An Experiment with Inline Substitution

Keith D. Cooper
Mary W. Hall
Linda Torczon

Department of Computer Science
Rice University
Houston, Texas 77251-1892

ABSTRACT

This paper describes an experiment undertaken to evaluate the effectiveness of inline substitution as a method of improving the running time of compiled code. Our particular interests are in the interaction between inline substitution and aggressive code optimization. To understand this relationship, we used commercially available FORTRAN optimizing compilers as the basis for our study. This paper reports on the effectiveness of the various compilers at optimizing the inlined code. We examine both the run-time performance of the resulting code and the compile-time performance of the compilers. This work can be viewed as a study of the effectiveness of inlining in modern optimizers; alternatively, it can be viewed as one data point on the overall effectiveness of modern optimizing compilers. We discovered that, with optimizing FORTRAN compilers, (1) object-code growth from inlining is substantially smaller than source-code growth, (2) compile-time growth from inlining is smaller than source-code growth, and (3) the compilers we tested were not able to capitalize consistently on the opportunities presented by inlining.

INTRODUCTION

In applying code improvement techniques, an optimizing compiler is often limited by the amount of context provided to it. Realization of this fact led compiler builders to extend the scope of their techniques from a single basic block to encompass entire procedures. Recently, the literature has seen a spate of articles on cross-procedural optimizations.[6,8,15,16,17,18,19,21,22,23] A number of these have focused on the specific optimization called *inline substitution*. This paper describes an experiment that uses commercially available compilers to assess the efficacy of inline substitution in FORTRAN.

Inline substitution is not a new idea. As early as 1966, Ershov recognized that general procedure linkages presented an opportunity for optimization.[9] Allen and Cocke described several types of procedure linkages in their 1972 catalog of optimizations; among these was *inline substitution*, also called *procedure integration* or an *open linkage*.[1] To perform inline substitution, the compiler replaces the text of a procedure call with the body of the called procedure, applying appropriate substitutions to model the effects of parameter binding.

Inlining a procedure can improve run-time performance. The increased speed derives from several distinct sources.

(1) The actual code for the procedure linkage is eliminated. This can include the code to save and restore registers, allocate new local storage and establish its addressability. Additionally, it saves the actual branch to the subroutine.

(2) In the caller, the code surrounding the call site can be improved. Analyzing and optimizing the combined procedures can yield better results than treating them separately.

(3) The code from the callee can be tailored to the context of the specific call site. This can improve constant folding, expose additional dead code, and lead to more efficient code.

It is widely believed that inlining often leads to substantial run-time improvement. Several studies have either shown or predicted improvement. Scheifler, in a study of CLU programs, demonstrated improvements in the range of five to twenty-eight percent.[20] Hecht, in his optimizer for SIMPL-T, showed a one to two percent reduction in the number of quads for a program, before optimization, and predicted further improvement from optimization.[11] Davidson and Holler, in a study of C programs, averaged twelve percent across a variety of programs and four different compilers.[8,12] (None of the compilers did significant

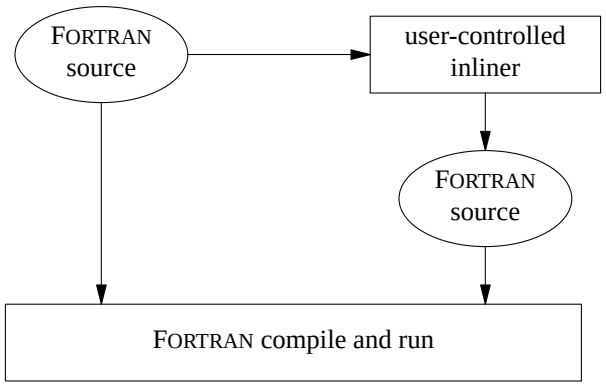

Figure 1 - Structure of the Experiment

optimization.) Richardson and Ganapathi, working with an optimizing Pascal compiler, obtained an average twenty percent improvement.[18] Huson, working inside the Parafrase system, looked at the impact of inline substitution on parallelization.[13] In reading the papers describing other studies, it appeared to us that the results varied both with the programming language and the amount of optimization performed in the compiler. To test this idea, we began an investigation into the efficacy of inlining in FORTRAN, a language with a long tradition of high quality optimizing compilers.

We began this study with two distinct goals. First, we were interested in understanding the interactions between single-procedure optimization and inline substitution. Folklore, the literature, and common sense all suggest that the value of inline substitution should be enhanced by subsequent global optimization.[11,20] In particular, prior work suggests that improvements will arise from (1) propagating constant-valued parameters through the body of the called procedure, (2) enabling code motion across the former call site, and (3) exposing more information to the register allocator.[2,18,24] Second, we were interested in discovering a set of heuristics that would guide a compiler to profitable application of inline substitution. Here, our aim was to discover a set of simple, inexpensive heuristics that accurately predicted run-time improvement.

To explore these issues, we conducted the experiment depicted in figure 1. We built a user-controlled, source-to-source inlining facility. This tool allows us to examine a FORTRAN source program, apply an inlining strategy by manually marking call sites, and automatically produce a transformed FORTRAN source that reflects the inlining. Because both the original source and the transformed source are valid FORTRAN programs, we can then compile and execute them on a variety of target machines. To serve as a basis for our study, we transformed a set of eight programs. We then compiled and ran them on five different machines, taking measurements throughout the process.

The remainder of this paper divides into three major sections and an appendix. The next section discusses the experimental methodology in more detail. The second section summarizes our measurements and discusses what these findings show about changes in source-code size, object-code size, compilation time, and execution time. The final section draws together conclusions from the preceding sections and summarizes them. The appendix discusses issues that arose in the implementation of the inlining tool itself. It provides an overview of the implementation, followed by some detailed discussion of cases where the tool cannot inline a call site.

EXPERIMENTAL METHODOLOGY

The experiment took place in two distinct phases. In the first phase, we created a set of transformed programs. In the second phase, we compiled both the original source and the transformed source on each

machine, and took measurements of the compile time and run time of each.

Phase 1

To create the transformed source, we extended tools in the programming environment to create a facility for user-directed inline substitution.[5] The facility has two distinct parts. First, we built an interface to the environment's program compiler that allows the user to specify, on a call site by call site basis, those calls that are to be inlined. The level of granularity is important; selecting individual call sites gives us maximal freedom in designing inlining strategies to evaluate. Of course, we can use our tool to simulate other granularities. For example, to simulate a procedure level specification of inlining, we can simply mark all call sites for a procedure whenever we mark one of them. The tool allows the user to navigate around the call graph, simplifying the task of finding all such call sites.

The second step of the process actually constructs the transformed source. The inliner reads the program description and the annotations that specify the inlining pattern. Using these two data structures, it constructs a new abstract syntax tree for each procedure in the transformed program. To do this, it reads in the abstract syntax tree for the base procedure and then merges into that tree the abstract syntax trees for the inlined procedures. It then prettyprints the tree to produce a text file containing the transformed FORTRAN source.

Call sites are processed in reverse topological order. Thus, all call sites inside some procedure *p* will be processed before any calls to *p* are inlined. This avoids redundant inlining. For a recursive program, no topological order exists; the tool will process the recursive cycle in arbitrary order and rely on topological order outside the cycle. None of the programs in the study are recursive.

The tools inside the environment imposed a limit on our ability to inline. The abstract syntax tree used to represent FORTRAN source is relatively large, around 1000 bytes of tree per line of source text. This imposed a practical limit of 2500 lines on the size of any single procedure. For larger trees, the performance of the tools on our workstations degraded quickly due to the large virtual working sets. Given this restriction, we inlined any call site that met one of the following criteria:

(1) It invoked a procedure of fewer than 25 source-code lines.

(2) It was the sole call to a procedure of less than 100 source-code lines.

(3) It was contained in a loop and invoked a procedure of less than 175 source-code lines.

Due to limitations imposed by features of FORTRAN or details of the tool's implementation, we could not inline some call sites. The five cases that arose are discussed in detail in Appendix A.

Initially, we examined 12 numerically intensive FORTRAN programs. As the study progressed, we dropped four of the programs due to limitations in the inlining tool, specific properties of the individual programs, or implementation limits in the surrounding programming environment. The remaining programs are of moderate size, ranging from 297 to 5979 non-comment lines of code. The eight programs included in the study are:

vortex — a particle dynamics code simulating the dynamics of a one-dimensional vortex sheet via discrete vortices;

shal64 — a simple atmospheric dynamics model based on the "shallow-water" equations;

efie304 — a code for solving electromagnetic scattering problems involving arbitrarily shaped conducting surfaces;

wanal1 — a conjugate gradient code for boundary control of the wave equation;

wave — a 2-dimensional relativistic electromagnetic particle simulation used to study plasma phenomena;

euler — a 1-dimensional spectral code modeling shock waves propagating in a tube, bursting diaphragm flows and colliding shock wave flows;

Name	Total Calls	% Inlined Stat	% Inlined Dyn	Original Source Total Lines	Original Source Total Procs	Original Source Avg Proc Length	Transformed Source Total Lines	Transformed Source Total Procs	Transformed Source Avg Proc Length
vortex	19	100	100	534	19	28	527	1	527
shal64	25	96	100	297	8	37	321	2	161
efie304	40	83	100	1248	18	69	1456	8	182
wanal1	43	84	100	1252	11	114	1751	8	219
wave	223	52	75	5979	92	65	8820	53	166
euler	31	65	57	1098	13	84	1646	4	412
cedeta	247	79	82	4269	48	89	9296	20	465
linpackd	34	44	100	417	10	42	988	4	247
Average	83	75	89	1887	27	66	3101	13	297

Figure 2 — Characteristics of Original Programs

cedeta an implementation of the Celis-Dennis-Tapia method for equality constrained global minimization; and

linpackd the classical ‘‘Dongarra’’ benchmark of LINPACK routines.

Figure 2 gives some basic information about each of the programs included in the study, in both the original and transformed state. The programs are ordered by percentage growth in text size. We use this ordering throughout the paper, in both tables and graphs, except where explicitly stated otherwise. The ‘‘Total Calls’’ column includes all calls to subroutines, functions, and libraries. It excludes calls to intrinsics. The statistics on percentage of calls inlined gives a direct measure of our heuristics’ effectiveness. The column labeled ‘‘Stat’’ gives a static count of inlined calls. This shows that, on average, seventy-five percent of the call sites were eliminated from the programs. Every program except vortex grew longer with inlining. Average procedure length grew in every case.

The column under ‘‘% Inlined’’ labeled ‘‘Dyn’’ shows the number of executions of the procedure call instruction that were eliminated. This number is expressed as a percentage of the calls executed by the original program on the same data. On average, our heuristics were able to eliminate eighty-nine percent of the dynamically executed procedure calls. Thus, the transformations eliminated the vast majority of the time spent in procedure call overhead at run time. In five of the programs, we eliminated over ninety-nine and one-half percent of the dynamic calls.

Machine	CPU	Number CPUs	Memory Size (mb)	Compiler & Version
IBM 3081d	3081	1	32	VS FORTRAN 2
MIPS	R2000	1	16	MIPS FORTRAN 2.0
Sequent S81	80386	1	80	SVS FORTRAN 3.2
Convex C240	Convex	4	512	Convex FORTRAN V5
Stardent Titan-p2	R2000	4	64	Stardent FORTRAN 2.1.1

Figure 3 — Target Machines and Compilers

Phase 2

To evaluate the effectiveness of the inlining strategy, we compiled each program, both the original and transformed version, on each of our target machines. We recorded compile time and run time for each program at each level of optimization provided by the compiler. To obtain reliable timing information, we repeated each operation multiple times.

This part of the experiment was performed on each of five machines. Three of them are scalar machines: an IBM 3081d, a MIPS 120/5, and a Sequent Symmetry S81 (used as an 80386 uniprocessor). The other two machines are vector multiprocessors: a Stardent Titan and a Convex C240. These two machines provided us with some insight into the expected results on a large class of modern supercomputers or minisupercomputers. Figure 3 shows configuration information about each of the target machines.

While this collection of machines is small, we feel that it is reasonably representative of the market today. Each machine is a well established platform that, within its market niche, is regarded as having relatively stable and solid software.

Figure 4 provides baseline data for later comparison. The first table shows execution times for each program on each machine. The measurements were made using the original source code, with no code optimization. Throughout the paper, execution times are given as percentages of these base times. The second table shows compile times for each program on each machine. In each case, the measured compilation is at

	Execution Times — Base Numbers (*hours:minutes:seconds*)				
	3081d	M120	S81	C240	Titan
vortex	17:20.9	15:10.4	49:05.8	8:21.5	38:41.2
shal64	1:37:00.5	42:20.4	2:57:36.6	11:03.1	32:27.8
efie304	16.8	14.6	1:02.9	7.9	29.1
wanal1	6:57:45.0	4:51:03.9	15:22:09.6	54:36.1	2:53:53.8
wave	1:23:18.3	1:11:55.7	2:52:40.2	25:12.8	1:11:55.5
euler	1:09.7	47.1	3:09.6	20.8	32.9
cedeta	34.1	30.8	1:17.1	14.5	41.0
linpackd	34.7	25.1	1:59.4	14.6	31.0

	Compilation Times — Full Optimization (*minutes:seconds*)				
	3081d	M120	S81	C240	Titan
vortex	2.6	10.3	7.2	13.4	39.9
shal64	1.7	7.9	5.1	7.8	21.5
efie304	7.6	52.6	13.3	26.2	1:32.0
wanal1	19.8	8:42.4	20.2	1:37.0	3:49.6
wave	35.0	2:54.7	1:05.2	2:14.6	9:25.4
euler	5.0	29.6	12.9	23.4	1:15.0
cedeta	18.1	1:39.1	2:01.1	1:12.4	8:55.2
linpackd	2.6	11.3	5.8	12.6	31.6

Figure 4 — Baseline Data For Comparisons

the highest level of optimization available on the machine (vector multiprocessor mode on the Convex and Stardent machines).

RESULTS

One of our primary goals was to gain an understanding of the efficacy of inline substitution in reasonable quality commercial FORTRAN compilers. Three key issues arise: (1) growth in object-code size, (2) growth in compile time, and (3) improved run-time efficiency. In this section, we examine the data collected in our experiments and draw some conclusions about each of these issues.

In general, the conclusions that can be drawn from our experiment present a mixed picture. To simplify the presentation, we will first discuss overall trends and conclusions, and then consider any differences between the scalar machines and the vector multiprocessors.

In the experiment, we gathered data on each compiler at each of its various levels of optimization. Unless otherwise stated, we will cite numbers for each compiler at its highest level of optimization.

Program Characteristics

The three issues listed above all arise because inlining changes fundamentally the characteristics of the user's program. As shown in figure 2, the transformed program is almost always larger than the original program, both in terms of total source lines and the size of an average procedure.

Because the procedures in the transformed source are constructed mechanically, their name spaces are likely to be larger than those of equivalent procedures written by humans – a human would re-use temporary names where the inliner will merge the two name spaces and rename to avoid conflicts. Thus, if we inline procedure q into p at two distinct call sites, the resulting code will contain two complete copies of q's local name space, rather than the single copy (or less) that a version of the same code written by a human would use.

Finally, the transformed source, with its larger procedures, should provide more contextual information to the compiler and its optimizer. Expanding the callee's body at the call site exposes a wealth of information to the compiler. Aliases that arise from parameter binding are made explicit. Constant-valued formal parameters are exposed. Definitions and uses are now susceptible to standard single-procedure analysis techniques. Loops that previously contained calls are now susceptible to analysis with standard techniques, like strength reduction, that are not applied to a loop that contains a call. (On the SVS compiler for the Sequent, this effect was particularly noticeable. The improvement in execution time due to enabling strength reduction was always larger in the inlined programs than in the original versions.) In some cases, this information could be derived by an aggressive interprocedural analysis of the original source code. In many cases, however, the program is subject to more precise analysis after inlining because of the additional information that is exposed through inlining.

During the study, we encountered a number of coding practices that made inline substitution impossible. Most of these also violate the FORTRAN standard. Procedures that rely on a default assumption of static allocation for all local variables was the most common problem encountered. In general, these variables appeared in DATA statements. Explicitly passed array dimensions that varied across the invocations of call sites was a second common problem. In most cases, this arose in connection with an array used to provide temporary storage. (The lack of dynamic allocation has given rise to this style of programming.) We encountered several call sites where actual parameters and their corresponding formals had different types. Finally, two of the programs contain jumps into loops from outside the loop. The programs containing these problems were either corrected or removed from consideration in the study.

Object-Code Size

Most discussions of inline substitution include a cautionary word or two about growth in the size of the program's executable image. Inlining can increase object-code size. This can have a negative effect on the program's execution time in several ways. A general increase in object-code size might lead to a larger working set size. This, in turn, could cause thrashing in a demand paged virtual memory system. Increasing the size of an individual loop might result in an instruction cache overflow.

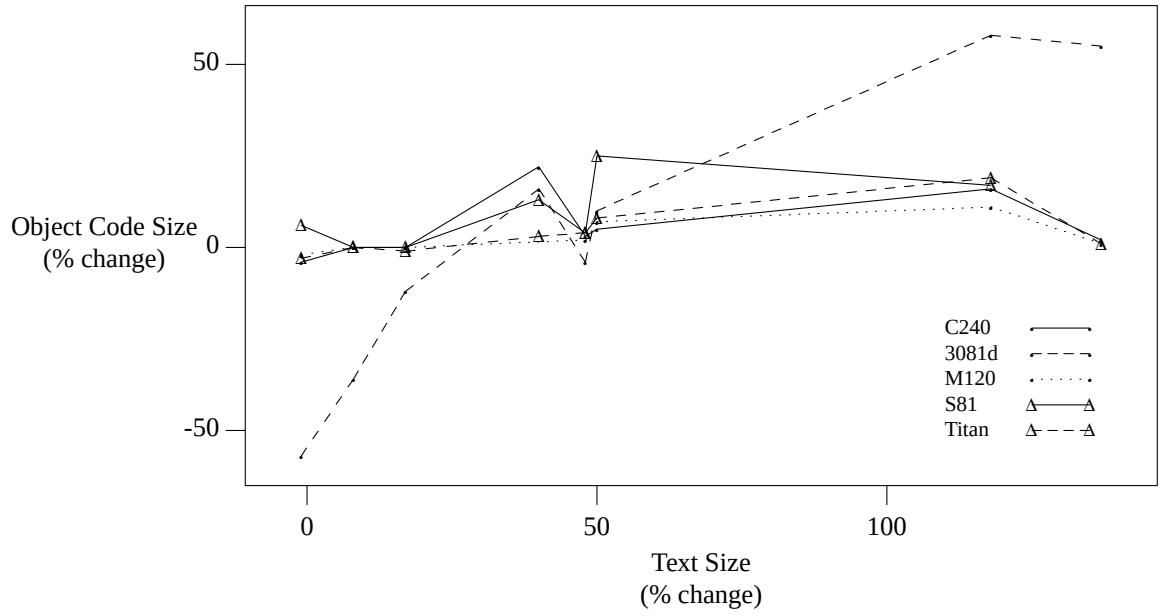

Figure 5 – Object Size versus Source-Text Size

We found no obvious evidence of either thrashing or instruction cache overflow. The only thrashing we encountered was at compile time with one of the compilers. Our previous experience suggests that expanding a loop to the point where it overflows the instruction cache has a dramatic effect on execution time; we saw no striking degradations in execution speed. Measuring other effects of object-code growth on running time is a subtle issue, well beyond the scope of our experiment. In her thesis, Holler examined both virtual memory and cache effects that result from inlining; she saw little change in the paging or cache behavior of the programs in her study.[12] Hwu and Chang report improved instruction cache behavior from inlining.[14]

Even though we saw no striking degradation in run-time speed, object-code growth is still of interest. It can change compile times. It can radically change the compiler's memory use. Furthermore, it provides some insight into the extent to which post-inlining optimization can tailor the executable code. The arguments presented in the introduction suggest that optimization should moderate object-code expansion. Those improvements have two sources: eliminating unneeded code and generating specialized code. The former directly shrinks the object code. The latter is expected to eliminate control flow, expose additional dead code, and increase the effectiveness of techniques like common subexpression elimination and constant folding. These, in turn, should lead to some reduction in object-code size.

Figure 5 plots object-code growth against source-text growth for the eight programs. It shows the change in object-code size, for full optimization, between the original and inlined programs. (For the Convex and Stardent machines, both figures 5 and 6 are based on compilations for a scalar, uniprocessor configuration – no attempt to use vector or parallel hardware.) While the results varied with both program and compiler, none of the compilers exhibited linear growth in object-code size. Four of the five compilers exhibited average growth of less than six percent, while the fifth (the SVS FORTRAN compiler) averaged less than ten percent. The MIPS compiler showed minimal variation; its growth numbers all fall within eleven percent of the original code. The largest variation came with the VS FORTRAN compiler; it showed a range from negative fifty-seven percent on vortex to fifty-eight percent on cedeta. Almost all of the individual data points lie in the range between negative five and twenty-five percent growth.

	Text	Change in Object-Code Size %									
		3081d		M120		S81		C240		Titan	
	Size %	orig	inl'd	orig	inl'd	orig	inl'd	orig	inl'd	orig	inl'd
vortex	-1	-17	-28	-3	-3	-6	6	0	-2	0	0
shal64	8	-25	-35	-1	-2	0	0	-2	-2	0	*
efie304	17	-19	-23	-6	-7	0	0	-2	-2	-3	-3
wanal1	40	-36	-36	-10	*	0	-6	-2	0	-7	-7
wave	48	-21	-25	-8	-8	-1	-1	0	0	0	0
euler	50	-22	-27	-4	-13	0	-3	0	-2	0	-3
cedeta	118	-21	-26	-13	-24	11	9	-1	-1	-2	-2
linpackd	137	-17	-25	-1	-9	-5	*	0	0	0	0
Average	52	-22	-28	-5	-9	-1	1	-1	-1	-2	-2

Figure 6 — Effect of Optimization on Object-Code Growth

This suggests that optimization did indeed mitigate growth in object code. To examine this issue in more detail, we computed the reduction in object-code size that results from full optimization. Improvement was calculated as the difference in size between unoptimized and optimized code divided by the unoptimized code size. This is a simple measure of the optimizer's effectiveness at reducing code size. To understand the interaction with inlining, we compared the numbers for the original programs with those for the inlined programs. Figure 6 gives these numbers. For each compiler, the left column shows the change in object-code size for the original program and the right column shows the change for the inlined program. An asterisk indicates that the compiler did not successfully translate this version of the program. When a program did not compile on some machine, we discarded that program when computing averages for that compiler. We use this convention throughout the paper.

The ability of the individual compilers to reduce object-code size through optimization differs greatly. The Convex and Stardent compilers show average decreases in the range of one to two percent. The MIPS compiler produced decreases of five and nine percent. The VS FORTRAN compiler does better; it produces average decreases between of twenty-two and twenty-eight percent. The SVS compiler exhibits code growth as a result of optimization in three cases. To isolate the problem, we compared object-code sizes generated by different settings of the optimization flag. On the inlined version of vortex, the problem first appears when global register allocation is enabled. On both versions of cedeta, the increased size appears when, according to the compiler manual, only constant folding is enabled.

The general trend in the data in figure 6 suggests that decreases in object-code size as a result of optimization are greater for the inlined version of the program than for the original version. With the exception of the SVS compiler, all compilers showed a greater percentage decrease in object-code size in the inlined version for at least one program. The differences are very small for the Convex and Stardent compilers, but are significant for the IBM and the MIPS compilers. Comparing the thirty-seven pairs of entries in the table, there are seventeen pairs where the reduction for the inlined program is greater than for the original program. There are only three where the reduction is greater for the original program, and two of these occurred with the SVS compiler. Three pairs are excluded from the count because the numbers for the inlined versions were unavailable.

Taken as a whole, these numbers suggest that the object-code growth resulting from inlining is manageable. Given the large percentage of dynamic calls eliminated by our heuristic, it appears that, in most situations, the compiler can eliminate the majority of calls, and, hence, most procedure call overhead, without a substantial increase in object-code size.

Compile Time

A second argument used against inline substitution is the potential for dramatic increases in compile time. In practice, compilers use some algorithms that have non-linear asymptotic complexities. This raises the concern that inlining can lead to unacceptable increases in compile time.

Figure 7 plots increase in compile time against increase in program size. All compilations are shown with full optimization enabled. From the plot, it appears that euler has some fundamentally different characteristics from a compiler's viewpoint. Each compiler showed a large jump in growth of compile time between wave and euler. The times for cedeta and linpackd appear to better fit the trend established without euler.

Most of the compilers had fairly stable compilation times. Compile time on the Convex never grew faster than source-text size. The Stardent and SVS compilers each had one program where growth in compile time exceeded growth in source text. The VS FORTRAN compiler had mixed results; its compile-time growth exceeded source-text growth half of the time. (On the other hand, it alone showed substantial decreases in object-code size!) Finally, the MIPS compiler appears to be extremely sensitive to average procedure size. It alone showed consistent superlinear growth in compile time.*

The performance of both the Convex and Stardent compilers is particularly satisfying since both perform sophisticated data-dependence analysis to support their vector and parallel hardware. Despite the costs associated with this analysis, neither compiler saw extraordinary growth in compile time when dependence

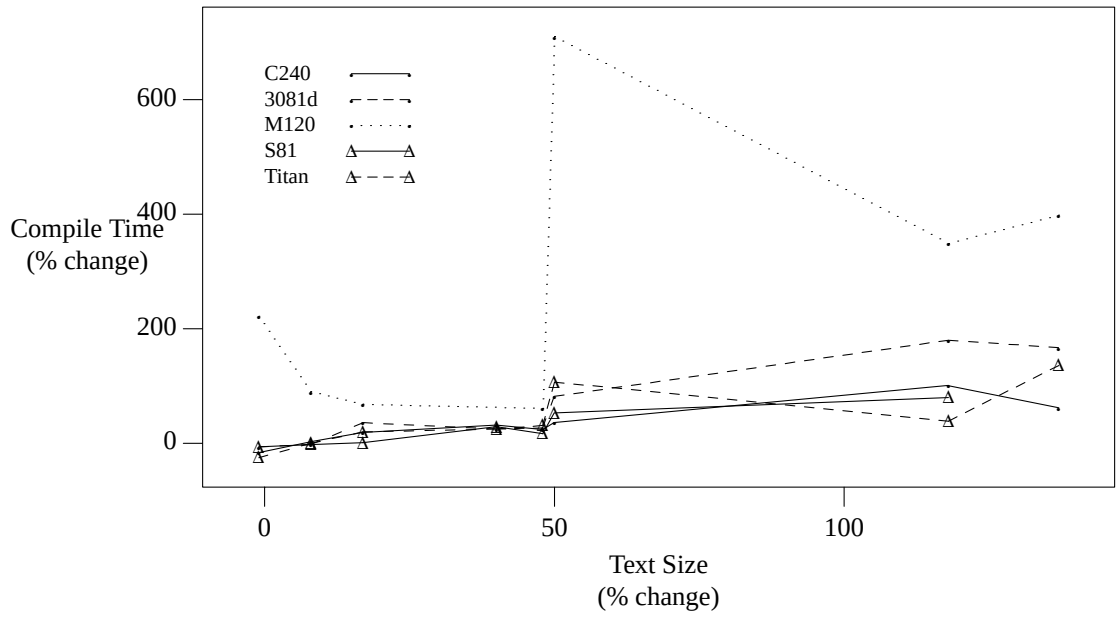

Figure 7 – Compile Time versus Source-Text Size

* The MIPS compiler also showed a surprising growth in working set size for compilation. To obtain timings that were not dominated by paging overhead required a machine with forty-eight megabytes of memory. Running the compiler as the only job on a machine with sixteen megabytes of memory resulted in radical increases in compile time. For example, wanal1 *before inlining* required over ninety five hours of wall time to compile. On the forty-eight megabyte system, that compilation required eight minutes and forty-two seconds.

analysis was enabled. While the numbers in figure 4 suggest that the Stardent compiler is somewhat slower than the other compilers in the study, the speed problem shows up when compiling the original source without optimization for a uniprocessor machine. This strongly suggests that it is not related to either optimization or dependence analysis.

Execution Time

Given that compilers can mitigate the potential for explosion in object-code size and compile time, one key question remains. Can compilers capitalize on the opportunities presented by inlining to improve the actual execution time of programs? Figure 8 shows the overall effectiveness of inlining by plotting the change in execution times for each combination of program and machine. Figure 9 plots the change in execution time as a function of change in compile time. In both figures, the times shown for the Convex and Stardent machines are for multiprocessor vector execution.

Figure 10 gives the raw data from which these plots are derived. In figure 10, the averages for a compiler were computed by completely discarding any program that the compiler failed to successfully translate.

Scalar Results

In figure 8, it is clear that there is no real trend, either by compiler or by program. The results are mixed, with many instances each of improvement and of degradation.

- The IBM compiler had the most stable performance, across the board. It produced the best improvement of the three scalar machines, sixteen percent on cedeta. There were two cases where inlining caused it to produce slower code; in each case the degradation amounted to less than one percent of the execution time. In general, the IBM compiler was able to improve the code after inlining.
- The MIPS compiler exhibited relatively stable performance. It produced little improvement or degrada-

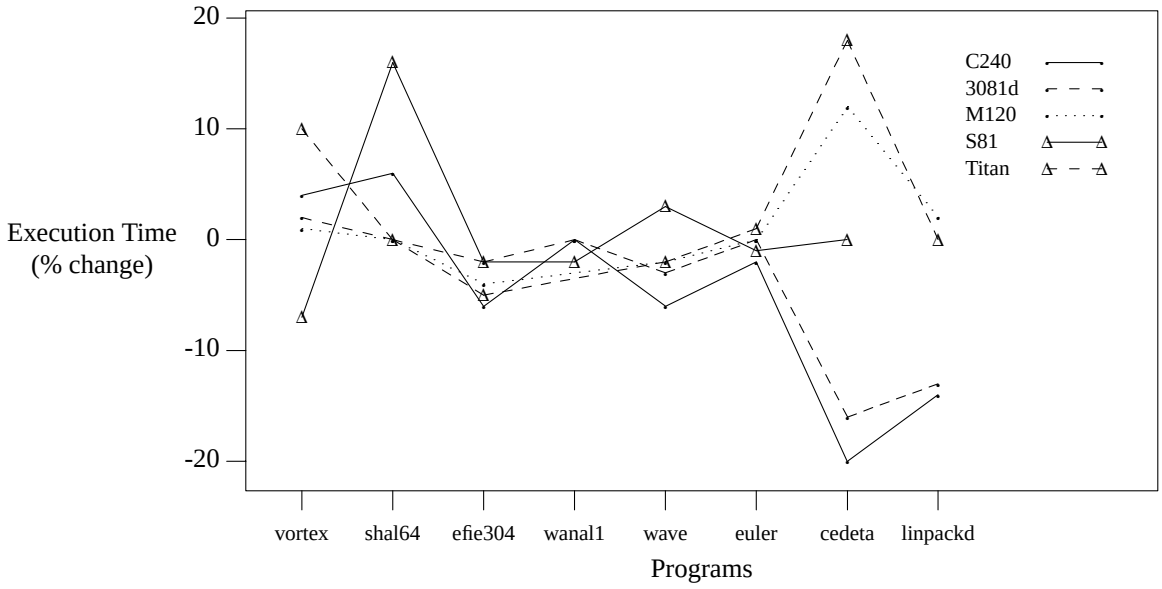

Figure 8 – Change in Execution Time from Inlining

tion as a result of inlining, with one exception being a twelve percent degradation on cedeta.*

- The SVS compiler had mixed results — four wins and two losses. It produced the single largest degradation among the scalar machines, fifteen percent on shal64.
- The Convex compiler, targeting a uniprocessor vector configuration, profited from the transformation. Six of the programs showed improvement; the two degradations were both small. It showed the largest overall improvement, twenty percent on cedeta.
- The Stardent compiler, targeting a uniprocessor vector configuration, showed little improvement from inlining. Its best improvement was five percent on efie304. Its worst degradation was eighteen percent on cedeta.

There does not appear to be a clear trend in the data, either by compiler or by program. Overall, wins outnumber losses. However, the vast majority of the wins fall in the range between infinitesimal improvement and five percent.

Figure 9 suggests that increased compile time also fails to predict improved execution time. Three of the compilers get marginally better results at the high end of the compile-time scale; the MIPS compiler often fails to get back to the original program's execution time.

These results provide insight into another issue: the relative importance of procedure call overhead. The transformation process eliminated most procedure calls; in five of the programs, we eliminated over ninety-

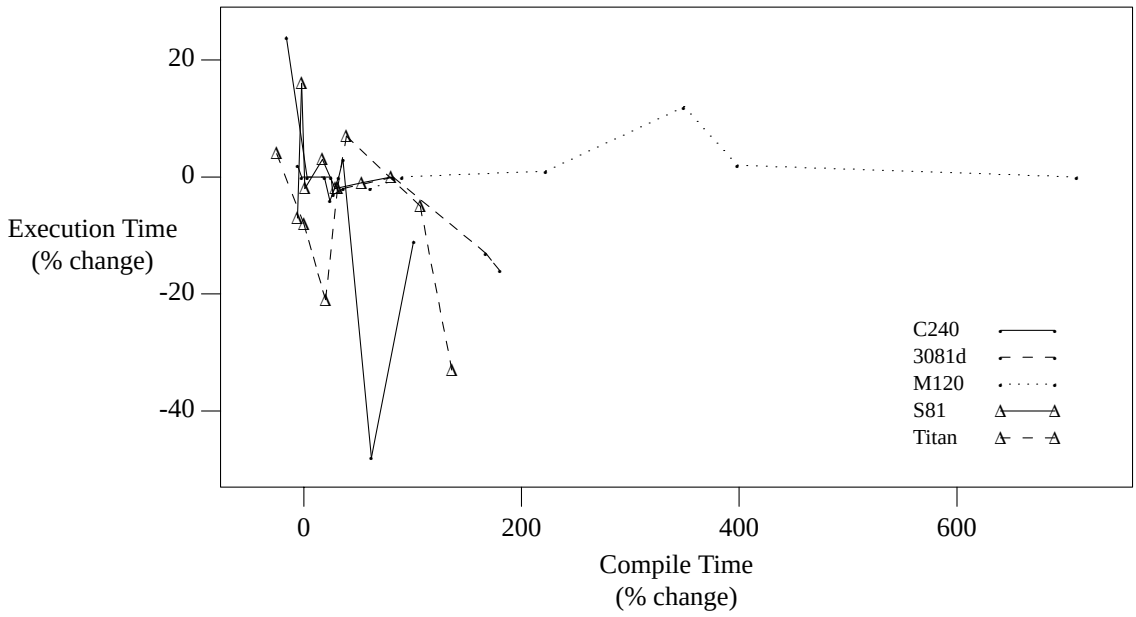

Figure 9 – Change in Execution Time versus Compile Time

* At its highest level of optimization, the MIPS compiler performs interprocedural register allocation. Because the effects of interprocedural register allocation could mask the effects of inlining, the numbers reported for the MIPS are without interprocedural register allocation. In most cases, the performance change after interprocedural register allocation was negligible, with four exceptions: the inlined version of cedeta exhibited a fifteen percent improvement due to interprocedural register allocation; both versions of linpackd were slowed down by about five percent after interprocedural register allocation; and the inlined version of efie304 was slowed down by twenty-seven percent.

nine percent of the executed calls! This did not lead to consistently faster execution.

Two conclusions are possible. Either call overhead is a negligible part of program execution, or program properties introduced by inlining resulted in decreased code quality that masked the savings in call overhead. We believe the latter statement. To reinforce this point, look at the numbers for unoptimized execution of the transformed source in figure 10. The three scalar machines show little improvement from eliminating calls; the Convex and Stardent compilers do a somewhat better job. This suggests that the other effects in the compiler are hiding the call overhead savings. For example, we know from experience with our own compiler that inlining often increases register pressure.

Parallel Results

Two of the machines are vector multiprocessors, the Stardent Titan and the Convex C240. They both have stable restructuring compilers — compilers that attempt to automatically discover opportunities for vector and parallel execution. Both compilers use techniques based on data-dependence analysis. Thus, they provide us with the opportunity to examine the issue of inlining as an aid in automatic parallelization.

	IBM 3081d			MIPS M120/5			Sequent S81		
	uopt + inl	opt	opt + inl	uopt + inl	opt	opt + inl	uopt + inl	opt	opt + inl
vortex	100	65	66	102	68	69	99	82	76
shal64	100	19	19	100	33	33	100	38	44
efie304	97	46	45	88	50	48	100	92	90
wanal1	100	16	16	100	34	*	100	46	45
wave	101	37	36	100	41	40	100	67	69
euler	100	34	34	103	30	30	100	79	78
cedeta	96	61	51	115	43	48	108	85	85
linpackd	193	39	34	171	44	49	137	67	*
Average	110.9	39.6	37.6	111.3	44.1	45.3	101.0	69.9	69.6

	Convex C240					Stardent Titan				
	uopt + inl	vect	vect + inl	para	para + inl	uopt + inl	vect	vect + inl	para	para + inl
vortex	93	26	27	21	26	63	60	66	28	29
shal64	100	18	19	6	6	*	32	32	12	11
efie304	94	64	60	40	40	87	79	75	96	76
wanal1	117	19	19	7	7	101	*	13	*	*
wave	101	32	30	26	25	98	63	62	60	59
euler	99	51	50	69	71	93	84	85	200	190
cedeta	92	89	71	160	143	101	101	119	168	179
linpackd	109	21	18	27	14	92	20	20	21	14
Average	100.6	40.0	36.8	44.5	41.5	89.0	67.8	71.2	97.5	91.2

Figure 10 — Changes in Execution Time

In general, the compilers were able to generate better code for the transformed source text. For both compilers the difference was just under seven percent. Looking at specific programs, each compiler had two cases where the transformed code ran slower than the original. On the Convex, vortex ran twenty-four percent slower; on the Stardent cedeta slowed down by seven percent. The remaining cases range from minimal differences to major improvements — forty-eight percent for linpackd on the Convex and thirty-three percent for linpackd on the Stardent.

This suggests that seven of the programs could be improved by inlining. (Both compilers failed to improve vortex.) Unfortunately, both compilers failed to generate parallel loops in situations where inlining appears to expose the parallelism in a loop. After looking at the diagnostic information provided by each compiler, we became interested in the question: what would be required to enable these compilers to capitalize on the opportunities presented by inlining? To shed some light on this question, we applied, by hand, several transformations to the transformed sources. In particular, applying constant propagation and loop invariant code motion, and removing unreachable code, between the inliner and the manufacturer's compiler, increased the number of parallel loops discovered.[10]

Both compilers had problems deciding when parallel execution was profitable. Compare the execution times of the original code in uniprocessor vector mode against the results for multiprocessor vector mode. On the Convex, three of the programs ran slower as parallel programs than as uniprocessor programs. On the Stardent, four programs exhibited this behavior. This was particularly bad for both euler and cedeta. It appears that both compilers assume that parallel execution is profitable unless it can discover a tight bound on loop iterations that conclusively proves otherwise.

Implications

Overall, the five compilers were unable to consistently capitalize on the opportunities provided by inlining. Recall, from figure 2, that our inlining strategy eliminated nearly all procedure calls — on average, eighty-nine percent of the procedure calls executed in the original source were eliminated in the transformed source. The expected run-time savings do not appear during execution, as shown in figure 10. We did not find a consistent and appreciable improvement in run-time speed. Several factors appear to account for this phenomenon.

First, inlining often increases register pressure. Both vortex and euler contain call sites that pass global variables from several COMMON blocks as actual parameters. In the original code, the actual parameters in the callee have one-register names. After inlining, they have two-register names, a base address plus an offset. On the MIPS machine, the transformed version of euler executed two percent more loads and stores — that is, one million, nine hundred thousand more loads and stores — than the original version. Those cycles mask other improvements. On those compilers where the application of global register allocation is controlled by a user-supplied switch, the results of global allocation were mixed.

Second, the inlined code may have fundamentally different properties. For example, in linpackd on the MIPS, inlining resulted in a six percent degradation in run-time speed. Closer investigation showed that the number of floating point interlocks per floating point operation rose from 0.62 to 1.1 after inlining. Since the program executes almost twenty million floating point operations, that factor is significant.

Closer investigation revealed that the call from dgefa to daxpy passes two regions inside a single array in two distinct parameter positions. Unfortunately, complicated range analysis would be required to show that the regions do not overlap. Thus, when the call is inlined, the body of the key loop in daxpy is a single statement that both reads two locations and writes one location inside the array. Without complex analysis, the compiler must assume that the references can overlap. This introduces the data interlocks that we observed.

Finally, design decisions that are justifiable for code written by a programmer may have unforeseen consequences when applied to code generated by the inliner. For example, many compilers place a hard upper limit on the number of variables subject to data-flow analysis; typically, they summarize all remaining vari-

ables with a single bit position.* With inlined code, the growth in name space as a function of procedure length is probably much greater than in human-generated code. This may exacerbate the deleterious effects of summarization in data-flow analysis.

On the two parallel machines, we observed several cases where the decision to run a loop nest in parallel resulted in a disastrous slowdown. This happened with both euler and cedeta. It appears that both compilers assume that the parallel loop is profitable in those cases where the number of iterations is unknown. A simple strategy would allow the compilers to avoid these major slowdowns: in those cases where the number of iterations is unknown, generate both the uniprocessor and multiprocessor versions of the loop and insert a simple run-time test. Our results suggest that the growth in object code may be small. If doing so eliminated these slowdowns while allowing the compilers to aggressively generate parallel code, it would almost certainly make up for the minor object-code growth.

CONCLUSIONS

We studied the effects of inlining on object-code size, compile time, and execution time using five commercial FORTRAN compilers. We found that

(1) Object code grew more slowly than source code. The compilers were able to eliminate a larger percentage of the object code from the inlined program than from the original program.

(2) The Convex, Stardent, and SVS compilers exhibited fairly stable compile times. Their compile times, in general, grew more slowly than source-text size. The VS FORTRAN compiler had mixed results, and the MIPS compiler showed consistent superlinear growth in compile time.

(3) None of the compilers exhibited a consistent improvement in execution time as a result of inlining, even on the five programs where virtually all of the procedure calls were eliminated. Secondary effects of inlining often overcome the savings from eliminating call overhead.

The first two conclusions are positive. In general, the compilers performed well. Unfortunately, the five commercial compilers we tested are unable to consistently capitalize on the opportunities presented by inlining.

Inlining should be profitable. The weaknesses that the inlined programs exposed should be corrected. Other researchers, like the IMPACT-I group, are working to build compilers that take consistent advantage of inline substitution.[15] A major thrust of our own research program is to address these problems. We are investigating techniques to improve the predictability and robustness of register allocators based on the graph coloring paradigm.[3] We are also working on a goal-directed approach to inlining – one that only applies the transformation to enable another high-payoff optimization.[4]

ACKNOWLEDGEMENTS

Control Data Corporation, IBM Corporation, and the National Science Foundation have all supported this work. Many people have encouraged this work, helped us obtain access to the machines used in the study, and helped us understand the results. Our thanks go to Steve Wallach, Ron Gray and Randy Mercer at Convex; Randy Scarborough at IBM; and Alex Wu and Fred Chow at MIPS; Gary Graunke and Joe Bonasero at Sequent; and Randy Allen at Stardent. Our colleagues at Rice have provided us with valuable feedback. Rick Stevens and his staff at Argonne provided us with time and access to several machines that belong the ACRF. Sunil Bhargava, Preston Briggs, Horace Flatt, Kevin Harris, Ken Kennedy, Peter Markstein, and Tom Spillman all encouraged this work. Finally, the referees made a number of insightful suggestions that improved the final paper.

REFERENCES

* This strategy makes all bit-vector operations have a known, uniform length. It allows the compiler writer to hard code the bit-vector operations, and eliminate the loop based on bit-vector length on each operation.

(1) F.E. Allen and J. Cocke. ‘‘A catalogue of optimizing transformations’’, in *Design and optimization of a compiler* (J. Rustin, *ed.*), Prentice-Hall, 1972.

(2) J.E. Ball. ‘‘Predicting the effects of optimization on a procedure body’’, *Proceedings of the SIGPLAN 79 Symposium on Compiler Construction, SIGPLAN Notices* 14(8), August 1979.

(3) P. Briggs, K.D. Cooper, and L. Torczon. ‘‘Aggressive live range splitting’’, Department of Computer Science Technical Report 90-146, Rice University, Houston, TX, November, 1990.

(4) P. Briggs, K.D. Cooper, M.W. Hall, and L. Torczon. ‘‘Goal-directed interprocedural optimization’’, Department of Computer Science Technical Report 90-147, Rice University, Houston, TX, November, 1990.

(5) A. Carle, K.D. Cooper, R.T. Hood, Ken Kennedy, Linda Torczon, and S.K. Warren. ‘‘A practical environment for scientific programming’’, *IEEE Computer* 20(11), November 1987.

(6) F.C. Chow. ‘‘Minimizing register usage penalty at procedure calls’’, *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23(7), July 1988.

(7) R. Conradi, D. Svanaes. ‘‘FORTVER — a tool for documentation and error diagnosis of FORTRAN-77 programs’’, University of Trondheim, Division of Computer Science Technical Report, January 1985. (*presented at IFIP WG2.4 meeting, Bonn, January 1985*)

(8) J.W. Davidson and A.M. Holler. ‘‘A study of a C function inliner’’, *Software—Practice and Experience* 18(8), August 1988.

(9) A. Ershov. ‘‘ALPHA — an automatic programming system of high efficiency’’, *Journal of the ACM* 13(1), January 1966.

(10) M.W. Hall. ‘‘Improving parallelism after inline substitution’’, Department of Computer Science Technical Report 90-127, Rice University, Houston, TX, August, 1990.

(11) M. Hecht, *Flow analysis of computer programs*, Elsevier North Holland, New York, 1977.

(12) A.M. Holler. ‘‘A study of the effects of subprogram inlining’’, Ph.D. dissertation, Department of Computer Science, University of Virginia, August, 1990.

(13) C.A. Huson. ‘‘An in-line subroutine expander for Parafrase’’, M.S. Thesis, University of Illinois at Champaign-Urbana, Technical Report UIUCDCS-R-82-1118, December 1982.

(14) W.W. Hwu and P.P. Chang. ‘‘Achieving high instruction cache performance with an optimizing compiler’’, *Proceedings of the 16th Annual Symposium on Computer Architecture*, May, 1989.

(15) W.W. Hwu and P.P. Chang. ‘‘Inline function expansion for inlining C programs’’, *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 24(7), July 1989.

(16) T.P. Murtaugh. ‘‘An improved storage management scheme for block structured languages’’, Computer Science Technical Report, Williams College, Williamstown, MA, April 1989.

(17) S. Richardson and M. Ganapathi. ‘‘Interprocedural optimization: experimental results’’ *Software—Practice and Experience* 19(2), February 1989.

(18) S. Richardson and M. Ganapathi. ‘‘Interprocedural analysis versus procedure integration’’, *Information Processing Letters* 32(3), August 1989.

(19) V. Santhanam and D. Odnert. ‘‘Register allocation across procedure and module boundaries’’, *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 25(6), June 1990.

(20) R. Scheifler. ‘‘An analysis of inline substitution for a structured programming language’’, *CACM* 20(9), September, 1977.

(21) P.A. Steenkiste and J.L. Hennessy. ‘‘A simple interprocedural register allocation algorithm and its effectiveness for LISP’’, *ACM TOPLAS* 11(1), January 1989.

(22) D. Wall. ‘‘Global register allocation at link-time’’, *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), July 1986.

(23) D. Wall. ‘‘Register windows vs. register allocation’’, *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23(7), July 1988.

(24) M. Wegman and F.K. Zadeck. ‘‘Constant propagation with conditional branches’’ *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, January 1985.

Appendix A

IMPLEMENTATION ISSUES

Although our principal goal in performing this experiment was to increase our understanding of the interactions between inline substitution and global optimization, a subtask of the experiment was to construct the tools required to produce the transformed sources. Building the tools was a learning experience. In this section, we describe some of our lessons.

Implementation Overview

As discussed earlier, we built the inliner as part of the program compiler in the programming environment. Inline substitution is implemented as a source-to-source transformation. It is performed on an abstract syntax tree (AST) representation of the program. The first step in inlining is a check to ensure that inlining is possible; this test checks for five separate conditions that might make the transformation illegal in our system.

After the compiler has proven that a particular substitution is legal, the actual transformation takes place in two phases. The first phase iterates over all of the symbols appearing in the called procedure, assigning unique names to its local variables and labels and building a symbol table of the new names hashed on their original names. The second phase walks the AST, updates the names of all variables and labels, and moves the actual statements from the body of the callee to their new locations in the caller.

During the first phase, all global variables retain their original names. Common block definitions are added to the caller as needed. In the absence of name conflicts, local variables and labels retain their original names. If a conflict arises, local variable names are textually altered, and new labels are generated by incrementing the label number until a unique label is generated.

Formal parameters are renamed to their corresponding actual parameters from the call site. Two cases of interest arise: expression-valued parameters and array parameters. If the actual is an expression rather than a variable, the inliner generates a temporary and inserts an assignment before the procedure body to evaluate the expression and save its value. Constant-valued expressions are a special case. Unless the formal parameter appears on the left-hand-side of an assignment, no temporary is generated. (Although the FORTRAN standard forbids assignments to constant-valued formals, it happens often enough in real programs to warrant handling it.)

The most complicated mapping of formals to actuals occurs with array-valued actuals. The mapping relies on the fact that Fortran uses column-major storage. For an n-dimensional formal f, the inliner currently requires that the actual must match the formal in each of its first n-1 dimensions. The actual may contain more dimensions than the formal. Alternatively, the actual parameter may specify a location other than the first element of the array. This results in passing a subsection of the actual array.

The second phase of the inliner renames all references to variables and labels based on the translation table built in the first phase. Next, it moves the statements that comprise the callee's body into an appropriate position in the caller. The executable statements are moved together.

- For a subroutine call, these statements simply replace the CALL statement.
- For a function call, they are moved immediately above the statement containing the call site. A temporary is created to hold the function's return value.

If the statement containing the call site has a label, the label is moved to the beginning of the inserted code.

Declarations for the variables from the callee are inserted with those of the caller. COMMENT, IMPLICIT,

and ENTRY statements are removed.*

A number of minor issues arise in translating CALL and RETURN statements.

- RETURN statements are converted to GOTO statements that refer to a label immediately following the inlined procedure body.
- If the callee is a FUNCTION, then an assignment to the temporary designated for the return value is generated at each RETURN statement.
- If the procedure uses FORTRAN's alternate return mechanism to change the return address, then the RETURN statement is translated to a GOTO that targets the appropriate label-valued parameter.
- If the call site specifies an entry point rather than a procedure, the inliner must inline the entire procedure and add a jump to the location of the ENTRY statement. (That part of the callee's body preceding the ENTRY statement must remain intact in case the subsequent code jumps back to it.)

The inliner handles each of these appropriately.

Figure 11 shows an example which may clarify some of the issues. The placement of declarations and data statements ensures that meaning is preserved. Rather than attempt to resolve implicit typing conflicts between procedures, the inliner creates declarations for all implicitly typed variables from the callee. The declarations and data statements originating from *b* are sandwiched between *a*'s declarations and data statements so that they remain in their original order. Any new declarations for *b* precede *b*'s declarations to ensure that dependences between declarations are satisfied. An example of this might be a PARAMETER statement that is used to dimension an array.

Failure to inline

Five situations can prevent the inliner from constructing a valid FORTRAN program. Some of these are fundamental problems that any inlining tool will encounter; others are idiosyncratic to our implementation.

(1) Inlining replaces the formal parameters of the callee with the actuals from the call site. For this to be meaningful, the formal and its corresponding actual must agree in type. If the formal and actual have different types, the transformation cannot be performed. A program that contains such a call

---

```
      subroutine a                    subroutine a
      a's declarations                a's declarations
      a's data statements             new declarations for b
       ...                            b's declarations
10    if (b(x)) then                  b's data statements
       ...                            a's data statments
                                       ...
                                  10   begin body of b
                                       ret = return value for b
                                       goto 1
                                   1   if ret then
                                       ...
```

Figure 11 — An Example of Inlining a Call Site

---

* Comments are removed to limit the growth of the inlined program. The implicit statements are removed because they may conflict with implicit typing in the caller. Instead, type declarations are added when they do not already exist for every variable in the called procedure including those temporaries generated by the inliner. Finally, entry statements are removed so as to not have multiply defined subroutines.

site does not conform to the FORTRAN standard.

Nonetheless, few FORTRAN compilers enforce this restriction. We found such call sites in our study; others have reported similar results.[7] Some, quite obviously, had been carefully crafted to achieve specific behavior. For example, in euler, one call site passes an array of reals to a formal that is a complex array, relying on the standard's requirement that complex numbers must be implemented as pairs of reals.

(2) Another class of problems arises when an actual parameter and its corresponding formal parameter are declared with different dimension information. These problems manifest themselves in two different ways: the number of elements in each dimension and the number of dimensions.

The inliner requires that the sizes of the first $n-1$ dimensions specified in the callee be identical to the corresponding dimensions in the caller. While more complex mappings can conform to the storage, such remapping can introduce a level of complexity into all the subscript expressions.

If the actual parameter has more dimensions than its corresponding formal, the inliner translates references to the formal into references to the actual in a straightforward manner. (This case is the classical FORTRAN dimension reduction at a call site.) If the actual parameter has fewer dimensions than the formal parameter, the references can still be translated. However, the resulting references can be substantially more complex than their original forms.

In both cases that remap storage, changes to the size of an inner dimension or an increase in the number of dimensions, we decided to disallow inlining.

(3) To allow for safe application of anchor pointing, the transformer does not inline any call site that appears in the second term or subsequent terms of a boolean expression. This optimization, also called a *short circuit* optimization, cuts short the evaluation of an expression as soon as its value has been fully determined. In the expression (a.and.b), the compiler can avoid evaluating b if evaluating a yields false.

Strictly speaking, any program whose behavior changes under this transformation does not conform to the FORTRAN standard. Nevertheless, we felt that the inliner should preserve the original program's behavior in this case. To transform the source in a way that preserves its original behavior under inlining and anchor pointing requires the introduction of additional control flow operations. To simplify this situation, we elected to disallow inlining of any call except the leftmost in a boolean expression.

(4) A call site that invokes a procedure-valued parameter cannot be inlined unless the compiler can, through interprocedural analysis, determine that the procedure variable has a single value across all invocations of the caller. If the variable has multiple values, the transformer cannot replace the call site with the body of any single procedure.

Even with this restriction, the transformer handles the most common use of procedure-valued parameters in scientific FORTRAN programs. Programmers often pass into a procedure the name of another procedure that implements the mathematical function being manipulated. This simplifies applying the overall algorithm to different functions, but retains the property that, within a single compilation, the procedure-valued parameter has a single consistent value. In our experience, this is, by far, the most common use for procedure-valued parameters.

(5) FORTRAN provides a mechanism to declare a variable static, the SAVE statement. The value of such a variable is preserved across invocations of the procedure in which it is declared. To preserve the correct behavior of these variables requires the introduction of a generated COMMON block in every instance of the procedure body. To date, we have not implemented this transformation. Therefore, the tool will not inline a procedure that contains a SAVE statement.

As shown in figure 2, even with these restrictions, we were able to eliminate most of the dynamically executed procedure calls.