or possible. Opportunities depend on the resources available; the next few sections will detail the benefits of prospective procedure cloning.

### 5.2.1 Execution Efficiency Benefits

Using constant parameters for the unchanged scalar arguments, most optimized BLAS routines spend less time in serial execution. The following tables document these reduced times. As discussed in the second chapter, interprocedural optimization makes this result possible. As array size decreases or increases, the relative importance of this optimization increases or decreases, respectively. As more processors become available, the importance of this optimization increases substantially. We note that, on a parallel/vector machine, the relative improvement would be greater. Compilation is done using f77 on a Sun-4 computer.

In figure 5.3, we examine the *Level 1* BLAS. These routines are the building blocks of LINPACK. Thus, an examination of their optimization potential gives a basis for evaluation of LINPACK. The tests use loop limits of eight. Each routine is called a number of times so that around a half million iterations occur; this iteration length achieves a good time average. ISAMAX is sensitive to increasing data values as it has to save information when a new maximum is found. SROTG has no loop, and its time is spent doing costly calculations. SROT has a loop and does costly calculations.

The Original Time column above has the execution time with parameter checking and all other serial activity done in the original version. The Improved Time

| Routine Name | Original Time | Improved Time |
|---|---|---|
| ISAMAX | 12.2 | 11.4 |
| SASUM | 10.8 | 10.4 |
| SAXPY | 12.7 | 11.0 |
| SCOPY | 11.0 | 9.2 |
| SDOT | 13.8 | 12.2 |
| SNRM2 | 29.4 | 27.1 |
| SROT | 28.0 | 27.5 |
| SROTG | 14.0 | 14.3 |
| SSCAL | 13.0 | 11.9 |
| SSWAP | 16.1 | 14.9 |
| average | 16.4 | 15.0 |

(in microseconds with 8 element vector)

Figure 5.3: Experiment Three – BLAS *Level 1* Execution Times

column shows the results of parameter checking reduction and of conditional cloning the unit stride(s).

The average times are not too meaningful because unusual routines skew the column totals. Removal of SROTG's results creates averages of 16.3 and 14.8. Further removal of SROT's results has averages of 14.9 and 13.5. Thus, a tenth of the user time is now saved. SCOPY has the best relative improvement; it and SSWAP perform simple, noncomputational tasks. Also, SAXPY and SDOT show considerable improvement. SROTG operates more slowly, its object does not change.

The *Level 2* BLAS perform $O(n^2)$ tasks. In figure 5.4, the tests use 4 by 4 matrices. Each routine was executed over five thousand times to achieve a good average with each matrix element initialized to its index value times 1.013 [DDHH88].

|  |  | Original | Improved |
|---|---|---|---|
| Routine | Total | Serial | Serial |
| Name | Time | Time | Time |
| SGBMV | 11.2 | 1.25 | 0.50 |
| SGEMV | 19.5 | 2.50 | 3.50 |
| SGER | 0.9 | 0.13 | 0 |
| SSBMV | 4.5 | 0.50 | 0.30 |
| SSPMV | 4.1 | 0.50 | 0 |
| SSPR | 0.5 | 0.09 | 0.06 |
| SSPR2 | 0.8 | 0.06 | 0 |
| SSYMV | 9.5 | 1.25 | 0.50 |
| SSYR | 0.7 | 0.06 | 0.13 |
| SSYR2 | 1.0 | 0.13 | 0.13 |
| STBMV | 1.8 | 0.38 | 0.25 |
| STBSV | 1.8 | 0.44 | 0.13 |
| STPMV | 1.1 | 0.31 | 0.13 |
| STPSV | 0.6 | 0.22 | 0.13 |
| STRMV | 3.4 | 0.75 | 0.25 |
| STRSV | 1.6 | 0.31 | 0.19 |
| average | 3.9 | 0.55 | 0.39 |

(in microseconds for 4 by 4 matrices)

Figure 5.4: Experiment Four – BLAS *Level 2* Execution Times

| Routine Name | Total Time | Original Serial Time | Improved Serial Time |
|---|---|---|---|
| CGERC | 5.4 | 0.19 | 0.19 |
| CGERU | 5.4 | 0.19 | 0.06 |
| CHBMV | 13.0 | 0.50 | 0.38 |
| CHEMV | 26.0 | 1.00 | 0.50 |
| CHER | 3.5 | 0.37 | 0.06 |
| CHER2 | 3.3 | 0.16 | 0.06 |
| CHPMV | 6.2 | 0.50 | 0.25 |
| CHPR | 2.5 | 0.09 | 0.03 |
| CHPR2 | 3.0 | 0.09 | 0.03 |
| average | 7.6 | 0.34 | 0.17 |

(in microseconds for 4 by 4 matrices)

Figure 5.5: Experiment Five – BLAS Complex *Level 2* Execution Times

The *Level 2* BLAS improved serial times shown above vary greatly. We see average improvement of 4% when the improved serial time is subtracted from the original serial time and then divided by the total time. For a few routines, improved serial time appears as 0 because too few iterations occured to get a true measurement. SGEMV and SSYR show speed degradation, and others have gains up to 17%.

In figure 5.5, we examine those *Level 2* BLAS doing O($n^2$) tasks for complex numbers that have no corresponding real routine. The tests use 4 by 4 matrices. Again, each routine was executed over five thousand times to achieve a good average with each matrix element in X and Y initialized to CMPLX(I*1.013, -I * 0.987) and in A to CMPLX(I*1.013, REAL((J-1)*N + I)) [DDHH88].

The *Level 2* complex BLAS have 2% speedup. The complex number type requires four times the computation as is required for reals; thus, the parameter

| Routine Name | Total Time | Original Serial Time | Improved Serial Time |
|---|---|---|---|
| SGEMM | 56.5 | 2.25 | 0.50 |
| SSYMM | 10.1 | 0.50 | 0.19 |
| SSYRK | 5.3 | 0.19 | 0.09 |
| SSYR2K | 8.5 | 0.22 | 0.09 |
| STRMM | 1.8 | 0.19 | 0.06 |
| STRSM | 2.0 | 0.14 | 0.03 |
| average | 14.0 | 0.58 | 0.16 |

(in microseconds for 4 by 4 matrices)

Figure 5.6: Experiment Six – BLAS *Level 3* Execution Times

checking code is a smaller percentage of the total time, and they exhibit less speedup.

The *Level 3* BLAS perform $O(n^3)$ tasks. As before, the tests use 4 by 4 matrices. Also as before, each routine was executed over five thousand times to achieve a good time average with each matrix element B(I,J) or C(I,J) initialized to 1.0E0 + REAL(I+(J-1)*N) / (M*N+1) [DDDH88] where M and N are the dimensional bounds.

In figure 5.6, the *Level 3* BLAS achieve an average 3% speedup when the routine being executed does not change for five thousand calls. However, as figure 5.9 shows, on average 6380 of a routine's 7440 bytes are no longer in the object; thus, a more significant effect could occur when the called routine changes frequently. A much smaller object needs less traffic between memory and instruction cache, or disk and virtual memory.

The BLAS offers a variety of routines, and LINPACK has forty routines. This large number of choices is necessary, but greater variety would make it difficult for a user to deal with. The solution is to use parameters to customize routine action; many routines perform many different actions determined from their arguments.

In the LINPACK routines, there is little that can be improved except in how they interface with *Level 1* BLAS subroutines. Some routines use parameters to choose actions; the options could be used to create a shorter routine. While seven LINPACK condition routines call corresponding LINPACK factor routines, little optimization opportunity is available since the call is unconditional. Every call to *Level 1* BLAS includes increment(s) which are always equal to unity. Some LINPACK routines make the exact same call many times; thus, if parameters become mutually known, parameterless invocation is possible.

The LINPACK subroutine SGBCO offers a good chance for optimization. It makes four identical invocations of SASUM, eight identical invocations of SSCAL, another call to SASUM, two calls to SAXPY, and a call to SDOT. Also, there is an unconditional call to the LINPACK routine SGBFA. With an 8 by 8 array argument, we evaluated SGBCO's operation on a Sequent Symmetry. By eliminating BLAS *Level 1* parameter checking, a 2% speedup resulted. By eliminating now constant tests, another 1% was saved. The obvious elimination of the unit stride parameter passing yielded only inconsequential savings.

### 5.2.2 Object Size Reduction Benefits

With constant arguments used for the unchanged scalar parameters, prospective procedure cloning can effectively tailor most BLAS routines; the resultant object size is substantially smaller. The following tables show the decreased object space (in bytes) created by prospective procedure cloning. Smaller embedded programs could save on memory requirements and thus on hardware costs.

Individually, procedure size reductions are not too important. However, memory operates at a much lower speed than a processor does, so more significant time

| Routine Name | Original Object | without Parameter Checking | with Prospective Cloning |
|---|---|---|---|
| ISAMAX | 796 | 756 | 376 |
| SASUM | 1048 | 976 | 644 |
| SAXPY | 1256 | 1136 | 656 |
| SCOPY | 1156 | 1124 | 660 |
| SDOT | 1276 | 1204 | 716 |
| SNRM2 | 1756 | 1676 | 1676 |
| SROT | 1008 | 976 | 420 |
| SROTG | 932 | 932 | 932 |
| SSCAL | 960 | 928 | 604 |
| SSWAP | 1220 | 1188 | 684 |
| average | 1140 | 1089 | 736 |

Figure 5.7: Experiment Seven – *Level 1* BLAS Object Size

savings may come from reduced cache and virtual memory traffic. Thus, if many programs can be made smaller, decreased memory demands can result.

In this section, separate compilation occurred on a Sun computer for all routines. The Fortran compiler is invoked on each routine with the -c flag. Each object file has 36 bytes of header and other information. See Appendix C for details on the computation environment. In figure 5.7, the sizes given are those for the object files; thus, some bytes of header and relocation information are present.

All routines but SROTG have a few statements of parameter checking that become unnecessary. Except for SNRM2 and SROTG, the *Level 1* BLAS have a special case for unit stride(s) which we use here. Without the large SNRM2 and SROTG routines, the average 35% object size reduction increases to 45%. This latter number reflects the elimination of the nonunit increment case(s). The explicit loop unrolling in six routines limits the object size reduction to less than half; with no unrolling, reduction by more than half would occur.

Figure 5.8 repeats the measurements for the *Level 2* BLAS. The object size reduction averages 80% for the average routine. The typical invocation uses parameters having unit strides, real values other than zero or one, and the parameter TRANS set to 'N'. Elimination of parameter checking reduces object size by 14%, and conditional cloning removes the other 66%.

Figure 5.9 shows that the *Level 3* BLAS have even more customization potential than previous levels. An object size reduction of 85% is possible for the average routine. This typical invocation uses parameters having unit strides, real values other than zero or one, the parameter TRANS set to 'N'. the parameter SIDE set to 'L', DIAG set to 'N', and UPLO set to 'U'. Elimination of parameter checking reduces object size by 14%, and conditional cloning removes the other 71%.

LINPACK makes many calls to *Level 1* BLAS routines. These *Level 1* BLAS routines require less code space, but LINPACK itself has little opportunity for object size reduction. The reduced code space affects any decision to inline.

### 5.2.3 Programmer Feedback Benefits

With prospective procedure cloning, the programmer and user can know how much serial code they may remove from the program image. Interprocedural information permits a range of optimization techniques. The Chapter 6 outlines a larger, more useful feedback capability that allows management of the optimization process.

## 5.3 Routine Clone Selection

Hall's dissertation [Hal91] proposes a *merging* phase to be performed as the program comes together. After constant propagation, a variety of cloning activity may be indicated for a routine. It is important to know which variables contribute to

| | | without | with |
|---|---|---|---|
| Routine | Original | Parameter | Prospective |
| Name | Object | Checking | Cloning |
| CGERC | 2332 | 1764 | 936 |
| CGERU | 2256 | 1688 | 888 |
| CHBMV | 7304 | 6600 | 1584 |
| CHEMV | 6560 | 5816 | 1440 |
| CHER | 5560 | 6564 | 1772 |
| CHER2 | 7252 | 4932 | 1392 |
| CHPMV | 6256 | 5648 | 1328 |
| CHPR | 4656 | 5876 | 1520 |
| CHPR2 | 6428 | 4164 | 1176 |
| SGBMV | 5216 | 4292 | 972 |
| SGEMV | 4096 | 3188 | 828 |
| SGER | 1832 | 1296 | 620 |
| SSBMV | 5368 | 4680 | 1108 |
| SSPMV | 4272 | 3680 | 848 |
| SSPR | 2548 | 3000 | 704 |
| SSPR2 | 3528 | 2060 | 540 |
| SSYMV | 4648 | 3920 | 964 |
| SSYR | 2748 | 3160 | 792 |
| SSYR2 | 3832 | 2124 | 596 |
| STBMV | 6444 | 5440 | 808 |
| STBSV | 6444 | 5440 | 760 |
| STPMV | 4956 | 4056 | 576 |
| STPSV | 4956 | 4056 | 580 |
| STRMV | 5372 | 4336 | 672 |
| STRSV | 5372 | 4336 | 700 |
| average | 4809 | 4084 | 964 |

Figure 5.8: Experiment Eight – *Level 2* BLAS Object Size

| | | without | with |
|---|---|---|---|
| Routine | Original | Parameter | Prospective |
| Name | Object | Checking | Cloning |
| SGEMM | 6228 | 4748 | 1020 |
| SSYMM | 6164 | 4908 | 1044 |
| SSYR2K | 7688 | 6340 | 1316 |
| SSYRK | 6496 | 5276 | 972 |
| STRMM | 8460 | 7160 | 952 |
| STRSM | 9608 | 8308 | 1060 |
| average | 7440 | 6123 | 1060 |

Figure 5.9: Experiment Nine – *Level 3* BLAS Object Size

subscript calculation as these determine the potential improvement in dependence analysis inside a routine. Each clone can be selected with the generation of a bit vector; thus, the effort of mergability detection is reduced. However, an important variable may not be constant at the call site or may not have a special case in the calling routine; in this situation, further work is necessary. Efficient, nonduplicative clone generation is important, especially since unnecessary customization would expand memory requirements without any benefit.

Space management is most important when program size is a concern. If execution time is adversely affected by program size, prospective procedure cloning offers size reduction. Hall suggests a limit on increasing program size – a doubling; however, if prospective procedure cloning works effectively, freed space becomes available so that Goal-Directed Interprocedural Optimization can suffer less of a space penalty.

# Chapter 6

# Conclusions

In this thesis we found that prospective procedure cloning was effective in optimizing linear algebra routines, especially in terms of decreased object size. If the proportion of serial to parallel execution is a concern, prospective procedure cloning offers a method to reduce that proportion; the discussion in Chapter 4 explains Amdahl's Law and this point. Other observations made during this project suggest future work on ways of using ParaScope to manage program creation and execution.

## 6.1 Object Size Reduction

Object size reduction is significant with almost all BLAS routines but inconsequential with the LINPACK routines. *Level 1* BLAS average a 35% reduction (with a greater decrease of 45% without the two large unoptimizable routines). The *Level 2* BLAS average an 80% diminution. The greatest average decrease is 85% for the *Level 3* BLAS. LINPACK offers no object size reduction itself; however, LINPACK's use of the *Level 1* BLAS presents a minor opportunity for improvement.

The success in object size reduction just noted results from inherent characteristics of the linear algebra routines. These routines offer up to 40 routine choices per group and any greater choice would make selection overly difficult. Thus, parameters provide the means to further select actions so that for some routines less than an eighth of the code applies to an invocation.

The user of linear algebra routines has a better interface than if more choices faced him or her. After routine selection, the user selects parameter values to customize operation. However, a penalty in object code space and execution time will occur with conventional language tools.

The programmer of linear algebra routines need not develop additional separate routines. Prospective procedure cloning selects or generates the appropriate object for the particular call site. It has been noted that good programming style can conflict with efficient execution; interprocedural optimization lessens this problem. BLAS *Level 2* and *Level 3* routines contain very substantial customization potential; it could be said that each routine is like a page in a categorized menu in a restaurant offering many choices. This ease of programming is an important consideration – as was the ease of selection for the user.

The cost for this ease of development and use is increased object size and execution time. Prospective procedure cloning provides a practical, low expense solution to this problem. As the first and second experiments in the fifth chapter show, ParaScope compilation and linkage require no more than twice the time taken by conventional compilers and linkers. The inclusion of prospective procedure cloning added no more than 40% to the total of ParaScope's user and system times (and would add much less with a more expert implementation).

## 6.2 Execution Efficiency

Prospective procedure cloning is just one step towards optimal program execution. Interprocedural capabilities are necessary to obtain a comprehensive program view.

A variety of optimization techniques, used in concert, will provide consistently superior execution performance.

Serial execution efficiency is improved by prospective procedure cloning. *Level 1* BLAS save about 10% of execution time with a vector length of 8. *Level 2* BLAS, using 4 by 4 arrays of reals, operate 4% faster; *Level 3* BLAS operate 3% faster. LINPACK routines, using 8 by 8 arrays, achieve a modest 3% time savings. However, the tests did not stress the code load time but rather focused on pure execution time savings. If optimized BLAS routines trade off execution, more time savings may result.

Prospective procedure cloning works best when array size is small and/or significant parallel capability is available. Amdahl's Law points out that parallel improvement can not exceed 1/s, where $s$ is the serial portion. Prospective procedure cloning can also increase the success achieved by dependence analysis.

With interprocedural optimization, Hall and others offer very worthwhile techniques outside the serial domain. Execution speed doubles after Hall's effort with *matrix300*. In Hall's inlining study (discussed at the end of the fourth chapter), parallel capability provides speedups of five or more with LINPACK.

## 6.3 Programmer Feedback

The programmer needs feedback on where execution expense can be reduced by changing parameters or by adding information. When ParaScope offers the features in Hall's dissertation, execution time estimation will be closer to realization. Indeed, Goal-Directed Interprocedural Optimization bases its choices on such time estimation.

There are multiple measures of performance with clock time being the most simplistic but most reliable. For a machine, the maximum or unconstrained floating point operation, FLOP, rate would be known and adjustments to it computable by the programming environment. These adjustments include: dependence preservation, serial code, inadequate parallelism, and data interlocks.

With a feedback facility added to the Parascope environment, the programmer could see alternatives in the use of the procedures. The advantages of constant parameters would become better known. The dependence problems are accessible. The improvements due to parallel computation become evident.

## 6.4 Future Work

Prospective procedure cloning specifically works with code features found in the BLAS routines. Further code surveys can point out additional opportunities. Nonetheless, prospective procedure cloning is a specialized technique and may not provide improvement in many circumstances. As a part of Goal-Based Interprocedural Optimization, the technique addresses a particular goal.

ParaScope is a large program, and prospective procedure cloning is just a small portion of this greater whole which includes Goal-Based Interprocedural Optimization. Dependence analysis and a wide range of parallel optimizing techniques are open to use. Also, ParaScope provides many features to the programmer to speed the process of program development and debugging.

A variety of optimizations is needed to produce general improvement. Prospective procedure cloning achieves more with serial code so its optimizations retain their advantages with small array usage. Hall's Goal-Directed Interprocedural Optimization focuses on making large array accesses efficient; inlining and cloning occur to create opportunities for other optimizations. ParaScope discovers much information that allows superior parallel loop execution, but this arena is beyond the scope of this thesis; a good overview of parallel optimization is by Padua and Wolfe [PW86].

If reusable software is an important goal, procedure cloning retains the advantages of separate routines but removes the penalty of including code without any chance of execution.

Future work could integrate the code size management aspect of prospective procedure cloning with Hall's Goal-Directed Interprocedural Optimization. An effective tool to manage object size would attract demand from applications requiring speed or being mass-produced. Parascope could perform its work as resources are available so as to provide its capabilities at little cost.

Finally, a programming environment capable of creating optimal program execution would fully realize software's potential. Performance – in terms of execution time – could be managed explicitly. Performance is a tradeoff with many variables, and interprocedural analysis helps to determine and order these variables. Scientific programmers could depend on knowing any performance degradation came from known computational requirements or compromises chosen because optimization cost exceeded execution savings. The programming environment could show these constraints so that they are readily understood.

# Bibliography

[AC72] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In J. Rustin, editor, *Design and Optimization of Compilers.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Amd67] G. M. Amdahl. Validitity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Reston, Virginia, 1967.

[Bal79] J.E. Ball. Predicting the effects of optimization on a procedure body. In *SIGPLAN 79 Symposium on Compiler Construction*, pages 214–220, August 1979.

[CCH+87] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon, and Scott K. Warren. A Practical Environment for Scientific Programming. *IEEE Computer*, 20(11):75–89, November 1987.

[CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural Constant Propagation. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pages 152–161, Palo Alto, CA, June 1986.

[CHT90a] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An Experiment with Inline Substitution. Technical Report TR 90-128, Rice University, Houston, Texas, August 1990.

[CHT90b] Keith D. Cooper, Mary Wolcott Hall, and Linda Torczon. The Perils of Interprocedural Knowledge. Technical Report TR 90-132, Rice University, Houston, Texas, September 1990.

[CK87a] Keith D. Cooper and Ken Kennedy. Complexity of Interprocedural Side-effect Analysis. Technical Report TR 87-61, Rice University, Houston, Texas, October 1987.

[CK87b] Keith D. Cooper and Ken Kennedy. Interprocedural Side-effect Analysis in Linear Time. Technical Report TR 87-62, Rice University, Houston, Texas, October 1987.

[Con83] Reidar Conradi. Interprocedural Optimization of Object Code. Technical Report TR 25/83, University of Trondheim, Trondheim, Norway, September 1983.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide.* SIAM Publications, Philadelphia, PA, 1979.

[DCDH88] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subroutines. Technical report, Argonne National Laboratory, Argonne, Illinois, August 1988.

[DCHH86] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. Technical Report Technical Memorandum No. 41 (Revision 3), Argonne National Laboratory, Argonne, Illinois, September 1986.

[DDDH88] J. J. Dongarra, J. DuCroz, I. S. Duff, and R. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs. Technical Report ANL-MCS-P88-2, Argonne National Laboratory, August 1988.

[DDHH88] J. J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Software*, 14(1):18–32, March 1988.

[DG87] Jack Dongarra and Eric Grosse. Distribution of Mathematical Software Via Electronic Mail. *Communications of the ACM*, 30(5):403–407, May 1987.

[Gus88] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[Hal90] Mary Wolcott Hall. Improving Parallelism After Inline Substitution. Technical Report TR 90-127, Rice University, Houston, Texas, August 1990.

[Hal91] Mary Wolcott Hall. Managing Interprocedural Optimization. PhD Dissertation TR 91-157, Rice University, Houston, Texas, April 1991.

[Hus82] C.A. Huson. An in-line subroutine expander for parafrase. M.S. thesis TR UIUCDCS-R-82-1118, University of Illinois, Urbana-Champaign, December 1982.

[KDLS86] David J. Kuck, Edward S. Davidson, Duncan H. Lawrie, and Ahmed H. Sameh. Parallel Supercomputing Today and the Cedar Approach. *Science*, 231:967–974, February 1986.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[PW86] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Sch77] Robert W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. *Communications of the ACM*, 20(9):647–654, September 1977.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[WZ89] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. Technical Report TR CS-89-36, Brown University, May 1989.

# Appendix A

# Level 1 BLAS SAXPY subroutine

First, the unchanged saxpy routine is presented followed by versions using increments of one.

## A.1 Original subroutine

```
      subroutine saxpy(n,sa,sx,incx,sy,incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loop for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      real sx(1),sy(1),sa
      integer i,incx,incy,ix,iy,m,mp1,n
c
 100  if(n.le.0)return
 101  if (sa .eq. 0.0) return
 102  if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
```

```
c           not equal to 1
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        sy(iy) = sy(iy) + sa*sx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20   m = mod(n,4)
        if( m .eq. 0 ) go to 40
           do 30 i = 1,m
              sy(i) = sy(i) + sa*sx(i)
   30      continue
           if( n .lt. 4 ) return
   40      mp1 = m + 1
      do 50 i = mp1,n,4
        sy(i) = sy(i) + sa*sx(i)
        sy(i + 1) = sy(i + 1) + sa*sx(i + 1)
        sy(i + 2) = sy(i + 2) + sa*sx(i + 2)
        sy(i + 3) = sy(i + 3) + sa*sx(i + 3)
   50 continue
      return
      end
```

## A.2 With Prospective Procedure Cloning

```
      subroutine saxpy(n,sa,sx,incx,sy,incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loop for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      real sx(1),sy(1),sa
      integer i,incx,incy,ix,iy,m,mp1,n

 20   m = mod(n,4)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        sy(i) = sy(i) + sa*sx(i)
   30 continue
      if( n .lt. 4 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,4
        sy(i) = sy(i) + sa*sx(i)
        sy(i + 1) = sy(i + 1) + sa*sx(i + 1)
        sy(i + 2) = sy(i + 2) + sa*sx(i + 2)
        sy(i + 3) = sy(i + 3) + sa*sx(i + 3)
   50 continue
      return
      end
```

## A.3 With n a Factor of 4

```
      subroutine saxpy(n,sa,sx,incx,sy,incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loop for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c
      real sx(1),sy(1),sa
      integer i,incx,incy,ix,iy,m,mp1,n

      do 50 i = 1,n,4
         sy(i) = sy(i) + sa*sx(i)
         sy(i + 1) = sy(i + 1) + sa*sx(i + 1)
         sy(i + 2) = sy(i + 2) + sa*sx(i + 2)
         sy(i + 3) = sy(i + 3) + sa*sx(i + 3)
   50 continue
      return
      end
```

# Appendix B

# Level 2 BLAS SGBMV subroutine

The Level 2 BLAS routine SGBMV is presented here with statements selected by the invocation:

CALL SGBMV('N', 1, 1, 1, 1, 0.5, A, 10, X, 1, 0.5, Y, 1)

highlighted by use of

**`boldface`**

type and a caret in the first column. Much comment has been removed from the subroutine header.

```
***************************************************************
      SUBROUTINE SGBMV ( TRANS, M, N, KL, KU, ALPHA, A, LDA, X,
     $                   INCX, BETA, Y, INCY )
*     .. Scalar Arguments ..
      REAL               ALPHA, BETA
      INTEGER            INCX, INCY, KL, KU, LDA, M, N
      CHARACTER*1        TRANS
*     .. Array Arguments ..
      REAL               A( LDA, * ), X( * ), Y( * )
*
*     .. Parameters ..
      REAL               ONE         , ZERO
      PARAMETER        ( ONE = 1.0E+0, ZERO = 0.0E+0 )
*     .. Local Scalars ..
      REAL               TEMP
      INTEGER            I, INFO, IX, IY, J, JX, JY, K, KUP1, KX,
     $                   KY, LENX, LENY
*     .. External Functions ..
      LOGICAL            LSAME
      EXTERNAL           LSAME
*     .. External Subroutines ..
      EXTERNAL           XERBLA
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
*     ..
*     .. Executable Statements ..
*
*     Test the input parameters.
*
      INFO = 0
      IF     ( .NOT.LSAME( TRANS, 'N' ).AND.
     $         .NOT.LSAME( TRANS, 'T' ).AND.
     $         .NOT.LSAME( TRANS, 'C' )      )THEN
         INFO = 1
      ELSE IF( M.LT.0 )THEN
         INFO = 2
      ELSE IF( N.LT.0 )THEN
         INFO = 3
      ELSE IF( KL.LT.0 )THEN
         INFO = 4
      ELSE IF( KU.LT.0 )THEN
         INFO = 5
      ELSE IF( LDA.LT.( KL + KU + 1 ) )THEN
         INFO = 8
      ELSE IF( INCX.EQ.0 )THEN
         INFO = 10
      ELSE IF( INCY.EQ.0 )THEN
         INFO = 13
      END IF
      IF( INFO.NE.0 )THEN
         CALL XERBLA( 'SGBMV ', INFO )
         RETURN
      END IF
*
*     Quick return if possible.
*
      IF( ( M.EQ.0 ).OR.( N.EQ.0 ).OR.
     $    ( ( ALPHA.EQ.ZERO ).AND.( BETA.EQ.ONE ) ) )
     $   RETURN
```

```
*
*     Set  LENX  and  LENY, the lengths of the vectors x and y, and
*     set up the start points in  X  and  Y.
*
      IF( LSAME( TRANS, 'N' ) )THEN
.
^        LENX = N
^        LENY = M

      ELSE
         LENX = M
         LENY = N
      END IF
      IF( INCX.GT.0 )THEN

^        KX = 1

      ELSE
         KX = 1 - ( LENX - 1 )*INCX
      END IF
      IF( INCY.GT.0 )THEN

^        KY = 1

      ELSE
         KY = 1 - ( LENY - 1 )*INCY
      END IF
*
*     Start the operations. In this version the elements of A are
*     accessed sequentially with one pass through the band part.
*
*     First form  y := beta*y.
*
      IF( BETA.NE.ONE )THEN
         IF( INCY.EQ.1 )THEN
            IF( BETA.EQ.ZERO )THEN
```

```
                DO 10, I = 1, LENY
                   Y( I ) = ZERO
   10           CONTINUE
             ELSE
^               DO 20, I = 1, LENY
^                  Y( I ) = BETA*Y( I )
^  20           CONTINUE
             END IF
          ELSE
             IY = KY
             IF( BETA.EQ.ZERO )THEN
                DO 30, I = 1, LENY
                   Y( IY ) = ZERO
                   IY      = IY   + INCY
   30           CONTINUE
             ELSE
                DO 40, I = 1, LENY
                   Y( IY ) = BETA*Y( IY )
                   IY      = IY   + INCY
   40           CONTINUE
             END IF
          END IF
       END IF
       IF( ALPHA.EQ.ZERO )
     $   RETURN
^      KUP1 = KU + 1
       IF( LSAME( TRANS, 'N' ) )THEN
*
*         Form  y := alpha*A*x + y.
*
^         JX = KX
```

```
         IF( INCY.EQ.1 )THEN
^           DO 60, J = 1, N
^              IF( X( JX ).NE.ZERO )THEN
^                 TEMP = ALPHA*X( JX )
^                 K    = KUP1 - J
^                 DO 50, I = MAX( 1, J - KU ), MIN( M, J + KL )
^                    Y( I ) = Y( I ) + TEMP*A( K + I, J )
^  50             CONTINUE
^              END IF
^              JX = JX + INCX
^  60       CONTINUE
         ELSE
            DO 80, J = 1, N
               IF( X( JX ).NE.ZERO )THEN
                  TEMP = ALPHA*X( JX )
                  IY   = KY
                  K    = KUP1 - J
                  DO 70, I = MAX( 1, J - KU ), MIN( M, J + KL )
                     Y( IY ) = Y( IY ) + TEMP*A( K + I, J )
                     IY      = IY      + INCY
   70             CONTINUE
               END IF
               JX = JX + INCX
               IF( J.GT.KU )
     $            KY = KY + INCY
   80       CONTINUE
         END IF
      ELSE
*
*        Form  y := alpha*A'*x + y.
*
         JY = KY
         IF( INCX.EQ.1 )THEN
```

```
            DO 100, J = 1, N
               TEMP = ZERO
               K    = KUP1 - J
               DO 90, I = MAX( 1, J - KU ), MIN( M, J + KL )
                  TEMP = TEMP + A( K + I, J )*X( I )
   90          CONTINUE
               Y( JY ) = Y( JY ) + ALPHA*TEMP
               JY      = JY      + INCY
  100       CONTINUE
         ELSE
            DO 120, J = 1, N
               TEMP = ZERO
               IX   = KX
               K    = KUP1 - J
               DO 110, I = MAX( 1, J - KU ), MIN( M, J + KL )
                  TEMP = TEMP + A( K + I, J )*X( IX )
                  IX   = IX   + INCX
  110          CONTINUE
               Y( JY ) = Y( JY ) + ALPHA*TEMP
               JY      = JY   + INCY
               IF( J.GT.KU )
     $            KX = KX + INCX
  120       CONTINUE
         END IF
      END IF
*
^     RETURN
*
*     End of SGBMV .
*
^     END
*
*****************************************************************
```

# Appendix C

# The Computation Environment

The nine experiments shown in tabular form were done on dawn.rice.edu (internet address 128.42.1.127), a Sun 4/490 computer. The operating system is SunOS 4.1.1. The f77 version is 1.4.

Use of the Sun 4/490 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement No. CCR-8809615 and the State of Texas contract No. 1059.

# Biographical Note

The author arrived Fathers' Day, June 17, 1956, in Vancouver, Washington to David Elvis and Evadne Kelso Ammen Hilands.

A lifelong Portland, Oregon resident, his primary education was at Ainsworth School. His secondary education was from Catlin Gabel Upper School with graduation in June 1974.

The next four years were spent at Princeton University with the B.A. in History being received in June 1978. Over the years, coursework at Portland State University has occupied his time with additional bachelor degrees in Accounting and Computer Science received in August 1990.

CMSI employed him from September 1979 through March 1981. The author entered the Masters Degree Program at the Oregon Graduate Center in the fall of 1982. Tektronix employed Bill as a Software Engineer from October 1983 through May 1988. In the fall of 1990, the author reentered the Masters Degree Program at the now Oregon Graduate Institute and graduated with a M.S. in Computer Science and Engineering.