A Study of the Effects of Subprogram Inlining

Anne M. Holler

Computer Science Report No. TR-91-06
March 20, 1991

A Study of the Effects of Subprogram Inlining

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Anne M. Holler

May 1991

Abstract

This dissertation examines subprogram inlining, a widely-known code optimization technique whose effects were not well understood prior to this study. For a set of test programs on several computer systems, the sizes and runtimes of executable files produced using a tool that automatically substitutes subprograms inline were compared with those of versions in which call sites were not expanded inline. Issues raised by this experiment's results led to the formulation of equations hypothesized to estimate the execution time performance of noninlined and inlined versions of a program. The accuracy of the equations' description of inlined program execution time behavior was demonstrated on four computer systems. Investigation with the equations provided insights of interest to machine and compiler designers.

Using the equations, knowledge about the influence of certain factors on the speed of inlined code was gained. Contrary to a number of published reports in the literature, the increased size of inlined code was not found to affect its execution time performance on a set of demand-paged virtual memory machines. In addition, a modest improvement in the caching and paging behavior of test programs' inlined versions was noted. Situations in which inlining was not beneficial, which were associated with its interactions with register allocation, were characterized.

Furthermore, in connection with developing the inlining tool, discoveries were made: an efficient way of organizing an inliner was devised, a new optimization for supplementing inlining's effects was invented, inlining issues related to particular C constructs were identified, and a novel approach to inlining recursive subprograms was conceived.

Acknowledgements

I wish to convey my deep gratitude to my faculty advisor, Jack Davidson, for his time, patience, and guidance, without which this work would never have been completed. Bob Cook's continuing interest and comments on the research were invaluable. I appreciate the suggestions made at the dissertation research proposal meeting and on previous versions of the dissertation by my doctoral committee, comprised of Professors Aylor, Batson, Cook, Davidson, and Reynolds. Discussions with members of Davidson's compiler research group, particularly Dave Whalley, Ricky Benitez, Mark Bailey, and Frank Houston, were beneficial. Dave Albrecht offered a number of insights.

This work relied heavily on the University of Virginia Computer Science Department's computing facilities, which are skillfully managed by Gina Bull, Ray Lubinsky, and Mark Smith. Carolyn Duprey, Barbara Graves, Ginny Hilton, and Tammy Ramsey helped me with the administrative issues related to the doctoral degree program. The research was supported in part by Center for Innovative Technology under Grant INF-87-003 and by National Science Foundation under Grant CCR-8611653.

Finally, I would like to thank my parents for giving me the desire to pursue goals and my grandfather for giving me the belief that I could achieve them.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Introduction to Subprogram Inlining

Subprogram inlining is a code transformation in which a subprogram call is replaced by a copy of the subprogram's body, with suitable substitutions for parameters and conflicting identifiers. The subprogram inlining transformation, when done properly and applied judiciously, can shorten a program's execution time. Examples of an inline transformation are given in the four figures that follow. Figure 1.1

Original Version	Inlined Version

`int power(num,exponent)`
`  int num;`
`  int exponent; {`
`   register int result;`
`   int cnt;`
`    if (exponent < 0)`
`        result = 0;`
`    else if (exponent == 0)`
`        result = 1;`
`    else {`
`        result = num;`
`        for (cnt=2; cnt<=exponent; cnt++)`
`            result = result * num;`
`    }`
`    return(result);`
`}`
 	
`main() {`
`   int x;`
`    x = 7;`
`    (void)printf("%d squared is %d\n",x,power(x,2));`
`    return(0);`
`}`
	`main()`
	`{`
	`  int x;`
	`    x = 7;`
	`    {`
	`     register int result;`
	`     int cnt;`
	`      {`
	`       result = x;`
	`       for (cnt=2; cnt<=2; cnt++)`
	`           result = result * x;`
	`      }`
	`      {`
	`       (void)printf("%d squared is %d\n",x,result);`
	`      }`
	`    }`
	`    return(0);`
	`}`

Figure 1.1: An Inline Transformation given in C

Original Version	Inlined Version

integer function power(num,exponent)
integer num
integer exponent
integer cnt
if (exponent .lt. 0) then
power = 0
else if (exponent .eq. 0) then
power = 1
else
power = num
do 10 cnt=2,exponent
10 power = power * num
end if
return
end
program mainprog
integer x,power
x = 7
print *,x,' squared is ',power(x,2)
stop
end
	program mainprog
	integer x
	integer result
	integer cnt
	x = 7
	result = x
	do 10 cnt=2,2
	10 result = result * x
	print *,x,' squared is ',result
	stop
	end

Figure 1.2: An Inline Transformation given in Fortran 77

Original Version	Inlined Version

program main(output);
var x : integer;
function power(num:integer;exponent:integer) : integer;
var cnt,result : integer;
begin
if exponent < 0 then
power := 0
else if exponent = 0 then
power := 1
else begin
result := num;
for cnt := 2 to exponent do
result := result * num;
power := result
end
end;
begin
x := 7;
writeln(x,' squared is ',power(x,2))
end.
	program main(output);
	var x : integer;
	cnt,result : integer;
	begin
	x := 7;
	result := x;
	for cnt := 2 to 2 do
	result := result * x;
	writeln(x,' squared is ',result)
	end.

Figure 1.3: An Inline Transformation given in Pascal

Original Version

```
        .text
        .globl _power
_power: link   a6,#-8       #allocate stack space
        movl   d7,sp@       #save register d7
        tstl   a6@(12)
        jge    L14
        moveq  #0,d7
        jra    L15
L14:    tstl   a6@(12)
        jne    L16
        moveq  #1,d7
        jra    L15
L16:    movl   a6@(8),d7
        moveq  #2,d1
        movl   d1,a6@(-4)
        jra    LY0000
LY0001: mulsl  a6@(8),d7
        addql  #1,a6@(-4)
LY0000: movl   a6@(-4),d0
        cmpl   a6@(12),d0
        jle    LY0001
L15:    movl   d7,d0         #set return register
        movl   a6@(-8),d7 #restore register d7
        unlk   a6           #deallocate stack space
        rts                 #return from subprogram
        .data1
L25:    .ascii "%d squared is %d\012\0"
        .text
        .globl _main
_main:  link   a6,#-4       #allocate stack space
        moveq  #7,d1
        movl   d1,a6@(-4)
        pea    2            #pass parameter value
        movl   d1,sp@-      #pass parameter value
        jbsr   _power       #call subprogram
        addqw  #8,sp        #delete parameters
        movl   d0,sp@-
        movl   a6@(-4),sp@-
        pea    L25
        jbsr   _printf
        moveq  #0,d0         #set return register
        unlk   a6           #deallocate stack space
        rts                 #return from subprogram
```

Inlined Version

```
        .data1
L18:    .ascii "%d squared is %d\012\0"
        .text
        .globl _main
_main:  link   a6,#-12      #allocate stack space
        movl   d7,sp@       #save register d7
        moveq  #7,d7
        movl   d7,a6@(-4)
        moveq  #2,d1
        movl   d1,a6@(-8)
        jra    LY0000
LY0001: mulsl  a6@(-4),d7
        addql  #1,a6@(-8)
LY0000: cmpl   #2,a6@(-8)
        jle    LY0001
        movl   d7,sp@-
        movl   a6@(-4),sp@-
        pea    L18
        jbsr   _printf
        moveq  #0,d0         #set return register
        movl   a6@(-12),d7 #restore register d7
        unlk   a6           #deallocate stack space
        rts                 #return from subprogram
```

Figure 1.4: An Inline Transformation given in MC68020 Assembly Language

shows the original and inlined versions[1] of a program, written in the C programming language, that uses a power function to compute the square of a number. Figure 1.2 presents the two versions coded in Fortran 77, while Figures 1.3 and 1.4 contain the versions coded in Pascal and in MC68020 assembly language respectively. The assembly code in Figure 1.4 was produced from the C example by the Sun-3 C

[1]An *inlined* version of a program is a version in which some subset of the program's call sites have been expanded inline.

compiler with its optimizer enabled; similar assembly code is produced for the Fortran 77 and Pascal examples by the corresponding Sun-3 compilers.[2] (In fact, inter-language calls between subprograms written in these three high-level languages are supported on the Sun-3 [Sun86a, Sun86b].)

The primary reason for inlining subprograms is to reduce program execution time. Inlining eliminates the overhead associated with a subprogram call. This overhead, illustrated by the commented lines in Figure 1.4, consists of four components: transferring control to the subprogram and returning from it, along with the requisite storing and retrieving of the return address; passing any parameters specified and deleting them at the appropriate point; allocating and deallocating space for the callee's local variables;[3] and saving and restoring certain machine registers used, according to the target computer system's conventions. The overhead is significant; subprogram calls and returns have been cited [PaS82] as the most time-consuming operations in a high-level language program, based on their frequency of occurrence and on the number of machine instructions and memory references they include. Despite the overhead incurred, programmers are encouraged to use subprograms for software engineering aims, such as clarity, modularity, and software reuse.

In addition to the execution time saved by eliminating subprogram call overhead, inlining can decrease program execution time by augmenting the opportunities to employ other code improvement techniques. For instance, in the examples given in Figures 1.1 through 1.4, two code improvement techniques were applied to the inlined versions of the code which could not have been applied, in general, to the original versions. First, since the formal parameter *exponent* was replaced by the constant value 2 in the inlined versions of the subprogram *power*, a technique called compile-time expression evaluation was used to map each comparison of 2 (i.e., *exponent*) with 0 into the appropriate constant value. Then, because the comparisons became constant values, a technique called dead code elimination was used to remove the statements that were conditional on constants of value false. (Further enhancement of the code is possible, as will be shown in the next section.)

---

[2]The chief difference is that, for Fortran 77, the address of each actual parameter is passed rather than its value.

[3]Fortran 77 need not dynamically allocate the space for a subprogram's locals.

Subprogram inlining can shrink the size of a program's executable code. If a subprogram that is invoked from a single call site is inlined and its callable body is deleted, then the resulting object code is smaller than the original object code, because the subprogram call overhead code has been omitted. However, inlining usually increases the size of a program's executable code. When a subprogram's body is expanded inline at more than one call site, the size of the object code grows (unless the subprogram's inlined body is shorter than the call overhead code that was located at the call sites).

Although subprogram inlining is a well-known code improvement technique, understanding of the technique has been based largely on intuitive supposition. This dissertation addresses the need for a more rigorous treatment of subprogram inlining by exploring, in detail, the answers to several questions concerning inlining. The pivotal question is: What are the effects of subprogram inlining on program behavior? Factors that determine the speed of inlined code as compared with that of noninlined code are identified and their relative influence on inlined code performance is examined. The manner in which code placement changes in size and locality affect inlined programs' behavior is investigated. Another question considered is: What are the repercussions of expanding subprograms inline? The interactions between inlining and register allocation (widely recognized as an important code improvement technique, particularly since the advent of Reduced Instruction Set Computers [Pat85]) are discussed. Observations about inlined code that are of value to machine and compiler designers are offered. A third question probed is: How should inlining be done? A bottom-up algorithm for performing inline substitution efficiently is developed. Also, a new code improvement technique supplementing inlining's effects is presented. Furthermore, a novel approach to inlining recursive subprograms is invented.

1.2. Related Work

Subprogram inlining is not an idea that was recently conceived. In 1969, Lowry and Medlock reported that the OS/360 Fortran H compiler was using inline coding for many mathematical function subprograms [LoM69]. Source code directives for designating subprograms to be inlined have appeared as macro facilities [KeR78] and as subprogram attributes [Mac84, MMS79, Uni83].

The use of macro facilities or subprogram attributes to specify inline expansion can present difficulties. Both mechanisms burden the programmer. Specifying an inline attribute on subprogram declarations is tedious. Coding the definitions of subprograms to be substituted inline as macro definitions requires extra work, because macro definitions employ a distinct syntax from that of subprogram definitions. More importantly, macros differ semantically from subprograms in significant ways. Macro parameters are typically call by name (rather than call by value or call by reference), meaning that expression arguments are recomputed at each parameter reference in the macro body. Local scoping is not customarily supported with respect to macro bodies. Macro invocations are not interchangeable with subprogram calls; a macro may only be used at points in a program where textual substitution of the macro body would result in legal code. For these reasons, a C preprocessor macro could not be constructed that would replace any call to the subprogram *power* shown in Figure 1.1.

Compilers which automatically decide which call sites to inline have been developed by several researchers. For simplicity and to limit the impact on code size, Hecht's SIMPL-T quad improver only inlined calls to non-recursive static subprograms that contained a single return point at the end of their bodies and that were invoked from a single site [Hec77]. Scheifler formulated an algorithm for selecting call sites to inline, which was intended to minimize program execution time subject to a code size constraint [Sch77]. Ball suggested that Scheifler's method of estimating inlined code's size and execution time be modified to account for any constant value copy propagation and test code elision enabled by inlining [Bal79]. In all of this work, the code size increase associated with inlining was treated a priori as detrimental to execution time performance[4] and inlining was assumed to be advantageous in all other respects. For the research described in this dissertation, the influence of various factors on inlined code behavior was determined via experimental investigation. Inlined code size was not found to affect execution time on the computer systems studied. However, it was discovered that certain commonly-used register allocation schemes interact with inlining, in some cases yielding programs that run slower than their original, noninlined counterparts.

[4]The increased size of inlined code is clearly a problem if programs become too large to fit within the addressable memory on the target machine.

The implementation of a subprogram inliner has been outlined in previous work. MacLaren discusses an inliner that forms part of the VAXELN Pascal compiler [Mac84]. The inliner operates on an intermediate representation of Pascal code. To support the work presented in this dissertation, a source-to-source inliner for C code was designed [DaH88]. In conjunction with the development of this C inliner, functional aspects of source-to-source translation for inlining were explored, including the restructuring of statements, the renaming of data items, and the merging of multiple input modules into a single output module. (These aspects are handled similarly by Hall's source-to-source Fortran inliner [CHT90].) Also, particular inlining issues related to C constructs, such as local static variable definitions and call sites at which more parameters are passed to a function than it is declared to expect, were addressed. A bottom-up method for efficiently performing inlining was created, in which subprograms are processed in sorted order according to the relation "callee precedes caller".

Researchers have identified code improvement techniques that dovetail with inlining. As noted in Section 1.1, inlining can introduce opportunities for applying code improvements. The two improvements applied to the inlined code in Figures 1.1 through 1.4, compile-time evaluation of constant expressions and elimination of dead code, were suggested by Harrison [Har77]. Chow finds a number of global optimizations that capitalize on inlined code, including common subexpression elimination, strength reduction, loop-invariant code motion, and general copy propagation [Cho83]. Loop-invariant code motion, general copy propagation, and compile-time expression evaluation could be used to map the inlined code in Figure 1.1 into the code shown in Figure 1.5; the *for* loop is removed after the invariant

```
main()
{
    {
      {
        (void)printf("%d squared is %d\n",7,49);
      }
    }
    return(0);
}
```

Figure 1.5: An Improved Version of the Inlined Code in Figure 1.1

statement in it is moved out, the constant 7 is copy propagated, and the resulting multiplication of 7 by itself is evaluated and replaced by 49. Huson employs inlining to extend the Parafrase system's ability to vectorize Fortran DO loops [Hus82]. Parafrase's data dependence analysis [Kuc78] is hampered by the presence of CALL statements, at which the system makes worst case assumptions concerning the callee's accesses to parameters and COMMON variables. Interprocedural data flow analysis [ASU86, CKT86] could be used to track subprogram data references, but Huson chooses to utilize subprogram inlining instead, because inlining allows the exploitation of parallelism to be tailored to the calling environment. For example, interprocedural data flow analysis would support the vectorization of the single DO loop in the subprogram *sqvect* in Figure 1.6, whereas inlining *sqvect* into *main* permits the vectorization of the two nested DO loops together. In the same vein, Allen and Johnson profitably combine subprogram inlining with vectorization and parallelization in a C compiler for the Titan processor [AlJ88]. They find that, in addition to precluding conservative assumptions about the effects of subprogram calls on parameters and global variables, inlining can expose argument aliasing and can allow induction variable substitution to produce code amenable to vectorization. For the original code in Figure 1.7, the loop in the subprogram *daxpy* cannot be vectorized safely because x, y, and z could be pointers into overlapping parts of

Original Version	Inlined Version

`subroutine sqvect(vect,len)`
`  integer vect(len)`
`  integer j`
`  do 1 j = 1,len`
`  vect(j) = vect(j) * vect(j)`
`1  continue`
`  return`
`end`
`program mainprog`
`  integer square(4,4),i,j`
`  ...`
`  do 2 i = 1,4`
`  call sqvect(square(1,i),4)`
`2  continue`
`  ...`
`  stop`
`end`
	`program mainprog`
	`  integer square(4,4),i,j`
	`  ...`
	`  do 2 i = 1,4`
	`  do 1 j = 1,4`
	`  square(j,i) = square(j,i) * square(j,i)`
	`1  continue`
	`2  continue`
	`  ...`
	`  stop`
	`end`

Figure 1.6: An Inline Transformation that allows Additional Vectorization

Original Version	Inlined Version

`void daxpy(x,y,z,alpha,n)`
`  float *x, *y, *z, alpha;`
`  int n;`
`{`
`  if (n <= 0)`
`  return;`
`  for (;n;n--)`
`  *x++ = *y++ + (alpha * *z++);`
`}`
`main()`
`{`
`  float a[100], b[100], c[100];`
`  ...`
`  daxpy(a,b,c,1.0,100);`
`  ...`
`}`
	`main()`
	`{`
	`  float a[100], b[100], c[100];`
	`  ...`
	`  for (p=0,m=31; p<=99; p+=32,m=min(99,p+31)) /*parallelized*/`
	`  for (vi=p; vi<=m; vi++) /*vectorized*/`
	`  a[vi] = b[vi] + c[vi];`
	`  ...`
	`}`

Figure 1.7: An Inline Transformation that allows Vectorization & Parallelization

the same array. However, after *daxpy* is inlined into *main*, the compiler recognizes that the loop is operating on three distinct arrays and it produces code to implement the loop as a set of vector operations done in parallel.

The salutary effects of inlining are boosted when it is performed in conjunction with other code improvement techniques. Harrison notes that inlining benefits from being combined with copy propagation of actual parameter values [Har77]. Parameter copy propagation was included in the inline transformation examples given in Figures 1.1 through 1.4. Without parameter copy propagation, the quality of the inlined versions would have suffered; Figure 1.8 shows how the inlined version of the program in Figure 1.1 would have appeared without it. In the course of the research described in this dissertation, additional code improvements that supplement inlining's effects, pertaining to the handling of an inlined function's return value and to the allocation of registers when inlining, were invented.

The efficacy of subprogram inlining relative to that of other subprogram code improvement techniques is examined in ongoing work. Richardson and Ganapathi have found that interprocedural data flow analysis yields much less savings in program execution time than inlining [RiG89b]. Hall is investigating how inlining and interprocedural data flow analysis compare with cloning, in which variants of a subprogram are fashioned to suit different classes of calling environments [Hal89].

```
main()
{
  int x;
    x = 7;
    {
      int num;
      int exponent;
        num = x;
        exponent = 2;
        {
          register int result;
          int cnt;
            if (exponent < 0)
              result = 0;
            else if (exponent == 0)
              result = 1;
            else {
              result = num;
              for (cnt=2; cnt<=exponent; cnt++)
                result = result * num;
            }
            {
              (void)printf("%d squared is %d\n",x,result);
            }
        }
    }
    return (0);
}
```

Figure 1.8: Inlined Version of Code in Figure 1.1 without Parameter Copy Propagation

1.3. Organization of this Dissertation

On the whole, scant formal study of the effects of subprogram inlining on program execution time has been included in previous work. In this dissertation, equations representing the execution time performance of noninlined and inlined versions of a program are formulated, and their validity is verified. Through the use of these equations, an understanding of how their terms contribute to the execution time of inlined code is acquired. Two commonly-held notions about inlining are dispelled: first, the notion that the increased size of inlined code negatively affects its execution performance on demand-paged virtual memory machines [Sch77] and second, the notion that, aside from the detrimental effect ascribed to the increased size of inlined code, inlining is always beneficial [Bal79]. In addition, trace-driven simulation is employed to monitor the caching and paging behavior of noninlined and inlined code. Inlined code is shown to have slightly improved caching and paging behavior, using new measures designed for

comparing this behavior across versions of a program.

The dissertation is organized as follows. In Chapter 2, the tool created to perform inlining is described, and discoveries made during its development are noted. Program size and execution time measurements collected for noninlined and inlined code are given. In Chapter 3, equations that estimate the execution time behavior of noninlined and inlined versions of programs are presented, followed by a discussion of the experimental validation of the equations' accuracy on four computer systems. The influence of various factors on the speed of inlined versions of programs is examined. In Chapter 4, the negative impact on inlined program execution time of register save and restore overhead and of certain register allocation approaches is addressed. In Chapter 5, the effects of the code placement changes that can cause size increases and locality modifications in inlined programs are considered. Chapter 6 concludes the dissertation with a summary of the results reported and with areas for future work.

CHAPTER 2

Description of INLINER

To support the investigation of the effects of subprogram inlining, an inlining tool for C, called INLINER, was developed. Results found using INLINER are expected to apply to other procedural high-level languages that share with C the four basic components of subprogram call overhead.[1] In this chapter, a functional specification of INLINER is given, the implementation of INLINER is outlined, and measurements of code transformed by INLINER are presented.

2.1. Functional Specification of INLINER

Although the substitution of subprograms inline at call sites is a well-known method for improving a program's execution time, few available compilers perform the operation, and fewer still perform it without requiring explicit direction from the programmer. INLINER is a source-to-source filter that, for one or more input modules, automatically places C functions inline in the output module [DaH88]. INLINER can also be directed to search specific modules solely for needed function definitions in a manner similar to that used by linkers on archives of object files; these modules are called *source libraries*. Some calls are not expanded by INLINER; these include calls through function pointers, directly recursive calls, selected indirectly recursive calls, and calls to subprograms for which the source code is not input to INLINER, as may be the case with runtime library routines. INLINER provides options for controlling which functions are inlined.

[1]These components, listed in Section 1.1, are: transferring control to the subprogram and returning from it, along with the requisite storing and retrieving of the return address; passing any parameters specified and deleting them at the appropriate point; allocating and deallocating space for the callee's local variables; and saving and restoring certain machine registers used, according to the target computer system's conventions.

Input to INLINER is expected to be syntactically correct, and ought to be semantically correct as well, because of the difficulty of debugging code produced by INLINER. INLINER's output C code is ugly, with spacing compressed and comments removed. INLINER's inlining transformations interfere with the use of symbolic debuggers: function breakpoints are likely to be ineffective, variable names may be altered, and statements can be restructured, with new variables introduced.

In the following sections, INLINER's source code mappings for inlining are considered. First, its changes to C statements are described, and then its effects on C data declarations are discussed. Finally, certain code improvements made by INLINER in association with its inlining transformations are presented.

2.1.1. INLINER's C Statement Transformations

C has eleven statement types [KeR78]: expression, conditional, switch, return, iterative (*while*, *do*, and *for*), break, continue, null, compound, labeled, and goto. In this section, INLINER's source code transformations for each statement type are examined. Note that the C language distinction between statements and expressions complicates the transformations; in general, the statements comprising a C function body cannot directly replace a subprogram call in an expression. For an expression language like BLISS [WRH71], in which such substitutions are supported, simpler statement transformations could be employed.

INLINER's transformations for expression statements containing calls to be inlined are considered first. Inlined versions of the functions in the expression are placed immediately before the statement, and the statement itself is modified so that local variables, holding what would be the values returned by the inlined functions' callable versions, supplant the calls. An example is given in Figure 2.1. The order in which inlined function bodies precede an expression statement depends on the composition of the expression. When the expression contains function calls to be inlined that are nested, i.e., the return value of one call forms part of an argument expression for another call, the inlined version of the function in the

Original Version	Inlined Version

`int strlen(s)`
`  register char *s;`
`{`
`  register int n;`
`  n = 0;`
`  while (*s++) n++;`
`  return(n);`
`}`
	`/* function strlen deleted */`

`void func()`
`{`
`  ...`
	`void func()`
	`{`
	`  ...`
	`  {`
	`  register int AA002; /* temp for strlen's return value */`
	`  { /* inlined version of strlen */`
	`  register char *s; /* formal param mapped to local var */`
	`  s = string; /* actual param assigned to local var */`
	`  {`
	`  register int n;`
	`  n = 0;`
	`  while (*s++) n++;`
	`  { AA002 = (n);; }`
	`  }`
	`  }`

`  datarec.len =` `  strlen(string)+1;`	`  datarec.len =` `  AA002+1;`

`  ...`
`}`
	`  }`
	`  ...`
	`}`

Figure 2.1: Inline Transformation of Expression Statement

argument expression is output before the inlined version of the other function, as shown in Figure 2.2. Aside from expressions including nested calls to be inlined, the evaluation order of operands in C expressions is undefined (allowing the inlined versions of functions called in an expression to be produced in any order), except with respect to expressions containing the comma operator or the short-circuit operators.

For the comma operator, the left and then the right operand are evaluated, and the result is the value of the right operand. When calls to be inlined occur in an expression statement containing a comma operator, the statement is divided into two expression statements, the first consisting of the comma operator's left operand and the second consisting of the original expression statement with the comma operator and its left operand deleted. C's short-circuit operators are `&&` (logical and), `||` (logi-

Original Version	Inlined Version

`int square(a)`
`  int a;`
`{`
`  return(a*a);`
`}`
	`/* function square deleted */`

`int cube(a)`
`  int a;`
`{`
`  return(a*a*a);`
`}`
	`/* function cube deleted */`

`void func()`
`{`
`  ...`
 	
`  y = cube(square(x));`
 	
`  ...`
`}`
	`void func()`
	`{`
	`  ...`
	`  {`
	`  register int AA006; /* temp for square's return value */`
	`  {{ /* inlined version of square */`
	`  { AA006 = (x*x);; }`
	`  }}`
	`  {{ /* inlined version of cube */`
	`  { y = (AA006*AA006*AA006);; }`
	`  }}`
	`  /* stmt deleted; y set in inlined version of cube */;`
	`  }`
	`  ...`
	`}`

Figure 2.2: Inline Transformation of Expression Statement with Nested Function Calls

cal or), and ? (conditional selection). Operands for the && and || operators are evaluated from left to right, and evaluation stops as soon as the value of the expression can be determined. When transforming expressions in which these operators are combined with calls to be inlined, INLINER introduces integer variables, assignment statements, and control structures as necessary to ensure that inlined function bodies are not executed unless the proper conditions are met. The ? operator uses the value of its first operand to select which of its second and third operands to evaluate as its result, with the type of the result determined by analysis of the second and third operands' types. When calls to be inlined occur in either of the latter two operands, INLINER duplicates the semantics of the operator by adding a control structure (in which the value of the first operand is tested and into which functions from the second and third operands are expanded inline) and a local variable of the appropriate type (to which the resulting value is assigned). Figure 2.3 presents the inline transformation of an expression statement including two short-circuit operators.

Original Version

```
int iscntl(cin) char cin;
{ /* ASCII chars */
  return(cin<SPACE);
}

char uppercase(cin) char cin;
{
  return(((cin>='a')&&(cin<='z'))?
    ((cin-'a')+'A') : cin);
}

char func(s) char *s;
{
  ...

  achar = ((s==NULL)||(iscntl(*s))?
    SPACE : uppercase(*s);

  ...
}
```

Inlined Version

```
char func(s) char *s;
{
  ...
  { register int AA002; /* ? operator result, type converted */
      { register int AA003; /* || operator result */
          AA003 = 1;
          if (!(s == NULL))
          { register int AA009; /* iscntl's return val temp */
              { /* inlined version of iscntl */
                char AA007;
                  AA007 = *s;
                  {
                    { AA009 = (AA007<SPACE);; }
                  }
              }
              if (!(AA009))
                AA003 = 0;
          }
          if (AA003)
            AA002 = SPACE;
          else
          { { /* inlined version of uppercase */
              char AA005;
                AA005 = *s;
                {
                  { AA002 = (((AA005>='a')&&(AA005<='z'))?
                             ((AA005-'a')+'A') : AA005);; }
                }
          } }
      }
      achar = AA002;

  }
  ...
}
```

Figure 2.3: Inline Transformation of Expression Statement with Short-Circuit Operators

The transformations for conditional and switch statements whose test expressions contain calls to be inlined are similar to those for expression statements with calls to be inlined. Inlined bodies of functions called in a conditional or switch statement's test expression are placed in front of the statement, with variables that hold the inlined functions' return values substituted for the function calls in the statement. The entire statement is positioned within the scope of these variables' definitions. Figure 2.4 illustrates the inline transformation of a conditional statement.

Original Version	Inlined Version

`int strcmp(s1, s2)`
`  register char s1[], s2[];`
`{`
`  while (*s1 == *s2++)`
`  if (*s1++=='\0')`
`  return(0);`
`  return(*s1 - *--s2);`
`}`
	`/* function strcmp deleted */`

`void func()`
`{`
`  ...`
	`void func()`
	`{`
	`  ...`
	`  { register int AA002; /* temp for strcmp's return value */`
	`  { /* inlined version of strcmp */`
	`  register char *s1, *s2; /* formal type coercion */`
	`  s1 = str1; /* actual parm assigned to local var */`
	`  s2 = str2; /* actual parm assigned to local var */`
	`  {`
	`  while (*s1==*s2++)`
	`  if (*s1++=='\0')`
	`  { AA002 = (0); goto AA003; }`
	`  { AA002 = (*s1 - *--s2);; }`
	`  }`
	`  }`
	`AA003:`

`  if (!strcmp(str1,str2))`
`  {`
`  ...`
`  }`
 	
`}`
	`  if (!AA002)`
	`  {`
	`  ...`
	`  }`
	`  }`
	`}`

Figure 2.4: Inline Transformation of Conditional Statement

A return statement which includes calls to be inlined is also handled in the same way that an expression statement which includes such calls is handled. Inlined function bodies are put immediately preceding the statement, and variables into which the inlined functions' return values are deposited simply subrogate the corresponding calls in the return statement. On the other hand, return statements in inlined function bodies are modified significantly. A return statement not containing an expression is replaced by a goto statement to a label introduced by INLINER to mark the end of the inlined function body. A return statement containing an expression is mapped into a statement assigning the expression to the appropriate return value variable, followed by a goto statement to the end of the inlined function body. Any goto statement to its own execution flow successor statement is eliminated. Figures 2.1 through 2.4 show various transformations of return statements in inlined functions.

The inline transformations for iterative statements are complicated. As is the case with inline transformations for most other kinds of statements, function bodies for inlined calls in an iterative statement's initialization expression are placed directly before the iterative statement. However, function bodies for inlined calls in an iterative statement's test or update expressions must be positioned within the statement's loop body, and the statement itself must be restructured to force proper control flow to occur on each repetition and on termination of the loop. An iterative statement's restructuring necessitates the mapping of any break or continue statements in its loop body into goto statements to labels inserted by INLINER at the appropriate locations. Figure 2.5 presents a *for* loop's inline transformation.

Null statements are not modified by inlining. Inlining affects compound statements to the extent that the statements comprising them are affected. For inlined code, the label identifier in a labeled statement may need to be changed to avoid name conflicts; goto statements must be altered correspondingly to refer to the correct label identifier.

2.1.2. INLINER's Effects on C Data Declarations

INLINER changes an inlined function's formal parameters into local variables to which the inlined function's actual parameters are assigned,[2] as illustrated in Figure 2.4. The source-to-source mapping of a formal parameter declaration to a local variable declaration consists solely of placing a copy of the formal parameter declaration in the block into which the function is expanded inline, with two exceptions. One exception occurs for formal parameters that are not declared. In C, a formal parameter need not have an explicit declaration, and if it is undeclared, it is assumed to be of integer type. However, local variables must be declared, so when mapping formal parameters to local variables, INLINER inserts declarations for any undeclared formal parameters. The other exception arises when a formal parameter is declared to be of an array type. For formal parameters, C automatically treats array type declarations as pointer type declarations. INLINER mimics this formal parameter type coercion in inlined code by

[2]C allows a function's actual parameters to be evaluated in any order; for an inlined call, INLINER simply outputs the assignments of the actuals to the locals replacing the formals in the order in which the actuals appeared in the original function call.

Original Version	Inlined Version

`int getchbuff()`
`{register int ch;`
`  if ((ch=getchar()) != EOF)`
`    chbuff[char_index++]=ch;`
`  chbuff[char_index]='\0';`
`  return(ch);`
`}`
	`/* function getchbuff deleted */`

`int isdigit(x) int x;`
`{`
`  return((x>='0') && (x<='9'));`
`}`
	`/* function isdigit deleted */`

`int strtoi()`
`{`
`  register int val, c;`
`  val = 0;`
 	
`  for( c=getchbuff();`
`       isdigit(c);`
`       c=getchbuff() )`
 	
`    val = val*10 + c-'0';`
 	
`  return(val);`
`}`
	`int strtoi()`
	`{`
	`  register int val, c;`
	`  val = 0;`
	`  {`
	`    { { /* inlined version of getchbuff */`
	`      {register int ch;`
	`        if ((ch = getchar()) != EOF)`
	`          chbuff[char_index++] = ch;`
	`        chbuff[char_index] = '\0';`
	`        { c = (ch);; }`
	`      }`
	`    } }`
	`    { /* body of restructured loop */`
	`     register int AA007; /* return value for isdigit */`
	`  AA008:`
	`      { /* inlined version of isdigit */`
	`        { { AA007 = ((c >= '0') && (c <= '9'));; } }`
	`      }`
	`      if (AA007)`
	`      {`
	`        {`
	`          val = val*10 + c-'0';`
	`          { { /* inlined version of getchbuff */`
	`              {register int ch;`
	`                if ((ch = getchar()) != EOF)`
	`                  chbuff[char_index++] = ch;`
	`                chbuff[char_index] = '\0';`
	`                { c = (ch);; }`
	`              }`
	`          } }`
	`        }`
	`        goto AA008;`
	`      }`
	`    }`
	`  }`
	`  return(val);`
	`}`

Figure 2.5: Inline Transformation of Iterative Statement

appropriate transformations of the types of formal parameters when they are mapped into local variables,[3] as shown in Figure 2.4.

The C language does not require a function's actual parameters to match its formal parameters in number or type, and this feature can preclude inlining. A caller supplying fewer parameters than a callee is declared to accept does not present an obstacle to inlining. INLINER produces assignments of the actual parameters to the local variables representing the formal parameters. Any local variables representing formal parameters for which there are no corresponding actual parameters are uninitialized, just as such formal parameters are in the noninlined code. However, when a caller provides more parameters than a callee declares, inlining can result in code that does not operate as desired. Problems occur if the callee accesses arguments in addition to those declared by using positions on the call stack relative to the declared arguments. This method of coding is often used in the implementation of *printf* in the C standard I/O library. The routine *printf* is described as taking an argument containing a format string, followed by a variable number of arguments of any types. Written as shown in Figure 2.6, *printf* calls *_doprnt*, which examines the format string argument and fetches the actual arguments from the call stack using the address of the single integer argument specified as following the format string in the definition of *printf*. This technique does not work if *printf* is inlined; actual parameters other than those corresponding to formals are not available (since they have no locals to which to be assigned), and the assignment of the actual parameter following the format string to a local variable of type integer may engender an

```
printf(fmt, args)
  char *fmt;
{
    _doprnt(fmt, &args, stdout);
    return(ferror(stdout)? EOF: 0);
}
```

Figure 2.6: Definition of printf Function

[3]The type of a formal parameter may be expressed as a *typedef* name. If that name refers to a *typedef* of array type, INLINER replaces the *typedef* name in the formal parameter declaration with its type definition and then performs the coercion to pointer type.

unwanted type conversion. INLINER does not expand any call sites at which the number of actual parameters exceeds the number of formal parameters declared for the function.

In some cases, INLINER alters the names of defined entities. A global name accessible to a function is hidden from an inlined instance of that function, if the function into which the instance is substituted defines the name locally. To maintain the visibility of global names in inlined code, names defined in calling functions are made distinct from names defined globally. Similarly, inlining can make global or local names used in a function's actual parameter expressions inaccessible in the statements that assign those expressions to the local variables derived from the function's formals, if the global or local names match any of the formal names. When inlining a function, to prevent the local variables replacing the function's formal parameters from concealing the names used in its actual parameters, formal parameter names that are employed for other items within the input code are mapped to unique identifiers at the function's inlined sites. Also, since label names have function-wide visibility, inlining can result in labels being declared multiple times within a function. An inlined function's label names, if any, are mapped to avoid conflicts. Finally, a static function or global variable in an input module is renamed if its name is the same as that of a static or external function or global variable from another input module.

In C, automatic and register variables may be initialized using expressions containing function calls. Source-to-source inlining of such calls presents problems. Initialization expressions for a block's automatic and register variables must be evaluated in the order in which the variables are declared, although substituting a function inline within a block's declaration list violates the syntax of C. A general method for supporting the inlining of these calls is to remove the initialization expressions associated with a block's automatic and register variable declarations, and then to add statements at the beginning of the block's statement list to assign the initialization expressions to the appropriate variables in the proper order, handling the function calls to be inlined in the assignment statements in the usual manner. Figure 2.7 gives an example of this mapping. From a practical standpoint, however, function calls seldom appear in initialization clauses, and INLINER currently does not inline these calls.

Variables declared static within the scope of an inlineable function are promoted to static global variables with names unique in the output module, so that the variables can be referenced by all of the

Original Code	Modified Code
{	{
int x = f(a);	int x;
int y = x;	int y;
	x = f(a);
	y = x;
...	...
}	}

Figure 2.7: Example of Code Modification to Support Inlining Calls in Initializations

function's inlined versions as well as its callable version (if any). This transformation ensures that after inlining, these variables behave like they did as local statics in the noninlined program, i.e., they retain their values across executions of the function's code. If a static local variable is declared using a local *typedef*, then prior to making the variable global, the *typedef* in the variable's declaration is replaced by the definition of the *typedef* in terms of C's built-in types. Correct type compatibility is maintained by this mapping, because *typedef*s are considered to be synonyms for their definitions, not new types. If a static local variable is declared using a local *struct*, *union*, or *enum* type, then the situation is complicated. Since each definition of these types is treated as a new type, preserving the type compatibility qualities of the original code when promoting a local static defined using one of these types to a global static requires that the corresponding type name definition also become global, with its name modified if a conflict with the name of some other program entity occurs. The constant names in an *enum* definition may have to be altered as well. Moreover, field declarations for a local *struct* or *union* type might in turn depend on local *typedef*s, which would need to be replaced with their definitions, or on local *struct*, *union*, or *enum* declarations, which would again need to be promoted to global level and possibly renamed. INLINER does not change static local variables declared using local *struct*, *union*, or *enum* types to static global variables and does not inline functions containing such static locals.

When inlining is performed across more than one input module, several issues related to drawing global declarations from different modules affect INLINER's operation. First, the C language dictates that no two distinct external global variables may have the same name, and this rule also applies to global *typedef*s and type tags for *struct*s, *union*s, and *enum*s. However, C compilers typically process each of a

program's modules independently, and since *typedef*s and type tags are (usually) strictly compile-time objects, redefinition of their names in different modules is generally not flagged as incorrect. Such redefinition is considered to be erroneous input to INLINER. Next, to make a particular external global variable name visible to more than one module, most dialects of C allow any number of referencing declarations[4] for the name, but only one defining declaration[5] for it [HaS84]. Nevertheless, if an external global variable is given multiple defining declarations in separate modules, these redundant definitions are not detected by the compiler, and some linkers resolve them into a single data item without reporting an error. INLINER supports this common extension to the C language by combining duplicate external global variable definitions from separate source files into one definition in the output source file. Similarly, C permits global *typedef* and *struct*, *union*, and *enum* type tag names to have only one defining declaration,[6] although replicated defining declarations across modules are ordinarily not caught at compile or link time. As with external global variables, INLINER combines multiple definitions of global *typedef* and type tag names into a single definition.

2.1.3. INLINER's Improvements to Inline Transformations

Certain code improvements which may enhance inlining's effectiveness can naturally be performed in conjunction with inlining. These improvements involve the transmission of parameter values to and return values from inlined function bodies, as well as the allocation of registers and string constant space. INLINER includes such optimizations, as discussed in the paragraphs below.

An inlined function's actual parameter values are sometimes copy propagated, as is recommended by Harrison [Har77]. If a formal parameter is never written, its address is never taken, and the corresponding actual parameter meets certain conditions, then the actual parameter replaces the formal

---

[4] In a referencing declaration for a variable name, the variable is marked *extern* and is not initialized.

[5] In a defining declaration for a variable name, the variable is not marked *extern* and may be initialized.

[6] Any *typedef* name declaration is a defining declaration. A defining declaration for a *struct* or *union* type tag name is one that includes the specification of component fields, and a defining declaration for an *enum* type tag name is one in which a list of the enumeration constants appears.

parameter in the inlined function body, and the local variable representing the formal parameter along with the assignment of the actual parameter to that local variable are elided. The conditions to be met by the actual parameter value are designed to ensure that its copy propagation does not change the semantics of the code or increase the cost of accessing the parameter value. The actual cannot be a general expression, because of the expense and possible side-effects associated with evaluating the expression more than once; the actual may be either a simple variable or an integer, character, or string constant. Since the values of global class variables (excluding those of array type which are actually array addresses) may be altered by direct reference in the inlined function body, such variables are not copy propagated. Also, global and static class variables are typically costlier to access than locals of automatic class. Finally, if the formal parameter is of register class, then the actual parameter must be of register class as well.

Copy propagation of constant parameters into inlined code is complemented by compile-time constant expression evaluation [Har77]. INLINER computes arithmetic and logical operations on constant operands, and substitutes the resulting values for the calculated expressions. In addition, INLINER eliminates dead code [Har77]. Conditional statements based on control expressions with constant values, often stemming from constant parameter copy propagation and compile-time constant expression evaluation, are replaced by any dependent statements that are always executed.

INLINER makes several improvements in the handling of inlined function returns. When inlining a void function, a function that contains no return statements, a function whose return value is ignored by its caller, or a function that always returns a particular constant value, the local variable defined to hold the inlined function's return value and any assignments to that local are omitted. (In the case of an ignored return value, code is produced to evaluate return expressions for side-effects, and in the case of a constant return value, the constant value replaces the call site in the caller's code.) The local for an inlined function's return value and the assignments to that local are also deleted when a single inlineable call occurs in an expression or return statement. The caller's expression or return statement is moved back into the inlined callee's code; a copy of the statement is placed at each of the inlined callee's return statement sites, with the inlined callee's return statement expression substituted for the call to the inlined

function.[7] For example, INLINER actually produces a revised version of the inlined code presented in Figure 2.1, in which the local variable *AA002* is not declared, the assignment to *datarec.len* located outside the inlined code for *strlen* does not appear, and the assignment to *AA002* in *strlen*'s inlined body is replaced by the assignment *datarec.len = (n) + 1*. Finally, using limited flow analysis, INLINER removes any redundant goto statements produced when transforming an inlined function's return statements. These gotos thwart some C compiler optimizers.

The callable functions in inlined code, since they arise from a combination of the functions in the original input, tend to contain more variables than the functions occurring in noninlined code. The increased competition in inlined code of a function's variables for the allocable registers may cause variables assigned to registers in the noninlined version of a program to be assigned to stack memory locations in an inlined version. This displacement of variables from registers can hurt the performance of inlined code relative to noninlined, as will be discussed in detail in Section 4.2. INLINER, using knowledge of the target compiler's register allocation scheme and the target computer system's register set(s), will optionally forgo inlining that results in register variables becoming stack memory variables.

Many C compilers generate a new copy of each string constant found in the source code, even when a string constant is identical to one already encountered. To save space, INLINER promotes any string constants in functions that are inlined to static global arrays, so that the strings are not replicated for every inlined instance of the functions.

2.2. Implementation of INLINER

INLINER is composed of the four processing phases shown in Figure 2.8. The C preprocessor is run first, to expand macros and to input text from include files. Then, the Function Definition phase parses the input C source code, gathering symbol table information. Next, the Function Analysis phase

[7] For both Fortran 77 [Pra84] and Pascal [JeW78], a function's return value is specified by assigning it to the function identifier and the assignment need not directly precede exiting the function, so this code improvement would not be applicable to code written in these languages in some cases.

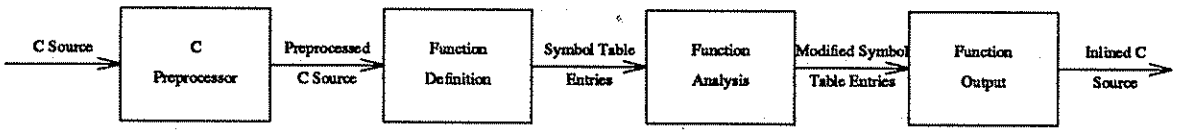


Figure 2.8: INLINER Organization

determines which call sites to inline. Finally, the Function Output phase performs the inlining transformations and produces callable versions of some functions. The last three phases are described in more detail in the following sections.

2.2.1. The Function Definition Phase of INLINER

INLINER's Function Definition phase parses the C source modules specified as input, optionally searching source libraries for any functions not defined in the input modules. Extensive information for INLINER's symbol table is collected, particularly with respect to three constructs: function definitions, function references, and global non-function declarations. For each function defined in the input, a tree is built to store the function's body, with a pointer to that tree placed in the function's symbol table entry. Call sites in the function that are prospects for inlining are indicated in the tree. Several actions related to the function's formal parameters are performed. Array type formal parameters are coerced to pointer type, and declarations are inserted for undeclared formal parameters. Any formal parameter to which a value is assigned or of which the address is taken is marked so that any corresponding actual parameter is not a candidate for the copy propagation optimization.

A number of issues are considered when a reference to a function is encountered in the input. If the reference elicits the address of the function, then the need for producing a callable version of the function is noted, since the function may be invoked through the pointer. If the reference is a call to an explicitly named function, data representing the count of actual parameters passed to the function is stored in the function's symbol table entry for later comparison with the count of formal parameters

declared for the function. For a directly recursive call, the necessity for a callable version of the function is logged. For any other call not executed through a pointer, the call site is added to the list of those that are potentially inlineable, and the site's caller and callee pair are recorded for use in a sort performed by the next phase of INLINER.

Global non-function declarations are placed in the symbol table, with duplicate variable, *typedef*, *struct*, *union*, and *enum* name declarations combined as described in Section 2.1.2. The declarations are output at the end of the Function Definition phase, after all of the input code has been read.

2.2.2. The Function Analysis Phase of INLINER

INLINER's Function Analysis phase uses the data collected in the Function Definition phase to sort the functions referenced in the input and to select which call sites to inline. The functions are sorted according to the relation "callee precedes caller". A cycle in the ordering indicates the presence in the code of indirect recursion, and is broken by arbitrarily choosing one of the call sites in the cycle and deleting it from the list of inlineable call sites. INLINER could be improved if it selected for removal from the list of inlineable call sites the least frequently executed call site in the cycle, based on typical call frequency data.[8]

After sorting the functions, INLINER calculates the information needed to decide which call sites in a function to inline, processing the input functions in sorted order so that when the inlining of a call site is considered, the callee's characteristics have already been computed. A call site at which more actual parameters are supplied than the callee is declared to accept is not inlined. The decision not to inline a call site may be based on the restriction that inlining not cause variables allocated to registers in the noninlined version of a program to be assigned to stack memory locations in an inlined version.[9] INLINER implements this option using knowledge of the target compiler's register allocation method

[8]For the set of test programs used to study inlining, indirectly recursive paths were seldom executed.

[9]Also, the decision not to inline a call site may be based on a static program block nesting level depth limit. This option is used when inlining on the MC68020, since the Sun-3 C compiler imposes such a limit.

and the target computer system's available registers. At the same time, statements containing both short-circuit operators and potentially inlineable calls are restructured, as described in Section 2.1.1, because any local variables introduced to support the restructuring may affect the allocation of registers in inlined code. In addition, the issues of the applicability of copy propagation for actual parameters at a call site and the handling of the return value (if any) from the called function are resolved, since these issues also influence the placement of variables in registers in inlined code.

2.2.3. The Function Output Phase of INLINER

INLINER's Function Output phase considers each stored function in "callee precedes caller" sorted order, mapping the function's data, producing a callable version of the function (if needed), and creating an inlineable version of the function (if needed). With respect to the function's data, if the function contains inlineable calls, the function's locals and formals are mapped to unique names, so that they will not hide globals from inlined function bodies. If the function itself will be inlined, its static locals are mapped to static globals and its string constants are mapped to static global character arrays.

If warranted, a version of the function that can be called is then output. The question of whether or not a function needs to exist in callable form is a complicated one to answer. In C, a function has external visibility by default, and the presence of an external call is typically not detected until link time. These facts imply that a function should be output in callable form, unless the function is declared static and all of its call sites are inlined. However, this approach will often produce unneeded callable versions of functions, because programmers seldom declare static functions as such, and because, when all of a program's user-defined modules are input to INLINER together, almost no programmer-defined functions are called from code external to that being processed by INLINER. INLINER uses the following algorithm to decide when to write out a callable version of a function.

(1) If the function is directly recursive, or a call to the function is designated as not inlineable in order to break an indirectly recursive call cycle, or the address of the function is taken, or the function is called with more parameters than it is defined to accept, or at least one call to the function is not

inlined because doing so would cause the displacement of register variables to stack memory,[10] or an option to INLINER specified that the function not be inlined, or a static local in the function cannot be promoted to a static global, or there is a call to the function from a variable initialization clause, then a callable version of the function is output.

(2) Otherwise, if the function is declared to be static, a callable version is *not* output.

(3) Otherwise, if the function is never called in the input code, a callable version of the function is output, on the assumption that the function is not redundant and therefore must be called from a function in a module not input to INLINER. A common example of such a function is *main*.

(4) Otherwise, if an option to INLINER directed that all functions be treated as though they were by default static, a callable version of the function is *not* output.

(5) Otherwise, a callable version of the function is output.

After the matter of providing a callable version of the function is addressed, the function may be converted to inlineable form. If the function will be inlined at one or more call sites, preparatory inlining transformations are applied to the tree denoting the function's body. These transformations include positioning a label at the end of the function's body, replacing the function's return statements with assignments and gotos as appropriate, deleting any resulting gotos that are redundant, inserting a node to indicate where the assignments of actual parameters to local variables representing formal parameters should be placed, and introducing a block enclosing the function's body in which the formal parameters are declared as local variables. Whenever an inlineable call to the function is encountered while writing out some function's callable version, call site specific inlining transformations are done as the inlined function body is being output. These transformations include changing labels in the inlined function body to unique names, handling each of the call site's actual parameters either by copy propagation or by assignment to a local variable holding the inlined function's corresponding formal parameter, performing compile-time expression evaluation and dead code elimination, and declaring a local variable, if necessary, for the inlined function's return value.

[10]Or at least one call to the function is not inlined because doing so would cause the static program block nesting level depth to exceed a specified limit.

2.3. Measurements of Code Transformed by INLINER

Timing and size measurements were gathered using the four computer systems and compilers listed in Table 2.1 for noninlined and inlined versions of the test programs given in Table 2.2. The VAX-8600 and MC68020 computer systems are considered to be complex instruction set computers, while the Clipper and Convex computer systems are classified by their manufacturers as reduced instruction set computers. When compiling the test programs, target compiler optimization was enabled, using the appropriate compiler option. The target compilers' code optimizers ranged in capability from that of the

Processor	Operating System	C Compiler
VAX-8600	UNIX 4.3BSD	Portable C Compiler
MC68020	SunOS Release 4.0.1	Sun-3 C Compiler
Clipper	CLIX System V Release 3.1	Green Hills C Compiler
Convex	Convex UNIX Release V6.2	CONVEX C Compiler

Table 2.1: Computer Systems used in Inline Experiments

Name	Description	Source	Modules
cache	Cache Usage Simulator	User Code	8
calls	Produce Static Call Graph	User Code	3
compact	Huffman Coding File Compression	UNIX System	1
diff	File Difference Utility	UNIX System	2
dry	Dhrystone Benchmark Program	Synthetic Benchmark	1
extract	Derive Design Material from Comments	User Code	1
grep	Search for Pattern	UNIX System	1
inltest	C Function Inliner (old version)	User Code	8
invert	Create a Reference Database for bib	UNIX System	3
mincost	VLSI Circuit Partitioning	User Code	1
nroff	Text Formatting Utility	UNIX System	11
sed	Stream Editor	UNIX System	3
sort	Sort or Merge Files	UNIX System	1
tsp	Traveling Salesperson	User Code	1
vpcc	Very Portable C Compiler	User Code	13
whetstone	Benchmark Program	Synthetic Benchmark	1
yacc	Create Parser Tables from Grammar	UNIX System	4

Table 2.2: Test Programs used in Inline Experiments

Portable C compiler, which performs some local optimizations, to that of the Green Hills C compiler, which performs certain local and global optimizations, including passing parameters in registers and automatic register allocation. Two kinds of inlined versions of the programs are examined: intramodule inlined, in which inlining is performed separately within each module in the program, and intermodule inlined, in which inlining is performed across all the modules in the program.

The test programs are drawn from application, system, and benchmark code. Table 2.3 presents size and call frequency data for the test programs. The size data includes lines of code, subprograms defined, and average lines per subprogram. The lines of code value given for each program is a count of the lines in the program's C source modules (not including runtime library modules), minus any empty or comment lines. Dividing that value by the number of subprograms defined in the modules yields the program's average lines of code per subprogram. The call frequency data for each program is comprised of total calls executed, total call sites exercised, average calls per call site, percentage of call sites executing 1 call, and percentage of call sites executing 1000 or more calls. These quantities were collected on the MC68020 with respect to the test data input to the programs. The average number of calls per call

Name	Lines of Code	Sub-programs Defined	Ave Lines per Sub-program	Total Calls Executed	Total Call Sites Exercised	Average Calls per Call Site	% Sites Executing 1 Call	% Sites Executing >=1000 Calls
cache	484	32	15	5,184,136	79	65,622	44	41
calls	615	17	36	465,070	68	6,839	10	15
compact	371	5	74	84,981	12	7,082	42	33
diff	821	28	29	819,914	79	10,379	62	23
dry	322	13	25	850,021	31	27,420	42	52
extract	2,396	30	80	33,543	110	305	23	9
grep	460	7	66	12,389	12	1,032	67	8
inltest	6,401	186	34	853,702	566	1,508	9	17
invert	285	12	24	505,023	74	6,825	49	32
mincost	412	16	26	972,145	53	18,342	13	62
nroff	5,267	227	23	610,135	497	1,228	18	13
sed	1,467	21	70	659,122	29	22,728	41	21
sort	850	20	43	854,794	59	14,488	53	14
tsp	352	13	27	37,265,742	121	307,981	17	77
vpcc	6,781	169	40	3,130,671	609	5,141	8	38
whetstone	217	8	27	693,042	96	7,219	20	75
yacc	1,828	48	38	98,581	176	560	41	9

Table 2.3: Size & Call Frequency Data for Test Programs

site is the quotient of the total number of calls executed and the total number of call sites exercised (both including runtime library calls). Since the number of calls per call site in a program varies widely, two call frequency percentages (the percentage of call sites at which only 1 call was executed and the percentage of call sites at which 1000 or more calls were executed) are also presented. A representation of execution call site trees for the test programs run on their data sets is given in Figures 2.9a-c. The rows in the figures correspond to dynamic call nesting depths and each character in a row signifies an exercised call site of unique parentage at that depth. (Call sites in the runtime library are included.) The particular character indicates call frequency and call graph structure information as explained in the figure's legend. Ellipsis is designated in wide or long trees using the string "...". Table 2.3 and Figures 2.9a-c show the diversity of the test programs' composition and behavior.

For constructing the test programs, several machine-independent routines from C's runtime libraries were supplied to INLINER in source library modules for inline expansion. Callable versions of the routines were placed in an archive file for access when linking the noninlined versions of the test programs as well as for access when linking the inlined versions of the test programs, since not all of the routines' call sites are necessarily inlined. The runtime library routines included are enumerated in Table 2.4.

In addition, when building the test programs, INLINER was directed not to expand any call sites which, if inlined, would cause variables placed in registers in the noninlined version of the program to be placed in stack memory locations in the inlined version. To implement this restriction, INLINER keeps a running tally of the variables in a function that the target compiler will assign to registers, and does not inline a call site in the function when that action would result in the tally exceeding the number of allocable registers on the target computer system. Counts of the allocable registers on the target systems under consideration are shown in Table 2.5. Note that the available registers on the MC68020 and on the Clipper are divided into two sets, according to the kind of information the registers are designed to contain. For each of the target compilers, INLINER knows which types of variables are allocated to registers, to what kind of register a variable of a particular type is assigned, and whether the variable must be

Legend	
*	Call Site exercised 1000 times or more to subprogram with executed children
+	Call Site exercised fewer than 1000 times to subprogram with executed children
/	Call Site exercised 1000 times or more to subprogram without executed children
-	Call Site exercised fewer than 1000 times to subprogram without executed children

cache

```
  0 +
  1 +++++-
  2 **+++++++------------------------
  3 ********////-------
  4 ***********/////--
  5 ********+++/////////////////----
  6 +++++++///////-
  7 ++-------
  8 --
```

calls

```
  0 +
  1 ++---
  2 ++++--------
  3 *+++++++++/-----------------
  4 ********///----------------
  5 *++++///-----------
  6 +----------
  7 +-------
...
 14 +------
 15 ----
```

compact

```
  0 +
  1 *++/-----
  2 //---
```

diff

```
  0 +
  1 +-----
  2 ++++++++++++++++-----------------
  3 ***************/////---------------------
  4 ********++/-----------
  5 ***////----------
  6 //------
```

dry

```
  0 +
  1 +-
  2 **++++///////------
  3 **///---------
  4 //
```

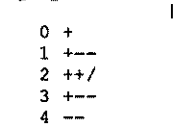
extract

```
  0 +
  1 *++++++++////-------------
  2 ++++++++/------------
  3 ++++++++++++++++++++++++++//-----------------------------------
  4 +++++++++------------------------------
  5 ++++++++++++++++++-
  6 ++++++++++++++++++++++++++++++++++-----------
  7 ++++++++++++++++++++++++++++++++----------------------------------------
...
 13 +-----------------------------------------------------------------------------------------------------------------
 14 ---
```

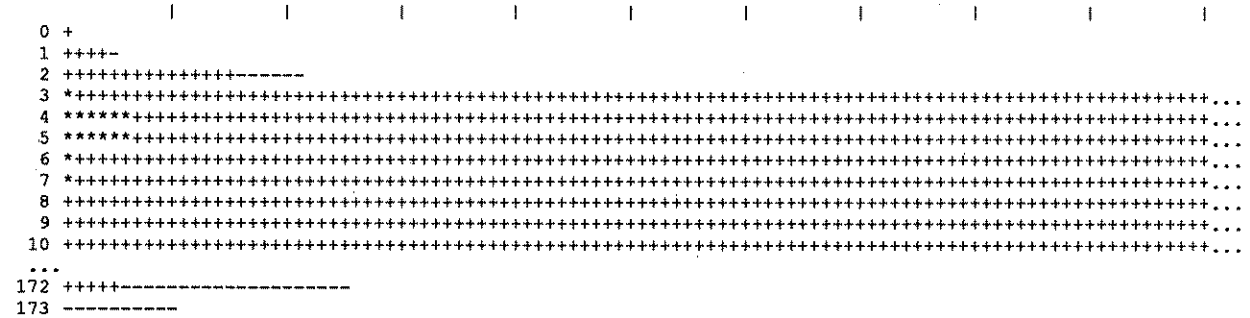
Figure 2.9a: Execution Call Site Trees for Test Programs

Legend	
*	Call Site exercised 1000 times or more to subprogram with executed children
+	Call Site exercised fewer than 1000 times to subprogram with executed children
/	Call Site exercised 1000 times or more to subprogram without executed children
-	Call Site exercised fewer than 1000 times to subprogram without executed children

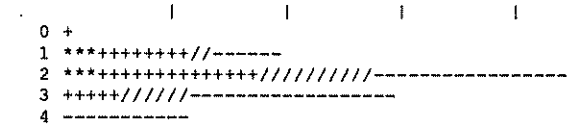
grep



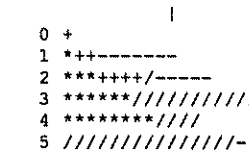
inltest



invert



mincost



nroff

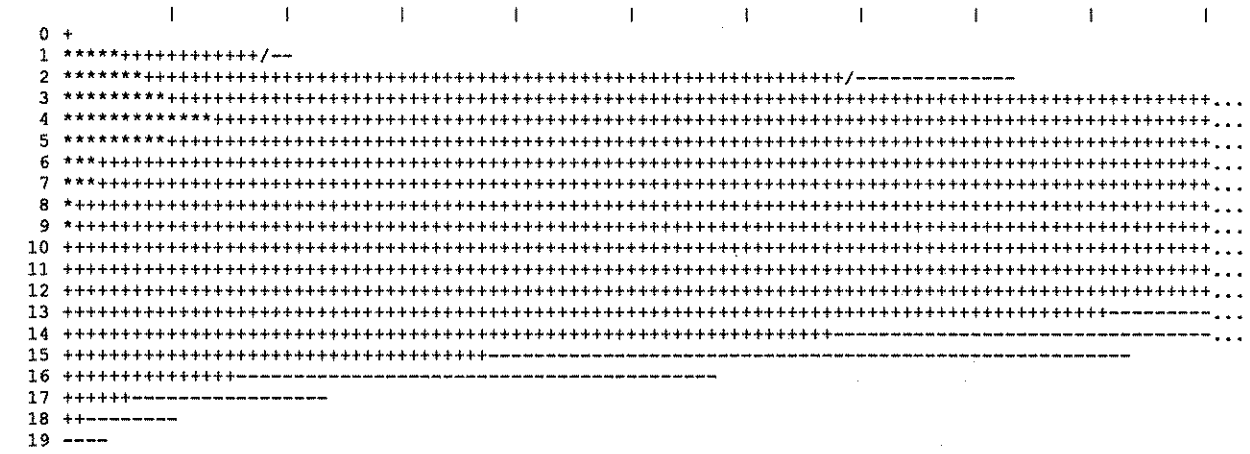
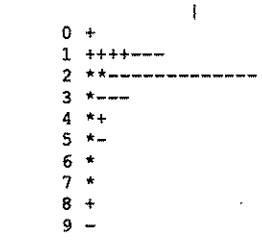


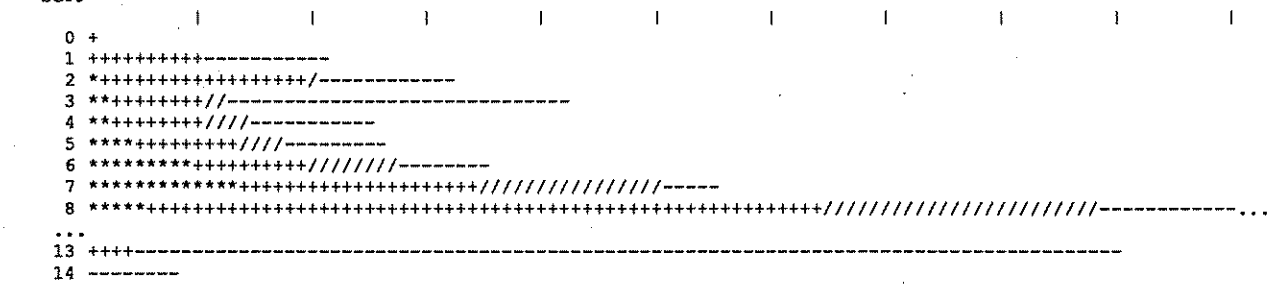
Figure 2.9b: Execution Call Site Trees for Test Programs, continued

Legend	
*	Call Site exercised 1000 times or more to subprogram with executed children
+	Call Site exercised fewer than 1000 times to subprogram with executed children
/	Call Site exercised 1000 times or more to subprogram without executed children
-	Call Site exercised fewer than 1000 times to subprogram without executed children

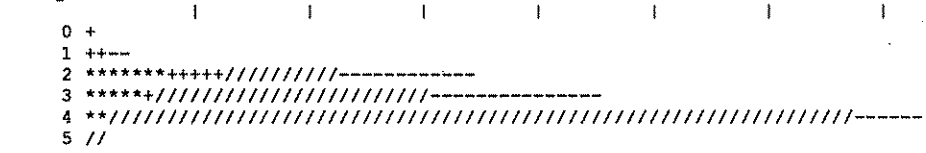
sed



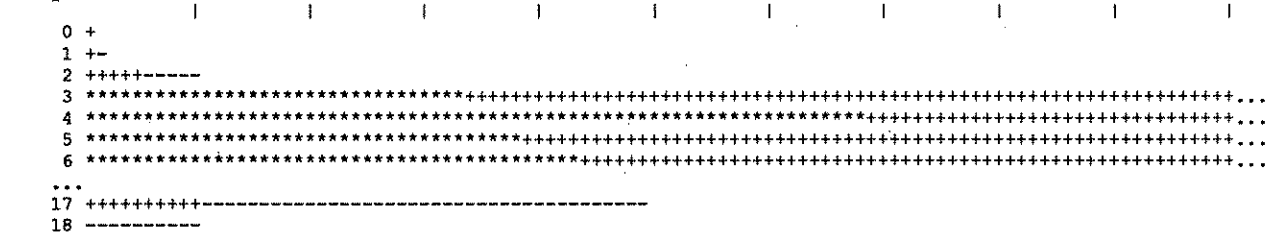
sort



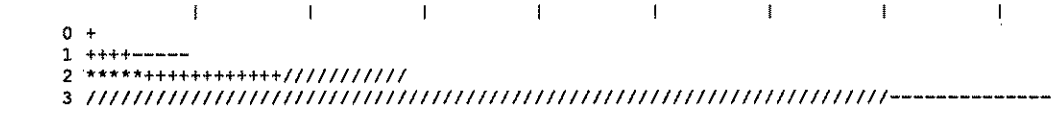
tsp



vpcc



whetstone



yacc

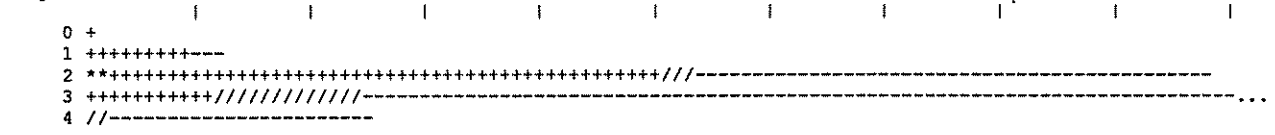


Figure 2.9c: Execution Call Site Trees for Test Programs, continued

Input/Output Routines			String Processing Routines			Other Routines		
fgetc	fputs	gets	index	strchr	strcpy	atoi	atol	mktemp
fgets	fread	puts	rindex	strrchr	strncpy			
fputc	fwrite		strcat	strcmp	strlen			
			strncat	strncmp				

Table 2.4: Runtime Library Routines Supplied for Inlining in Inline Experiments

Target Computer System		Allocable Registers		Register Allocation Factors	
Processor	Compiler	Count	Kind	Candidate Types for Registers	Class
VAX-8600	Portable C	6	General	pointer,integer,long,unsigned, unsigned long	register
MC68020	Sun-3 C	6	Data	integer,long,unsigned,unsigned long, char,short,unsigned char,unsigned short	register
		4	Address	pointer	register
Clipper	Green Hills C	9	General	pointer,integer,long, unsigned,unsigned long,enum	any
		4	Float	float,double	any
Convex	CONVEX C	4	General	pointer,integer,long,unsigned, unsigned long,char,short, unsigned char,unsigned short, float,double,enum,union,struct	register

Table 2.5: Register Allocation Data for Computer Systems used in Inline Experiments

declared register class in order to be associated with a register.[11] Table 2.5 also contains this information for the target compilers. The number of allocable registers and the variety of types of candidate variables vying for those registers influence the relative number of call sites inlined on each of the target computer systems.

Inlining was also restrained on the MC68020, because the Sun-3 C compiler imposes a limit on the static program block nesting level depth allowed in C source code. The limit is 50 units, where each { represents 1 unit, each *while* or *do* represents 2 units, and each *for* or *switch* represents 3 units. On the MC68020, INLINER avoided substituting functions inline whenever the limit would be overstepped.

[11]The Green Hills C compiler performs automatic register allocation, while the other three compilers simply attempt to place register class variables in registers.

When inlining the test programs, the decision to yield a callable version of a function was made as described in Section 2.2.3, with certain options stipulated. For intramodule inlining, INLINER was run with a switch instructing it to assume functions were static class by default. Consistent with this premise, any functions for which a needed callable version would not otherwise be output were explicitly defined to be external. For intermodule inlining, INLINER was run with a switch instructing it to treat all defined functions as static, even those specifically declared external.

A difference between noninlined and inlined programs, distinct from the differences attributable to the inlining transformations, is that the order of callable functions in noninlined and inlined executable files varies. The functions in noninlined executable files are in user-specified order, i.e., the functions occur in the order they appear in the user's source modules, with the modules arranged in the order they are listed in the user's link command. INLINER outputs functions in "callee precedes caller" sorted order, as explained in Section 2.2. Hence, in intramodule inlined executable files, although the user's modules are still arranged as given in the link command, the functions within those modules are situated in sorted order. In intermodule inlined executable files, all of the user's functions occur in sorted order. The reason that INLINER sorts functions in "callee precedes caller" order is to enhance the efficiency of certain of its operating phases. However, it is possible for the relative positions of functions in an executable file to affect program runtime [HaG71].

To see whether, on a commonly-used system, the execution speed of programs whose functions are located in sorted order diverges from the execution speed of programs whose functions are located in user-specified order, the performance of three noninlined versions of the test programs was compared on the MC68020 system: a user-ordered version, with the functions in user-specified order as described above; an intramodule-reordered version, in which the modules are positioned in the order dictated by the user's link command, but the functions within those modules are sorted in "callee precedes caller" order; and an intermodule-reordered version, in which all of the user's functions are placed in sorted order. All three versions of each program ran at essentially the same speed on the MC68020 system. To constrain the differences between noninlined and inlined versions of the test programs to be those introduced by inline substitutions, the noninlined versions of the programs are sorted in "callee precedes

caller'' order just as the inlined versions are.[12] For the timing and size measurements given shortly, the following policies are employed. For test programs with a single user module, the execution time of the intramodule inlined version is compared with the execution time of the intramodule-reordered noninlined version, and the code section size of the intramodule inlined version is weighed against the code section size of the intramodule-reordered noninlined version. For test programs with multiple user modules, the execution times of the intramodule and intermodule inlined versions are compared with the average of the execution times of the intramodule- and intermodule-reordered noninlined versions, while the code section size of the intramodule inlined version is weighed against the code section size of the intramodule-reordered noninlined version, and the code section size of the intermodule inlined version is weighed against the code section size of the intermodule-reordered noninlined version.

The volume of code produced by INLINER for certain programs taxed the target translation utilities. Several problems were encountered in building the intermodule inlined version of *vpcc*. For that version of *vpcc* on the VAX-8600, an assembly option was required to cause word-offset branches to be replaced by long jumps, and a special tool was employed to map certain word-offset jump tables to long-offset jump tables. When constructing that version of *vpcc* on the MC68020, the use of a compiler option was necessary to force long-offset jump tables to be substituted for word-offset jump tables. On the Convex, that version of *vpcc* would not compile, due to a bug in the CONVEX C compiler.

As one would expect, inlined versions of programs nearly always took longer to build than noninlined. Table 2.6 contains noninlined and inlined program build times and ratios collected on the Sun-3.[13] The build time quantities given are the average of three wall-clock elapsed-time values, measuring the time consumed by the combination of inlining, compiling, and linking, under the direction of the UNIX *make* utility. The ratios of inlined to noninlined build times found in this experiment are lower than those reported by Richardson and Ganapathi [RiG89b], whose average ratio of inlined to noninlined compile

[12]To sort the functions in the noninlined versions, INLINER is invoked on the modules in those versions with a command line parameter which instructs it to sort the functions in the input modules without inlining any of the call sites.

[13]It is interesting to observe that, for programs with multiple source files, the intermodule-reordered noninlined version built in less time than the intramodule-reordered noninlined version. This may be due to savings in system task and file overhead; for an intramodule-reordered noninlined build of a program with n source files, n compilations of a total of n separate files are executed, whereas for an intermodule-reordered noninlined build, only 1 compilation of a single source file is performed.

	Time in Seconds				Ratio of Inlined to Noninlined Time		
	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined	Intramodule Inlined 1 module	Intramodule Inlined >1 module	Intermodule Inlined >1 module
cache	57	62	33	40		1.09	1.21
calls	51	93	39	71		1.82	1.82
compact	31	36			1.16		
diff	67	164	67	179		2.45	2.67
dry	22	19			0.86		
extract	90	359			3.99		
grep	18	23			1.28		
inltest	194	325	157	388		1.68	2.47
invert	33	41	22	35		1.24	1.59
mincost	26	32			1.23		
nroff	241	348	209	1461		1.44	6.99
sed	67	93	53	109		1.39	2.06
sort	32	67			2.09		
tsp	28	33			1.18		
vpcc	289	523	247	2854		1.81	11.55
whetstone	35	41			1.17		
yacc	109	280	85	312		2.57	3.67
AVERAGE					1.62	1.72	3.78

Table 2.6: Program Build Times on the Sun-3 for Test Programs

times was 4.8. One reason their ratios were worse may be that, unlike the Sun-3 C compiler, their compiler performs subprogram-level global optimization, and they indicated that their global optimization algorithms were strained by the increased size of subprograms after inlining. Another reason for the difference may be that INLINER's method of processing subprograms in "callee precedes caller" sorted order is more efficient than the method used by Richardson and Ganapathi's inliner.

Percentages of the test programs' executed calls that were eliminated by inlining are shown in Table 2.7.[14] These percentages were computed from counts (given in Tables A.2, A.9, A.14, and A.23) of calls executed by the noninlined and inlined versions of the test programs. The amount of inlining that occurred varied across computer systems, due mainly to differences in the systems' register sets and register allocation policies. On a particular system, the level of inlining varied across test programs, depending largely on each program's utilization of variables allocable to registers, calls to runtime library routines unavailable for inlining, module structure (when intramodule inlining was performed on multiple module programs), and use of recursion, calls through pointers, and calls with extra actual parameters.

[14]The percentages for the intramodule inlined versions are subdivided into two columns: one for programs comprised of a single module and the other for programs comprised of multiple modules.

	VAX-8600			MC68020		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		53	77		58	84
calls		10	10		69	69
compact	100			100		
diff		91	93		93	96
dry	100			94		
extract	82			80		
grep	0			0		
inltest		51	63		51	63
invert		82	88		82	88
mincost	96			96		
nroff		52	44		41	50
sed		19	72		7	68
sort	64			66		
tsp	53			17		
vpcc		43	56		46	59
whetstone	66			7		
yacc		46	51		53	53
AVERAGE	70	50	62	57	56	70

	Clipper			Convex		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		63	90		54	83
calls		97	97		4	4
compact	56			56		
diff		99	6		2	4
dry	94			38		
extract	42			67		
grep	0			0		
inltest		67	68		48	59
invert		29	37		29	52
mincost	77			94		
nroff		54	58		26	18
sed		11	4		7	7
sort	0			64		
tsp	43			53		
vpcc		49	62		35	*
whetstone	100			56		
yacc		54	52		45	45
AVERAGE	51	58	53	53	28	34

Table 2.7: Percentage of Executed Calls in Test Programs Eliminated by Inlining

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

	VAX-8600				MC68020			
	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined
cache	10,768	11,108	10,760	12,952	35,584	35,896	35,496	37,576
calls	13,644	19,240	13,652	19,240	24,720	31,424	24,704	31,416
compact	11,656	13,740			22,576	24,504		
diff	26,540	38,500	26,536	39,260	51,024	75,240	50,960	76,144
dry	5,720	5,624			18,664	18,472		
extract	24,880	81,012			40,464	100,936		
grep	9,852	10,676			20,872	21,512		
inltest	34,552	52,604	34,548	108,324	65,152	89,696	64,232	142,128
invert	11,600	12,232	11,596	12,428	36,608	37,176	36,584	37,224
mincost	10,460	11,260			35,632	36,320		
nroff	46,144	81,372	46,132	205,040	72,800	104,584	72,008	419,624
sed	16,536	31,028	16,532	33,892	30,680	42,704	30,664	49,856
sort	12,900	21,464			23,864	33,304		
tsp	11,856	12,128			61,792	61,504		
vpcc	51,268	84,624	51,256	331,532	81,744	148,368	80,880	684,072
whetstone	10,960	11,268			60,336	60,784		
yacc	26,004	58,760	26,000	78,176	41,232	79,800	41,080	98,560

	Clipper				Convex			
	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined
cache	21,560	21,728	21,472	23,272	28,672	32,768	28,672	32,768
calls	19,768	29,344	19,744	28,680	36,864	40,960	36,864	40,960
compact	16,848	18,400			28,672	32,768		
diff	38,680	50,920	38,616	51,984	69,632	94,208	69,632	98,304
dry	12,176	12,176			20,480	20,480		
extract	34,752	56,664			49,152	110,592		
grep	15,296	16,288			28,672	32,768		
inltest	53,480	74,824	52,824	158,224	86,016	110,592	86,016	196,608
invert	23,128	23,352	23,112	23,568	32,768	32,768	32,768	32,768
mincost	21,912	21,960			28,672	28,672		
nroff	59,440	127,744	58,592	466,864	94,208	135,168	94,208	221,184
sed	24,112	29,872	25,096	31,600	40,960	61,440	40,960	61,440
sort	18,632	23,472			32,768	40,960		
tsp	24,952	24,992			49,152	49,152		
vpcc	70,720	100,432	70,016	287,616	122,880	151,552	*	*
whetstone	19,160	19,128			32,768	32,768		
yacc	36,952	42,424	36,792	44,632	57,344	69,632	57,344	94,208

Table 2.8: Program Code Section Sizes (in Bytes) for Test Programs

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

	VAX-8600			MC68020		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		1.03	1.20		1.01	1.06
calls		1.41	1.41		1.27	1.27
compact	1.18			1.09		
diff		1.45	1.48		1.47	1.49
dry	0.98			0.99		
extract	3.26			2.49		
grep	1.08			1.03		
inltest		1.52	3.14		1.38	2.21
invert		1.05	1.07		1.02	1.02
mincost	1.08			1.02		
nroff		1.76	4.44		1.44	5.83
sed		1.88	2.05		1.39	1.63
sort	1.66			1.40		
tsp	1.02			1.00		
vpcc		1.65	6.47		1.82	8.46
whetstone	1.03			1.01		
yacc		2.26	3.01		1.94	2.40
AVERAGE	1.41	1.56	2.70	1.25	1.42	2.82

	Clipper			Convex		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		1.01	1.08		1.14	1.14
calls		1.48	1.45		1.11	1.11
compact	1.09			1.14		
diff		1.32	1.35		1.35	1.41
dry	1.00			1.00		
extract	1.63			2.25		
grep	1.06			1.14		
inltest		1.40	3.00		1.29	2.29
invert		1.01	1.02		1.00	1.00
mincost	1.00			1.00		
nroff		2.15	7.97		1.43	2.35
sed		1.19	1.26		1.50	1.50
sort	1.26			1.25		
tsp	1.00			1.00		
vpcc		1.42	4.11		1.23	*
whetstone	1.00			1.00		
yacc		1.15	1.21		1.21	1.64
AVERAGE	1.13	1.35	2.49	1.22	1.25	1.56

Table 2.9: Ratio of Inlined to Noninlined Program Code Section Size for Test Programs

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

	VAX-8600			MC68020		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	160.7	140.2	134.0	137.0	128.1	124.3
calls	7.7	7.6	7.7	8.8	8.4	8.0
compact	6.1	6.0		5.5	5.2	
diff	66.3	60.5	60.5	56.4	49.0	48.9
dry	8.8	5.1		7.6	6.0	
extract	1.7	1.7		1.7	2.0	
grep	1.2	1.2		1.2	1.2	
inltest	12.1	10.2	10.4	9.9	9.1	9.4
invert	23.9	22.4	22.2	21.0	20.4	20.3
mincost	10.6	6.6		10.6	8.5	
nroff	8.0	7.0	7.9	8.1	7.8	8.8
sed	21.0	20.3	18.7	16.8	17.3	15.5
sort	18.0	15.7		19.2	18.1	
tsp	265.5	234.4		1426.6	1410.2	
vpcc	86.8	79.9	92.4	76.1	74.6	79.2
whetstone	6.2	5.7		52.3	51.5	
yacc	6.2	5.9	6.1	6.8	6.7	6.7

	Clipper			Convex		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	237.6	226.7	230.7	173.6	167.3	153.8
calls	6.5	6.1	6.0	9.5	8.8	8.8
compact	6.3	5.9		6.4	6.6	
diff	83.8	86.5	85.2	71.5	69.1	69.0
dry	9.2	7.8		11.7	10.1	
extract	2.4	2.2		2.3	2.3	
grep	1.6	1.4		1.3	1.2	
inltest	22.7	22.1	21.2	15.2	13.0	12.7
invert	24.6	24.5	24.7	40.1	40.1	39.8
mincost	29.1	27.9		10.3	6.6	
nroff	6.0	5.9	6.6	8.9	8.2	8.4
sed	16.9	17.1	17.7	20.4	20.4	21.0
sort	17.0	17.1		22.2	18.9	
tsp	241.0	227.2		206.7	166.7	
vpcc	96.5	98.5	100.1	94.3	94.9	*
whetstone	5.9	5.5		3.6	3.2	
yacc	6.5	6.4	6.5	6.8	6.6	6.6

Table 2.10: Program Execution Times (in Seconds) for Test Programs

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

	VAX-8600			MC68020		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		0.87	0.83		0.94	0.91
calls		0.99	1.00		0.95	0.91
compact	0.98			0.95		
diff		0.91	0.91		0.87	0.87
dry	0.58			0.79		
extract	1.00			1.18		
grep	1.00			1.00		
inltest		0.84	0.86		0.92	0.95
invert		0.94	0.93		0.97	0.97
mincost	0.62			0.80		
nroff		0.88	0.99		0.96	1.09
sed		0.97	0.89		1.03	0.92
sort	0.87			0.94		
tsp	0.88			0.99		
vpcc		0.92	1.06		0.98	1.04
whetstone	0.92			0.98		
yacc		0.95	0.98		0.99	0.99
AVERAGE	0.86	0.92	0.94	0.95	0.96	0.96

	Clipper			Convex		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		0.95	0.97		0.96	0.89
calls		0.94	0.92		0.93	0.93
compact	0.94			1.03		
diff		1.03	1.02		0.97	0.97
dry	0.85			0.86		
extract	0.92			1.00		
grep	0.88			0.92		
inltest		0.97	0.93		0.86	0.84
invert		1.00	1.00		1.00	0.99
mincost	0.96			0.64		
nroff		0.98	1.10		0.92	0.94
sed		1.01	1.05		1.00	1.03
sort	1.01			0.85		
tsp	0.94			0.81		
vpcc		1.02	1.04		1.01	*
whetstone	0.93			0.89		
yacc		0.99	1.00		0.97	0.97
AVERAGE	0.93	0.99	1.00	0.88	0.95	0.95

Table 2.11: Ratio of Inlined to Noninlined Program Execution Time for Test Programs

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

For the computer systems and test programs considered, an average of 53% of a test program's executed calls were eliminated by inlining.

The code section sizes for the noninlined and inlined versions of the test programs are given in Table 2.8.[15] These values are computed as the difference between *etext*, which the UNIX linker sets to be the first address beyond a program's code section, and *start*, which is the program's entry point and which is always located at the beginning of the code linked.[16] (Unfortunately, the Convex linker rounds *etext*'s position up to the next 4K byte boundary, so the size quantities found on that computer system are not as precise as those found on the other three systems.) Table 2.9 presents the ratios of the test programs' inlined sizes to their noninlined. The average size ratio across the four computer systems for intramodule inlined versions of the test programs comprised of a single module is 1.25 and of the test programs comprised of multiple modules is 1.40. The average size ratio across the four computer systems for intermodule inlined versions of the (multiple-module) test programs is 2.39.

The execution times for the test programs are listed in Table 2.10. These values were gauged on the four computer systems as the average of three runs of the test programs' versions on particular test data sets monitored by the UNIX timing tool */bin/time*. For each run, the sum of the user and system times was recorded. Table 2.11 shows the ratios of the test programs' inlined execution times to their noninlined. The average execution time ratio across the four computer systems for intramodule inlined versions of the test programs comprised of a single module is 0.91 and of the test programs comprised of multiple modules is 0.96. The average execution time ratio across the four computer systems for intermodule inlined versions of the (multiple-module) test programs is 0.96.

The execution time and code section size data gathered for noninlined and inlined versions of the test programs raises several interesting questions. What factors influence the speed improvements usually associated with inlining? Why do some programs run slower after inlining? How does the increased size of an inlined version of a program affect its performance? Chapters 3, 4, and 5 address these questions.

---

[15]Programs composed of a single module do not have entries in the *Intermodule* columns of the table.

[16]The UNIX *size* utility was not used, because the values it reports are rounded up to the next page boundary.

CHAPTER 3

The Effect of Inlining on Program Execution Time

Formal study of the effect of inlining on program execution time has been lacking. To address the need for such research, this chapter presents equations created to represent the execution time performance of noninlined and inlined versions of a program. The accuracy of the equations' description of inlined execution time behavior is demonstrated on four computer systems. Using the equations, the influence of various factors on the speed of inlined versions of programs is examined. Insights of value to machine and compiler designers, gained via investigation with the equations, are also presented.

3.1. Execution Time Behavior Equations

The execution time behavior of a version of a program, noninlined or inlined, can be approximated by the equation

Program Execution Time = Call Overhead Time + Constant Program Time
+ Secondary Inlining Effects Time + Errors

where

Call Overhead Time = Call and Return Operation Time + Parameter Stack Adjustment Time
+ Local Variable Stack Adjustment Time + Parameter Passing Time + Register Save and Restore Time

and

Call and Return Operation Time = Count of Call and Return Operation Pairs Executed
× Time Cost for each Call and Return Operation Pair

Parameter Stack Adjustment Time = Count of Parameter-Passing Call Operations Executed
× Time Cost for Removing the Parameters from the Stack after Returning from each Call [1]

Local Variable Stack Adjustment Time = Count of Calls Executed to Subprograms with Stack-allocated Locals
× Time Cost for Adjusting the Stack to Accommodate Stack-allocated Locals

Parameter Passing Time = Count of Parameters Passed during the Program's Execution
× (Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site)

Register Save and Restore Time = Count of Registers Saved and Restored during the Program's Execution
× Time Cost for Saving and Restoring each Register

[1] On some machines (e.g., the VAX-11), this cost is incorporated into the cost for the return instruction.

Constant Program Time is the amount of time expended by the portion of the program's code that is not altered by the inlining of its subprograms. *Secondary Inlining Effects Time* is the time consumed or saved by changes associated with inlining other than the primary changes, which are incorporated in *Call Overhead Time*. Such secondary changes include register allocation variations, widened jump offset fields, widened frame offset fields, differences in the operation of the compiler's code optimizer, increases in code size, modified caching and paging behavior, subsumption of return values, restructuring of short-circuit expressions containing inlineable calls, and compile-time expression evaluation, along with dead code elimination, enabled by constant propagation into inlined code. The *Errors* term comprises errors in measuring and estimating costs, noise in the timing data gathered, and so on. For the computer systems on which the execution time behavior equations were verified, the *Errors* term was found to be small and it is treated as if equal to zero in the remainder of this section.

Two instances of the *Program Execution Time* equation are used to capture the effects on a particular program's execution time of inlining a certain set of call sites: one describes the program's noninlined execution time and the other describes the execution time of the specific inlined version of the program. The time cost components of *Call Overhead Time* for the equations are gauged empirically for a selected computer system; the count components of *Call Overhead Time* for the equations are collected by monitoring runs of the noninlined and inlined versions of the program. Using a measured value of the noninlined version's *Program Execution Time*, along with the fact that *Secondary Inlining Effects Time* is zero for the noninlined version, the equation representing the execution time of the noninlined version is solved for *Constant Program Time* as shown below.

$$\textit{Constant Program Time} = \textit{Program Execution Time}_{\textit{noninlined}} - \textit{Call Overhead Time}_{\textit{noninlined}}$$

The calculated *Constant Program Time* is then substituted into the inlined version's *Program Execution Time* equation. Hence, the equation representing the inlined version's execution time can be solved for *Program Execution Time* as follows.

$$\textit{Program Execution Time}_{\textit{inlined}} = \textit{Call Overhead Time}_{\textit{inlined}} + \textit{Constant Program Time} + \textit{Secondary Inlining Effects Time}$$

Comparison of the computed *Program Execution Time*$_{inlined}$ with its actual value indicates the accuracy

of the equation-form description of the inlined version's execution time.

3.2. Experimental Validation of the Execution Time Behavior Equations

The descriptive power of the equations was tested on the four computer systems and compilers listed in Table 2.1. The VAX-8600, MC68020, and Clipper systems use the callee-save convention, meaning that a called subprogram saves and restores the values of any non-scratch registers it uses, while the Convex system uses the caller-save convention, meaning that a calling subprogram saves and restores register values across calls. Automatic source-to-source C function inlining was performed by INLINER [DaH88], which was discussed in Chapter 2. The test programs on which the experiments with the equations were run are enumerated in Table 2.2.

Although C was chosen for these experiments, the execution time behavior equations are intended to apply to other procedural high-level languages as well. Some languages present fewer obstacles to inlining than C. For instance, Fortran 77 [Pra84] does not support recursion, and neither Fortran 77 nor Ada [Uni83] admits calls through pointers, so in these cases, the inlining restrictions that may be imposed for such language constructs need not be considered. The counts in the *Call Overhead Time*$_{inlined}$ term of the *Program Execution Time*$_{inlined}$ equation reflect the amount of inlining performed for any particular program. There are languages whose subprograms may contain additional overhead which can be removed via inlining. For example, Pascal [JeW78] and Ada allow subprograms to be defined within the lexical scope of other subprograms' definitions, so the inlining of a subprogram can transform that subprogram's references to variables outside its local stack frame into cheaper references to variables inside the local stack frame of the routine into which the subprogram is expanded. The *Secondary Inlining Effects Time* term of the *Program Execution Time*$_{inlined}$ equation should include inlining's elimination of this overhead. In contrast, attributes of languages other than C may afford less improvement from inlining than is achieved with C. A case in point is that the inlining of a Fortran 77 subprogram with multiple entry points could lead to the need for a goto at the inlined call site. Again, the *Secondary Inlining Effects Time* term of the *Program Execution Time*$_{inlined}$ equation should include this inlined program

behavior change.

To validate the execution time behavior equations' accuracy, the execution times projected by the equations for inlined versions of the test programs on the four computer systems were compared with the actual execution times for the inlined versions. To perform the comparison, a number of data items were gathered. *Program Execution Time*$_{noninlined}$ (used in solving the execution time behavior equations) and *Program Execution Time*$_{inlined}$ (used to check the accuracy of the equations' projected values) were measured on the aforementioned computer systems as the average of three runs of the test programs' versions on particular test data sets monitored by the UNIX timing tool */bin/time*.[2] The resulting values are presented in Section 4.1. The counts in the *Call Overhead Time* terms of the execution time behavior equations' noninlined and inlined instantiations were collected for runs of the test programs' versions on their test data, and the time costs in the *Call Overhead Time* terms were gauged for each target system.[3] A detailed account of the collection of the *Call Overhead Time* counts and time costs is presented in Appendix A. The *Secondary Inlining Effects Time* component of the execution time behavior equations for inlined programs was initially assumed to be negligible, and investigation into the inaccuracy of the consequent projected inlined execution times uncovered the constituents of *Secondary Inlining Effects Time* that had measurable impact.

One perceptible secondary inlining effect, mentioned in Chapter 2, arose from the fact that a variable allocated to a register in the noninlined version of a program may not be allocated to a register in an inlined version of that program. When a register variable in noninlined code is assigned to a memory location in inlined code, the change in the cost of referencing the variable can cause far-reaching repercussions on the inlined code's overall execution time relative to the noninlined. The results given in Section 4.2 suggest that inlining is more effective when a noninlined version's register variables are retained in registers after inlining, rather than being displaced to storage addresses. Hence, for this study, inlining

---

[2]Although the target computer systems have low resolution clocks, the test programs with their associated data sets execute in times at least two orders of magnitude greater than the clock resolution times.

[3]For each computer system, almost every time cost had multiple values, depending on characteristics of the data to which the corresponding machine-level operation was to be applied. (For example, *Time Cost for Saving and Restoring each Register* hinged on the number and type of the registers involved.) The frequency counts were gathered in accordance with the relevant data qualities.

is restricted so that register variables are protected, i.e., a call site is inlined only if any associated caller and callee variables placed in registers in the noninlined version of the program are also placed in registers in the inlined version.

Other palpable secondary inlining effects originated from interactions between inlining's placement of code within an executable program and a computer system's span-dependent instructions. Span-dependent instructions have limited application or vary in speed based on the number of bits needed to represent the distance to the target address. For the intermodule inlined version of *vpcc* on the VAX-8600, inlined function code placed in the *case* bodies of *switch* statements caused some *switch* statements to require long-offset jump tables. Since the machine's *case* instruction, which is used to implement jump tables, only allows word-offset jump tables, a more expensive set of machine instructions was required to implement the long-offset tables necessary. Also for the intermodule inlined version of *vpcc* on the VAX-8600, the long jump instruction was sometimes needed to replace the byte- or word-offset branch instruction which sufficed in all other cases. In addition, the VAX-8600's instruction set contains only byte-offset conditional branches; when a longer offset conditional branch is needed, a two-branch sequence is produced (consisting of a short conditional branch with the original test negated around a word-offset or long unconditional branch to the original target). Function bodies expanded between a branch and its target sometimes forced a byte-offset conditional branch in the noninlined version of the code to be mapped into a two-branch sequence in the inlined. Similarly, the VAX-8600's byte-offset *aobleq*, *aoblss*, *sobgeq*, and *sobgtr* instructions had to be replaced in inlined code by their component operations, when the distance to their target addresses increased sufficiently. On the Convex, code moved by inlining between a branch and its target caused some byte-offset branches in noninlined programs to become word- or long-offset effective address jumps in inlined versions. Machine designers should be aware of inlining's changes in the placement of code when estimating typical distances to target addresses.

The impact of span-dependent instructions on code speed is difficult to capture, because detecting the instructions requires machine code analysis and determining their frequency of use requires collection of execution information at a very low level, i.e., at the level of control paths through the code. For this

study, the effect of span-dependent instructions was removed from the code, allowing other issues related to inlining's effects on execution speed to be examined. In both the noninlined and inlined versions of *vpcc* on the VAX-8600, all *switch* statements were mapped to code sequences employing long-offset jump tables and all byte- and word-offset branches were mapped to long jump instructions. The VAX-8600's conditional branches were always implemented as two-branch sequences, and the appropriate component operations were always substituted for its *aobleq*, *aoblss*, *sobgeq*, and *sobgtr* instructions. On the Convex, byte-offset branches were, in all cases, replaced by word- or long-offset effective address jumps.

Additional noticeable secondary inlining effects were due to conflicts between the structure of inlined code and the design of the target compilers' code optimizers. Figure 3.1 illustrates an optimization performed by the VAX-8600 Portable C compiler which saves space but, by introducing an extra jump, can cost time. This common-sequences-before-jumping optimization was disabled for all compilations of both noninlined and inlined code, since inlining provided more opportunities to apply it. This transformation was also performed by the CONVEX C compiler optimizer, and since its source was not available to the authors, the optimizer was not used in compiling any versions of the test programs on the Convex. The Sun-3 C compiler sometimes did not apply certain optimizations of unconditional jumps to inlined code, apparently because of optimizer limits on the size of the window of machine instructions

C Code Fragment	Unoptimized Assembly Code			Optimized Assembly Code		
switch (x) {		...			...	
case 1:	L27:			L27:		
...		...			...	
status = 1;		movl	$1,-8(fp)	L2000000:	movl	$1,-8(fp)
break;		jbr	L25		jbr	L25
case 2:	L28:			L28:		
...		...			...	
status = 1;		movl	$1,-8(fp)		jbr	L2000000
break;		jbr	L25			
...		...			...	
}	L25:			L25:		

Figure 3.1: Example of an Optimization which Saves Space and Costs Time

considered for the optimizations; such optimizations were disabled when building noninlined and inlined programs. These issues suggest that compiler writers need to consider the effects of inlining when developing code optimization strategies.

After the elimination of these secondary effects, no others were found to generally alter inlined program execution time discernibly. Table 3.1 contains, for the three computer systems affected, the average ratio of the execution time for a program with secondary effects removed to the execution time for a program without these effects removed.[4] The table shows that eliminating these effects had nearly the same overall impact on noninlined and inlined code.

Note that the additional opportunities in inlined code to apply compile-time expression evaluation and return value subsumption were not cited as introducing significant secondary inlining effects. Although the effects of applying these techniques may explain errors in the projected times for some inlined versions of the test programs, the techniques did not appear to change inlined program performance across-the-board. Richardson and Ganapathi observed similar results for inlined code produced by their optimizing Pascal compiler [RiG89b].

The ratios of the projected to actual execution times for inlined versions of the test programs on the four computer systems are given in Table 3.2. Two kinds of inlined versions are included: intramodule inlined, in which inlining was performed separately within each module in the program, and intermodule inlined, in which inlining was performed across all the modules in the program. The equations accurately described the execution times of the inlined versions of the test programs on the selected computer

	Noninlined	Inlined
VAX-8600	1.08	1.07
MC68020	1.01	1.00
Convex	1.08	1.10

Table 3.1: Ave. Ratio of Time for Prgms. w & w/o Secondary Inlining Effects Removed

[4]Register variable displacement was prevented in both cases.

	VAX-8600		MC68020	
	Intramodule Inlined	Intermodule Inlined	Intramodule Inlined	Intermodule Inlined
cache	0.99	1.00	1.01	1.03
calls	0.98	0.98	0.93	0.93
compact	0.98		1.01	
diff	1.02	1.02	1.11	1.11
dry	0.85		1.00	
extract	0.98		0.97	
grep	1.08		0.93	
inltest	0.96	0.91	1.00	1.00
invert	0.99	0.98	0.98	0.99
mincost	0.98		1.06	
nroff	0.95	0.97	1.03	0.95
sed	1.01	0.99	0.98	1.00
sort	1.00		1.01	
tsp	0.97		1.01	
vpcc	1.03	0.98	1.02	1.00
whetstone	1.00		1.01	
yacc	1.03	1.04	1.01	1.00
AVERAGE	0.99		1.00	
STD DEV	0.04		0.04	

	Clipper		Convex	
	Intramodule Inlined	Intermodule Inlined	Intramodule Inlined	Intermodule Inlined
cache	1.02	0.99	1.03	1.04
calls	0.96	0.98	1.01	1.00
compact	1.02		1.02	
diff	0.93	0.98	1.02	1.02
dry	0.97		1.04	
extract	1.08		1.01	
grep	1.15		0.94	
inltest	0.99	1.04	1.04	0.98
invert	1.00	0.99	1.00	1.00
mincost	0.99		1.03	
nroff	0.98	0.93	1.10	0.90
sed	1.01	1.00	0.95	1.04
sort	0.99		1.02	
tsp	1.02		1.11	
vpcc	0.95	0.93	0.98	*
whetstone	0.98		0.89	
yacc	1.00	0.99	1.00	0.97
AVERAGE	1.00		1.01	
STD DEV	0.05		0.05	

Table 3.2: Ratio of Projected to Actual Time for Test Programs (Validation Build)

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

systems. The overall average ratio of projected to actual time was 1.00 with a standard deviation of 0.045. The quality of the results demonstrates that the value of the *Errors* term was reasonably low.

3.3. Examination of Factors Affecting the Speed of Inlined Code

The execution time behavior equations can be used to examine the influence of various factors on the speed of inlined versions of programs. For the experiments that were performed to validate the equations' accuracy, the difference between a particular program's $Program\ Execution\ Time_{noninlined}$ and $Program\ Execution\ Time_{inlined}$ can be traced to variations in the values of the five factors comprising the *Call Overhead Time* components in the noninlined and inlined instances of the execution time behavior equations. These cost factors, labeled f^1 through f^5, are as follows.

f^1: *Call and Return Operation Time*
f^2: *Parameter Stack Adjustment Time*
f^3: *Local Variable Stack Adjustment Time*
f^4: *Parameter Passing Time*
f^5: *Register Save and Restore Time*

The change in each of these factors' values resulting from inlining some set of a program's calls may be expressed as

$$\Delta_i = f^i_{noninlined} - f^i_{inlined} \qquad i \in \{1,2,3,4,5\}$$

and the total magnitude of the changes may be computed as

$$\Delta_{total} = \sum_{i=1}^{5} |\Delta_i|$$

The percentage of the total difference that is attributable to each factor may be represented as

$$\%_i = \frac{\Delta_i}{\Delta_{total}} \times 100 \qquad i \in \{1,2,3,4,5\}$$

where a positive value for $\%_i$ indicates that the time consumed by the corresponding factor is diminished in the inlined version of the program and a negative value for $\%_i$ indicates that the time consumed by the corresponding factor is increased in the inlined version of the program.

	Type of Inlining	VAX-8600					MC68020				
		$\%o_1$	$\%o_2$	$\%o_3$	$\%o_4$	$\%o_5$	$\%o_1$	$\%o_2$	$\%o_3$	$\%o_4$	$\%o_5$
cache	intramodule	81	0	1	10	8	68	7	0	2	23
	intermodule	82	0	1	10	6	72	8	0	3	18
calls	intramodule	78	0	3	7	13	46	4	0	2	48
	intermodule	78	0	3	7	13	46	4	0	2	48
compact	intramodule	72	0	4	9	15	39	4	0	2	55
diff	intramodule	63	0	3	16	17	38	7	0	3	52
	intermodule	64	0	3	16	17	38	7	0	3	52
dry	intramodule	77	0	1	12	10	58	6	0	3	34
extract	intramodule	76	0	1	14	8	58	7	0	3	32
grep	intramodule	1	0	0	0	-99	0	0	0	0	-100
inltest	intramodule	82	0	1	10	7	69	6	0	2	23
	intermodule	85	0	1	9	5	76	6	0	3	16
invert	intramodule	75	0	0	10	16	44	4	0	2	50
	intermodule	74	0	0	10	15	44	4	0	2	50
mincost	intramodule	82	0	1	17	0	85	10	0	5	0
nroff	intramodule	87	0	-1	6	6	58	4	0	1	-36
	intermodule	89	0	-4	6	-1	63	4	0	1	-32
sed	intramodule	87	0	3	10	0	19	2	0	1	-79
	intermodule	71	0	3	11	14	48	4	0	2	45
sort	intramodule	83	0	0	7	10	79	7	0	2	12
tsp	intramodule	77	0	3	20	0	70	9	0	5	15
vpcc	intramodule	87	0	0	10	3	84	7	0	3	6
	intermodule	85	0	0	10	4	80	7	0	3	10
whetstone	intramodule	85	0	3	13	0	100	0	0	0	0
yacc	intramodule	75	0	0	12	13	53	5	0	3	39
	intermodule	75	0	0	11	14	53	5	0	3	39
AVERAGE MAGNITUDE		76	0	2	11	12	57	5	0	2	35
AVERAGE VALUE		76	0	1	11	4	57	5	0	2	16

	Type of Inlining	Clipper					Convex				
		$\%o_1$	$\%o_2$	$\%o_3$	$\%o_4$	$\%o_5$	$\%o_1$	$\%o_2$	$\%o_3$	$\%o_4$	$\%o_5$
cache	intramodule	52	1	13	9	25	83	5	1	5	-6
	intermodule	52	1	9	8	30	79	4	1	5	10
calls	intramodule	73	0	2	11	14	31	2	1	2	-65
	intermodule	73	0	2	11	14	31	2	1	2	-65
compact	intramodule	22	1	9	8	61	73	4	3	7	12
diff	intramodule	25	3	0	13	58	11	1	0	2	-86
	intermodule	42	0	8	6	44	73	4	2	8	-12
dry	intramodule	57	2	7	14	20	76	4	1	5	14
extract	intramodule	64	0	10	10	15	65	4	2	7	-23
grep	intramodule	0	0	21	0	-79	66	4	1	3	-26
inltest	intramodule	79	0	2	8	11	75	4	1	5	-15
	intermodule	87	0	2	9	1	83	4	1	5	-6
invert	intramodule	70	0	7	9	-14	42	2	1	4	-50
	intermodule	61	0	9	8	21	63	4	2	7	-24
mincost	intramodule	65	1	4	15	15	82	5	0	10	-3
nroff	intramodule	61	0	2	4	-34	84	5	0	5	-5
	intermodule	41	0	-4	3	-53	58	3	-1	2	-36
sed	intramodule	14	0	-2	2	-82	15	1	0	2	-82
	intermodule	4	0	-6	0	-91	15	1	0	2	-82
sort	intramodule	75	0	2	3	19	60	3	0	3	33
tsp	intramodule	44	0	11	11	33	53	3	2	8	-35
vpcc	intramodule	59	0	13	7	20	88	4	0	6	1
	intermodule	59	0	11	8	21	*	*	*	*	*
whetstone	intramodule	36	3	6	12	43	86	4	3	8	0
yacc	intramodule	61	0	-2	10	26	84	5	1	8	-2
	intermodule	61	0	-2	8	28	79	5	1	7	-8
AVERAGE MAGNITUDE		51	1	6	8	34	62	3	1	5	28
AVERAGE VALUE		51	1	5	8	6	62	3	1	5	-22

Table 3.3: Percentage of Inlined Code Speed Difference by Factor

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

Table 3.3 presents the five factors' $\%_i$ values[5] calculated for inlined versions of the test programs run on the four computer systems considered in the previous section. For each system, the table also gives the average magnitude and value for the five factors' $\%_i$ quantities. Due to round-off errors, the sum $\sum_{i=1}^{5} |\%_i|$ does not equal 100 for all versions of all programs. On the four systems, savings in factor f^1, *Call and Return Operation Time*, contributed most significantly to the speed of inlined code as compared with noninlined. Averaged across the systems, the factor was responsible for 62% of the time changes engendered by the inlining transformations. The next most influential factor was f^5, *Register Save and Restore Time*. Inlining sometimes reduced and sometimes augmented the time utilized for this factor, with the factor's fluctuations in sign causing its average value to diverge from its average magnitude. The time for this factor typically rose on the Convex after inlining, because the Convex is a caller-save system on which all of the registers that are used by a function (rather than just those that are live) are saved and restored across call sites and on which the save and restore operations are not wisely placed in the code. Section 4.1 addresses these problems. The factor dominated the percentages of inlined code speed differences for test cases in which the time attributable to the factor grew and only a small proportion of the executed call sites were inlined.[6] For example, on the MC68020, *grep*'s time cost for the factor increased while, although a mean of 61% of the calls executed by a test program on that system were eliminated by inlining, fewer than 1% of *grep*'s executed calls were inlined. *Register Save and Restore Time* will be discussed further in the next chapter.

[5]The $\%_i$ values for factor f^2 on the VAX-8600 and for factor f^3 on the MC68020 are all zero; the factors are considered to have no costs on the specified systems, since the factors' costs on the systems are inextricably included in the costs for factor f^1.

[6]Calls remaining in inlined code include recursive calls, calls through pointers, calls to some runtime library routines, intermodule calls in intramodule inlined code, and calls which, if inlined, would cause register variables to be displaced to memory.

CHAPTER 4

Interactions between Inlining and Register Allocation

Aside from the negative effects often ascribed to the increased size of inlined code, inlining is believed to always be beneficial [Bal79]. However, this is not true. Interactions between inlining and register allocation can cause an inlined version of a program to run more slowly than its noninlined counterpart. This chapter discusses these interactions.

This chapter also identifies register allocation techniques suited to inlined code. Although these generally-applicable techniques have previously been described in the research literature, they have not become a standard feature of production compilers, in part because of concerns that they may yield, for a large increased expense in program compile time, only a slight incremental advantage in program execution time over the simple methods currently in widespread use. To some degree, however, the relative effectiveness of simple methods is dependent on the subprogram structure of noninlined code. This chapter discusses the characteristics of subprograms in inlined code that can amplify the need for more sophisticated register allocation approaches.

4.1. Increased Register Save and Restore Overhead in Inlined Code

Table 4.1 contains the actual execution times for noninlined and inlined versions of the test programs, built as described in Section 3.2, and Table 4.2 gives, for each program, the ratio of its inlined actual execution time to its noninlined. While on the average, inlined versions of the test programs ran 6% faster than noninlined, there are cases in which inlined code ran slower than noninlined. Using the execution time behavior equations to examine the influence of various factors on inlined program speed, the factor found to cause inlined code to run slower is *Register Save and Restore Time.* As shown in Section 3.3, the time consumed by saving and restoring registers at calls in inlined code can exceed the time

	VAX-8600			MC68020		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	157.2	145.1	138.1	133.7	127.3	123.4
calls	7.9	7.9	7.9	8.7	8.5	8.5
compact	6.4	6.1		5.5	5.2	
diff	69.2	63.6	63.5	56.3	48.5	48.6
dry	9.4	6.1		8.1	6.4	
extract	2.0	1.9		1.9	1.9	
grep	1.4	1.3		1.2	1.3	
inltest	12.3	10.7	10.8	9.5	8.7	8.6
invert	24.1	22.5	22.4	21.7	21.1	20.7
mincost	11.0	6.8		10.1	8.2	
nroff	8.7	7.7	7.9	8.1	7.7	8.3
sed	22.0	21.3	19.7	17.0	17.6	15.8
sort	20.1	17.6		18.3	17.3	
tsp	267.0	237.7		1385.8	1364.9	
vpcc	117.5	108.6	111.4	82.1	78.4	79.1
whetstone	6.3	5.6		51.9	51.2	
yacc	7.0	6.6	6.5	6.9	6.7	6.8

	Clipper			Convex		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	237.6	226.7	230.7	186.6	172.7	163.7
calls	6.5	6.1	6.0	8.6	8.6	8.7
compact	6.3	5.9		7.9	7.5	
diff	83.8	86.5	85.2	76.7	75.4	75.1
dry	9.2	7.8		12.1	10.5	
extract	2.4	2.2		2.6	2.5	
grep	1.6	1.4		1.5	1.6	
inltest	22.7	22.1	21.2	15.4	13.5	13.9
invert	24.6	24.5	24.7	40.1	40.1	39.9
mincost	29.1	27.9		11.0	7.1	
nroff	6.0	5.9	6.6	10.5	9.0	11.5
sed	16.9	17.1	17.7	23.0	25.0	22.9
sort	17.0	17.1		25.4	21.7	
tsp	241.0	227.2		195.3	162.0	
vpcc	96.5	98.5	100.1	107.4	105.1	*
whetstone	5.9	5.5		3.5	3.2	
yacc	6.5	6.4	6.5	7.8	7.6	7.9

Table 4.1: Program Execution Times (in Seconds) for Test Programs (Validation Build)

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

	VAX-8600			MC68020		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		0.92	0.88		0.95	0.92
calls		1.00	1.00		0.98	0.98
compact	0.95			0.95		
diff		0.92	0.92		0.86	0.86
dry	0.65			0.79		
extract	0.95			1.00		
grep	0.93			1.08		
inltest		0.87	0.88		0.92	0.91
invert		0.93	0.93		0.97	0.95
mincost	0.62			0.81		
nroff		0.89	0.91		0.95	1.03
sed		0.97	0.90		1.04	0.93
sort	0.88			0.95		
tsp	0.89			0.98		
vpcc		0.92	0.95		0.96	0.96
whetstone	0.89			0.99		
yacc		0.94	0.93		0.97	0.99
AVERAGE	0.85	0.93	0.92	0.94	0.96	0.95

	Clipper			Convex		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		0.95	0.97		0.93	0.88
calls		0.94	0.92		1.00	1.01
compact	0.94			0.95		
diff		1.03	1.02		0.98	0.98
dry	0.85			0.87		
extract	0.92			0.96		
grep	0.88			1.07		
inltest		0.97	0.93		0.88	0.90
invert		1.00	1.00		1.00	1.00
mincost	0.96			0.65		
nroff		0.98	1.10		0.86	1.10
sed		1.01	1.05		1.09	1.00
sort	1.01			0.85		
tsp	0.94			0.83		
vpcc		1.02	1.04		0.98	*
whetstone	0.93			0.91		
yacc		0.99	1.00		0.97	1.01
AVERAGE	0.93	0.99	1.00	0.89	0.96	0.98

Table 4.2: Ratio of Inlined to Noninlined Program Execution Time (Validation Build)

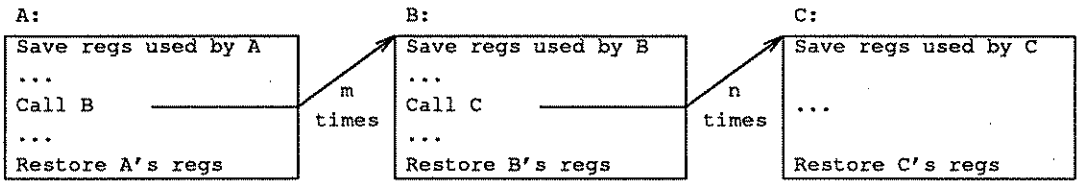
* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

consumed by saving and restoring registers at calls in noninlined code. Inlining can increase the number of times that registers are saved and restored by moving the save and restore operations to more frequently executed sites and by duplicating the save and restore operations.

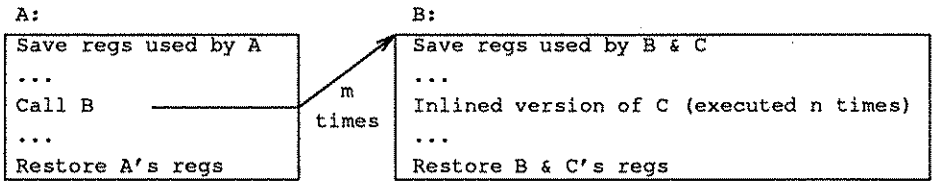
The fact that the number of times registers are saved and restored can be increased by inlining is an artifact of the way compilers typically produce code. On callee-save systems, compilers usually place the instruction(s) to save the values of the registers used by a subprogram in that subprogram's prologue code and the instructions(s) to restore the values of those registers in that subprogram's epilogue code. When a subprogram is inlined, its register save and restore operations are, in effect, moved to the prologue and epilogue of the subprogram into which the code is expanded. If, in this new location, the inlined subprogram's register save and restore operations are executed more frequently than they were in their location in the noninlined code, diminished performance can result. In this situation, if the compiler were to save the registers used in the inlined subprogram immediately before entering the inlined body and to restore them immediately after leaving it (thereby mimicking the save and restore behavior of the noninlined code), inlined program performance would not suffer. Chow has devised an appropriate register save and restore operation placement scheme based on data flow analysis; he characterizes the scheme as *shrink-wrapping* the saves and restores around the regions of the registers' activity [Cho88].

To illustrate these points, Figure 4.1 presents an example of some code in which inlining may increase the number of times registers are saved and restored on a callee-save system, and shows how the increase can be avoided. If the code sketched in Figure 4.1(a) is transformed, via inlining subprogram `C`, into the code sketched in Figure 4.1(b) and if the number of times `A` calls `B` exceeds the number of times `B` calls `C` (i.e., `m>n`), then the registers used by `C` are saved and restored more often after `C` is inlined (`m` times) than they were before it was inlined (`n` times). Were the compiler to place `C`'s register save and restore operations adjacent to its inlined body, as shown in Figure 4.1(c), then `C`'s registers would be saved and restored the same number of times when `C` is inlined (`n` times) as they were when it was not.

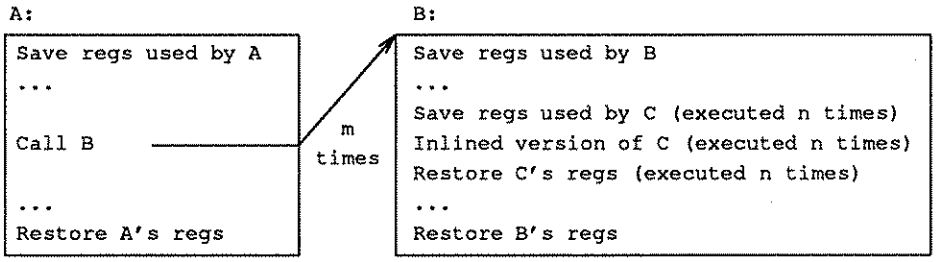
On caller-save systems, compilers usually place the instruction(s) to save the values of a subprogram's registers before each call in that subprogram and the instruction(s) to restore the values of



(a) No Call Sites Inlined



(b) Call to C Inlined into B, Register Save & Restore Overhead may Increase



(c) Call to C Inlined into B, Register Save & Restore Overhead is Same as Noninlined Version

Figure 4.1: Example of Register Save & Restore Placement on a Callee-Save System

those registers after each call. If all registers used by a subprogram are saved and restored around each call, inlining can cause the number of registers that are saved and restored to grow. The increase occurs when some but not all of the subprograms in a caller are inlined; the registers used only within inlined subprogram code are saved and restored across the calls not inlined, although the structure of the code ensures that such registers are dead across these calls. Were only live registers saved and restored across calls, this problem would not occur. Also, when a subprogram is inlined into a caller, the caller's register save and restore operations are moved and duplicated to surround any call sites in the inlined subprogram that were not expanded. As with the callee-save case, if the register saves and restores are exercised more often in the inlined code locations than they were in the noninlined code location, performance can be adversely affected. When this happens, were the compiler to save the registers used by the code

comprising what was the caller in the noninlined program immediately prior to entering inlined code and to restore them immediately after leaving it (imitating the save and restore behavior of the noninlined code), inlined program performance would not be penalized. An applicable general method using data flow analysis to determine the placement of register save and restore operations on caller-save systems, called *smarter caller*, was formulated by Davidson and Whalley [DaW89].

This section points out that, for both callee-save and caller-save systems, the diminished performance of inlined code associated with *Register Save and Restore Time* can be avoided if register save and restore operations are positioned as they would be in the noninlined version of the program. This is because the scheme that is typically used by compilers for the placement of register save and restore operations corresponds more closely in noninlined code to the registers' regions of activity than it does in inlined code. Inlined code is in greater need of schemes like *shrink-wrapping* or *smarter caller* to locate the register save and restore operations appropriately than noninlined code is.[1]

4.2. Displacement of Variables from Registers in Inlined Code

A variable that is placed in a register in the noninlined version of a program may not be placed in a register in a particular inlined version of that program. In producing the inlining results reported thusfar, INLINER was invoked with an option that specified avoiding any inlining that would cause variables assigned to registers in the noninlined version of a program to be displaced into memory locations in an inlined version. When this restriction on inlining is removed, inlined versions of programs containing frequently-used variables that are displaced from registers run measurably slower than projected by the equations describing inlined program execution time. Table 4.3 shows, for three such programs, how the ratio of projected to actual execution time diminishes when this restriction on inlining is not applied. Also, inlined versions containing frequently-used variables that are displaced from registers do not run as

[1]Noninlined code has little need for *shrink-wrapping*; Chow discovered that, on average, *shrink-wrapping* improved the execution time of noninlined code by less than 1% [Cho88]. Davidson and Whalley found *smarter caller* to be somewhat more effective; although the scheme did not change the number of instructions executed in noninlined code, it did reduce the number of memory references by more than 10% [DaW89].

	Built on VAX-8600 with register limitation on inlining		Built on VAX-8600 without register limitation on inlining	
	Intramodule Inlined	Intermodule Inlined	Intramodule Inlined	Intermodule Inlined
diff	1.02	1.02	0.96	0.93
sed	1.01	0.99	1.01	0.88
yacc	1.03	1.04	0.92	0.86

Table 4.3: Ratio of Projected to Actual Time for Prgms. w. Register Vars. Displaced

quickly as inlined versions built using the register limitation on inlining and, in fact, inlined versions with displaced register variables sometimes run more slowly than their noninlined counterparts. The data given in Table 4.4, which presents the three programs' ratios of inlined to noninlined execution times, demonstrates how inlined program performance can degrade when the register limitation on inlining is not employed.

No variable placed in a register in the noninlined version of a program need be displaced to memory in any inlined version. For example, noninlined and inlined versions of a program can always employ the exact same allocation of registers, with register spills and reloads being executed in inlined code at points corresponding to where register saves and restores are performed in noninlined code. Moreover, it is possible to do a better job of allocating registers in inlined code than in noninlined. Register assignments in inlined subprogram code can be tailored to the context of each site at which the

	Built on VAX-8600 with register limitation on inlining		Built on VAX-8600 without register limitation on inlining	
	Intramodule Inlined	Intermodule Inlined	Intramodule Inlined	Intermodule Inlined
diff	0.92	0.92	0.98	1.00
sed	0.97	0.90	0.95	0.98
yacc	0.94	0.93	1.01	1.09

Table 4.4: Ratio of Inlined to Noninlined Time for Prgms. w. Register Vars. Displaced

code is expanded, and the worst case assumptions compilers often make at a call site concerning the callee's accesses to parameters and global variables can be replaced with accurate information about such accesses when the callee's body is substituted for the call. However, achieving this hypothetical improvement in the allocation of registers for inlined code is challenging.

To perform effective register allocation for inlined code, several issues need to be addressed. First, register allocators usually operate on one subprogram at a time, allocating the machine's available registers to the set of variables belonging to that subprogram. Since inlining tends to cause the number of variables in each subprogram to proliferate, it increases the amount of competition for the available registers. Hence, inlining elevates the importance of carefully choosing which variables are placed in registers. Selection methods that incorporate variable usage counts [Fre74], ideally gathered by profiling the program, are appropriate. Second, the inline substitution of subprograms from different branches of the noninlined program's call graph often results in the introduction of a number of additional local variables with independent lifetimes. A technique such as register coloring [CAC81, Cha82], in which variables with non-overlapping lifetimes may be allocated to the same register, would be beneficial in these situations. Finally, an inlined version of a program is likely to contain many local variables with nested lifetimes, corresponding to the inlined bodies of subprograms from the same branch of the noninlined program's call graph. When there are too few registers to hold these variables simultaneously, a scheme such as live range splitting [ChH84, LaH86], which can divide the relatively long lifetimes of the variables declared at shallow nesting levels into multiple shorter lifetimes (thus allowing the more deeply-nested variables to share the registers), should be profitable.

Register allocation methods employing variable usage counts, register coloring, and live range splitting can be expensive. Computing variable usage counts requires measuring (or estimating) the execution frequency of all control flow paths in a program. Determining if a graph is n-colorable, where n is a natural number greater than 2, is NP-complete [AHU74], as is splitting a live range to minimize the number of jumps between blocks in the resulting live ranges [LaH86].[2] Particular characteristics of inlined code that were identified in this section may justify the costs of these methods.

[2]Researchers have included heuristics in their algorithms for solving these two problems [BCK89, LaH86].

CHAPTER 5

Code Placement Changes in Inlined Programs

Inlining alters the placement of code within a program. Multiple copies of subprograms expanded inline can cause an inlined version of a program to be longer than the noninlined version. Substitution of subprogram bodies for call operations can modify a program's locality. The effects of these code placement changes on inlined program behavior are addressed in this chapter.

5.1. The Effect of Inlined Code's Increased Size

The increased size of inlined code is often cited as negatively affecting inlined code's execution time performance. In a principal paper on subprogram inlining, Scheifler developed an inlining algorithm intended to minimize execution time within program size constraints [Sch77]. Bal and Tanenbaum incorporated this algorithm in the Amsterdam Compiler Kit [BaT86], and a similar algorithm was devised by Hwu and Chang [HwC89a]. Interprocedural data flow analysis has been suggested as a space-efficient alternative to subprogram inlining [Hal89]. The belief that there is inevitably a time cost associated with the increased size of inlined code is challenged by the data presented in this dissertation. Clearly it is true that if, as a result of inlining, a program becomes too large to fit in the address space of its target machine or to be handled by the tools on its host system, or if it consumes an unacceptable amount of space on a secondary storage device or requires more time to be preloaded than the time saved by inlining, then the increased size is detrimental to the inlined version of the program. However, these circumstances did not occur on the demand-paged virtual memory supermicro- and mini-computer systems on which the equations projecting inlined program performance were tested.

For the aforementioned systems, the equations demonstrated good descriptive power [DaH89], and did not include any factor which represented the inlined code's increased size attribute. Table 5.1

	VAX-8600				MC68020			
	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined
cache	10,996	11,376	10,988	13,292	35,704	36,056	35,624	37,784
calls	14,392	20,884	14,392	20,880	25,032	33,136	25,016	33,120
compact	11,972	14,208			22,688	24,632		
diff	27,756	41,080	27,752	41,880	51,520	76,824	51,456	77,712
dry	5,808	5,736			18,720	18,560		
extract	27,424	90,784			41,264	105,816		
grep	10,184	11,268			20,976	22,064		
inltest	41,108	61,976	41,100	127,292	68,144	94,904	67,128	149,080
invert	11,952	12,764	11,948	12,984	36,776	37,416	36,760	37,448
mincost	10,712	11,508			35,760	36,480		
nroff	49,704	89,564	49,700	223,060	70,432	105,240	69,664	431,304
sed	18,532	35,088	18,532	38,284	31,344	43,824	31,336	51,976
sort	13,628	23,460			24,072	34,176		
tsp	11,968	12,236			61,840	61,592		
vpcc	65,468	110,412	65,452	400,816	83,576	154,616	82,760	730,184
whetstone	10,984	11,292			60,408	60,848		
yacc	28,068	68,072	28,064	88,188	42,016	87,024	41,872	106,816

	Clipper				Convex			
	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined	Intramodule -Reordered Noninlined	Intramodule Inlined	Intermodule -Reordered Noninlined	Intermodule Inlined
cache	21,560	21,728	21,472	23,272	28,672	32,768	28,672	32,768
calls	19,768	29,344	19,744	28,680	40,960	45,056	40,960	45,056
compact	16,848	18,400			32,768	32,768		
diff	38,680	50,920	38,616	51,984	69,632	106,496	69,632	106,496
dry	12,176	12,176			20,480	20,480		
extract	34,752	56,664			53,248	126,976		
grep	15,296	16,288			32,768	32,768		
inltest	53,480	74,824	52,824	158,224	102,400	135,168	102,400	233,472
invert	23,128	23,352	23,112	23,568	32,768	32,768	32,768	32,768
mincost	21,912	21,960			28,672	28,672		
nroff	59,440	127,744	58,592	466,864	98,304	151,552	98,304	249,856
sed	24,112	29,872	25,096	31,600	45,056	69,632	45,056	69,632
sort	18,632	23,472			36,864	40,960		
tsp	24,952	24,992			49,152	49,152		
vpcc	70,720	100,432	70,016	287,616	135,168	176,128	*	*
whetstone	19,160	19,128			32,768	32,768		
yacc	36,952	42,424	36,792	44,632	61,440	77,824	61,440	102,400

Table 5.1: Program Code Section Sizes (in Bytes) for Test Programs (Validation Build)

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

	VAX-8600			MC68020		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		1.03	1.21		1.01	1.06
calls		1.45	1.45		1.32	1.32
compact	1.19			1.09		
diff		1.48	1.51		1.49	1.51
dry	0.99			0.99		
extract	3.31			2.56		
grep	1.11			1.05		
inltest		1.51	3.10		1.39	2.22
invert		1.07	1.09		1.02	1.02
mincost	1.07			1.02		
nroff		1.80	4.49		1.49	6.19
sed		1.89	2.07		1.40	1.66
sort	1.72			1.42		
tsp	1.02			1.00		
vpcc		1.69	6.12		1.85	8.82
whetstone	1.03			1.01		
yacc		2.43	3.14		2.07	2.55
AVERAGE	1.43	1.59	2.69	1.27	1.45	2.93

	Clipper			Convex		
	Intramodule Inlined		Intermodule Inlined	Intramodule Inlined		Intermodule Inlined
	1 module	>1 module	>1 module	1 module	>1 module	>1 module
cache		1.01	1.08		1.14	1.14
calls		1.48	1.45		1.10	1.10
compact	1.09			1.00		
diff		1.32	1.35		1.53	1.53
dry	1.00			1.00		
extract	1.63			2.38		
grep	1.06			1.00		
inltest		1.40	3.00		1.32	2.28
invert		1.01	1.02		1.00	1.00
mincost	1.00			1.00		
nroff		2.15	7.97		1.54	2.54
sed		1.19	1.26		1.55	1.55
sort	1.26			1.11		
tsp	1.00			1.00		
vpcc		1.42	4.11		1.30	*
whetstone	1.00			1.00		
yacc		1.15	1.21		1.27	1.67
AVERAGE	1.13	1.35	2.49	1.19	1.31	1.60

Table 5.2: Ratio of Inlined to Noninlined Program Code Section Size (Validation Build)

* Due to a bug in the CONVEX C Compiler, the intermodule inlined version of *vpcc* could not be built on the Convex.

contains the code section sizes of noninlined and inlined versions of the test programs, built as described in Section 3.2, and Table 5.2 presents the ratios of inlined to noninlined code section sizes for the test programs.[1] While register variable protection limited the size of the inlined code somewhat, certain inlined programs were 3 to 8 times larger than their noninlined counterparts without noticeable impact on their execution speed. Hence, on some computer systems, neither the use of an inlining algorithm that includes program size constraints, which is likely to be expensive,[2] nor the substitution of interprocedural data flow analysis for inlining, which is less beneficial in general [RiG89b], is warranted.

Interestingly, size differences between a program's noninlined and inlined versions can cause differences in the versions' execution control flow. A program's size can affect the location of its heap variables, which in turn can affect its behavior. For instance, the addresses of heap items allocated by the UNIX *malloc* routine depend, in many implementations, on the length of the program's code section. In the *inltest* test case, positions of symbol table entries are based on a hash function that operates on heap addresses, and since the lengths of the noninlined and inlined versions' code sections are unequal, the symbol table entries' positions in the versions are not the same. This variation causes the execution paths taken through the versions to differ slightly. Also, a program's size can affect the amount of heap space available to it, which can alter its behavior. For example, the *sort* test program was originally written using the UNIX *sbrk* and *brk* calls to obtain as large a segment of heap memory as possible; on a number of UNIX systems, code section length determines how much heap memory can be allocated in this fashion. The amount of heap memory acquired by a version of *sort* influenced how that version partitioned work and how it employed intermediate files. Because of the disparity in the code section lengths of the noninlined and inlined versions, there were significant dissimilarities in their operation. These differences interfered with measuring the basic effects of inlining on *sort*'s performance, and were eliminated by modifying *sort* to always allocate a 20K-byte block of heap space.

---

[1]Noninlined versions of the test programs are built with intramodule- and intermodule-reordering, as explained in Section 2.3. In computing the ratios of inlined to noninlined code section sizes, intramodule inlined sizes are divided by intramodule-reordered noninlined sizes and intermodule inlined sizes are divided by intermodule-reordered noninlined sizes.

[2]The problem of finding a sequence of inline substitutions that minimizes execution time subject to program size constraints was shown by Scheifler [Sch77] to be NP-complete.

5.2. The Effect of Inlining on Code Locality

The phenomenon that a program's past memory accesses are related to its future memory accesses is a characteristic of program behavior called locality. Spatial locality denotes the tendency of a program to refer to addresses near those recently referenced, and temporal locality denotes the tendency of a program to refer to the same addresses as those recently referenced. The transformations comprising inlining can change a program's locality. The replacement of executed calls by copies of the corresponding subprogram bodies can improve a program's spatial locality, while possibly diminishing its temporal locality, when called subprograms are expanded at more than one site. Inline substitution for calls that are not executed may decrease a program's spatial locality. A program's locality affects its behavior with respect to caching and paging systems.

The effect of inlining on a program's caching and paging behavior was not found to perceptibly alter its execution time. The execution time behavior equations, which were verified on machines with caching and paging systems, accurately described the execution speed of inlined code without including caching and paging behavior factors. To gauge inlining's effects on caching and paging behavior precisely, measurement methods based on trace-driven simulation were used.

Due to the expense of obtaining a trace of the addresses referenced by a program,[3] a subset of the test programs listed in Table 2.2, displaying a range of relative sizes and speeds when inlined, was used for the caching and paging behavior experiments. This study was not able to benefit from methods developed for speeding up trace-driven simulation. Trace sampling [LPI88], which is a heuristic technique employed to reduce the size of trace files collected for assessing competing cache memory system designs, eliminates the details of program operation needed to compare the caching and paging behavior of noninlined and inlined versions of a specific program. Statistical modeling of reference patterns [Sto87] is not applicable, because it is not known how to characterize the differences in the address sequences in noninlined and inlined code.

---

[3]Tracing (using UNIX *ptrace* system calls), along with caching and paging behavior simulation processing, took several days per version of each program.

5.2.1. Address Reference Traces for Noninlined and Inlined Code

The address traces used in comparing the caching and paging behavior of noninlined and inlined versions of test programs, built as described in Section 3.2, were gathered for runs on the programs' test data on the MC68020. Each memory address accessed by a version was reported, along with the size (1, 2, 4, 8, or 12 bytes) of the item accessed, the stream (instruction, (non-stack) data, or stack) to which the item belonged, and the kind of access (read, write, or modify) to the item. A particular machine instruction references one or more instruction locations and zero or more data and/or stack locations.

Table 5.3 contains the lengths of the address traces recorded. In all cases, inlined versions of the programs executed fewer instructions than the corresponding noninlined versions, although for some programs, an inlined version's use of longer variants of certain instructions caused the count of its instruction references to exceed the count of the noninlined version's instruction references. The quantity of

	Number of Instructions	Counts of References by Stream			Number of References
		Instruction	Data	Stack	
compact:					
Noninlined	19,643,471	34,452,467	4,547,876	8,532,318	47,532,661
Intramodule Inlined	18,989,540	34,438,960	4,547,708	7,485,627	46,472,295
extract:					
Noninlined	7,133,705	10,391,112	1,114,322	3,519,652	15,025,086
Intramodule Inlined	6,942,836	10,441,720	1,114,319	3,048,310	14,604,349
grep:					
Noninlined	5,961,124	8,672,836	1,835,426	174,813	10,683,075
Intramodule Inlined	5,961,052	8,721,683	1,835,426	224,170	10,781,279
mincost:					
Noninlined	32,197,544	57,656,590	7,952,753	20,796,210	86,405,553
Intramodule Inlined	24,917,450	50,057,900	9,311,379	10,801,419	70,170,698
nroff:					
Noninlined	26,121,516	46,033,049	6,152,161	6,326,586	58,511,796
Intramodule Inlined	24,334,241	43,832,551	6,150,595	7,002,472	56,985,618
Intermodule Inlined	23,958,622	43,289,514	6,148,898	6,701,184	56,139,596
yacc:					
Noninlined	26,867,541	44,266,682	9,440,756	5,569,899	59,277,337
Intramodule Inlined	26,459,530	43,696,328	9,481,826	5,157,816	58,335,970
Intermodule Inlined	26,456,710	43,758,726	9,539,416	5,042,144	58,340,286

Table 5.3: Lengths of Address Traces

data references varied across a program's noninlined and inlined instantiations. Inlining increased the number of data references at expanded call sites for which globals used as actual parameters were copy-propagated, and decreased the number of data references at expanded call sites for which accesses to globals used as return values were eliminated because the return values were ignored. For the majority of programs, the stack reference count for an inlined version was less than for the noninlined, since inlining removes the stack references related to subprogram linkage. However, the volume of stack references sometimes increased for inlined versions in which the inlining transformations resulted in the execution of more register save and restore operations than were performed by the original program, as discussed in Section 4.1.

5.2.2. Caching Behavior of Inlined Code

Richardson and Ganapathi mention that an inlined program tends to have a higher instruction cache miss ratio than its noninlined counterpart [RiG89a]. However, the cache miss (or hit) ratio computed on a program's address trace, which is often used for evaluating alternative cache designs, is inappropriate for comparing the caching behavior of nonlined and inlined versions of a program. The reason the cache miss ratio is not suited to this application is that the ratio does not measure the overall utilization of the cache by a program's noninlined and inlined versions, whose address traces are typically not the same length. For example, suppose that the intramodule inlined version of the test program *mincost* has a higher cache miss ratio (0.035) than the noninlined version (0.031). Applying these miss ratios to the number of references in the address traces for the versions, given in Table 5.3, the intramodule inlined version would incur 2,455,974 misses and 67,714,724 hits, and the noninlined version would incur 2,678,572 misses and 83,726,981 hits. In this scenario, although the inlined version has a higher cache miss ratio, it engenders fewer misses and fewer hits than the noninlined version, and can be viewed as having better caching performance, since the two versions are functionally equivalent and the inlined version requires less cache processing on its behalf.

A caching behavior measure which captures the amount of cache processing done on behalf of a program was developed. Reflecting published data on the relative costs of cache hits and misses [Smi82], the *caching work measure* is computed on a program's address trace as the sum of the number of cache hits multiplied by one work unit and the number of cache misses multiplied by ten work units. The caching work measure can be calculated for noninlined and inlined versions of a program, and the version having the lowest value is said to have the best caching behavior.

For the comparison of noninlined and inlined code caching behavior presented in this paper, the caching work measure was computed with respect to a set-associative cache of size 16K bytes, consisting of 256 sets of 4 lines each, with 16 bytes per line. A 16K-byte cache is smaller than many caches in current use [DeS90], but a caching system with that size cache is considered to be more sensitive to program behavior than a system with a larger cache. A unified instruction and data cache was utilized, employing the least-recently-used replacement algorithm. As was done in Smith's caching system studies [Smi82], context-switches were simulated by invalidating the entire cache every 10,000 units of time, with each hit consuming 1 unit of time and each miss consuming 10 units of time. Write accesses did not cause a line fetch on a miss, and all writes were considered to be write-through with no delay.

The caching work measure values found for noninlined and inlined versions of several test programs are listed in Table 5.4.[4] The cache miss ratios for some inlined programs were greater than the ratios for their noninlined versions, but the caching work measures for all except one of the inlined programs were less than the measures for their noninlined versions. Table 5.5 shows the ratios of the caching work measures for the inlined versions to the measures for the noninlined versions. The ratios indicate that the improvement in the inlined versions' caching behavior is, in most cases, slight. These results extend those of Hwu and Chang [HwC89b], who found that programs transformed using inline expansion and several instruction placement techniques had better miss ratios for direct-mapped instruction caches of various sizes than the miss ratios suggested by Smith [Smi87] as design targets for fully-associative caches of the same sizes.

---

[4]A version's number of cache accesses, given in Table 5.4, exceeds its number of references, given in Table 5.3, because references that modify a location are treated as two cache accesses, a read access followed by a write access.

	Cache Hits	Cache Misses	Number of Cache Accesses	Cache Miss Ratio	Caching Work Measure
compact:					
Noninlined	48,076,861	969,353	49,046,214	0.020	57,770,391
Intramodule Inlined	47,075,440	910,408	47,985,848	0.019	56,179,520
extract:					
Noninlined	15,389,792	406,956	15,796,748	0.026	19,459,352
Intramodule Inlined	14,962,293	413,715	15,376,008	0.027	19,099,443
grep:					
Noninlined	11,043,916	65,802	11,109,718	0.006	11,701,936
Intramodule Inlined	11,140,453	67,469	11,207,922	0.006	11,815,143
mincost:					
Noninlined	84,910,138	2,733,665	87,643,803	0.031	112,246,788
Intramodule Inlined	68,942,264	2,466,220	71,408,484	0.035	93,604,464
nroff:					
Noninlined	57,973,438	2,033,743	60,007,181	0.034	78,310,868
Intramodule Inlined	56,346,933	2,135,320	58,482,253	0.037	77,700,133
Intermodule Inlined	55,115,963	2,271,246	57,387,209	0.040	77,828,423
yacc:					
Noninlined	58,754,480	1,303,272	60,057,752	0.022	71,787,200
Intramodule Inlined	57,867,813	1,248,118	59,115,931	0.021	70,348,993
Intermodule Inlined	57,878,424	1,235,629	59,114,053	0.021	70,234,714

Table 5.4: Caching Behavior Measurements

	Intramodule Inlined	Intermodule Inlined
compact	0.97	
extract	0.98	
grep	1.01	
mincost	0.83	
nroff	0.99	0.99
yacc	0.98	0.98

Table 5.5: Ratios of Inlined to Noninlined Caching Work Measure Values

5.2.3. Paging Behavior of Inlined Code

Like the cache miss ratio, the page fault rate, which is used to examine candidate virtual memory system designs, does not lend itself to the comparison of noninlined and inlined program paging behavior, because the rate does not embody the overall utilization of the paging system by the competing versions of the program. A work measure for evaluating paging behavior similar to the one developed

for evaluating caching behavior is desired, but formulating a work measure for paging behavior is more difficult. Unlike the number of cache lines available to a program, which is fixed at the number supplied by the target machine's hardware, the number of physical pages available to a program is usually fewer than the number of hardware pages and depends on several factors, including the behavior of the program, the policies of the operating system, and the demands of competing processes. However, regardless of the specific operation of a given virtual memory scheme, a fundamental goal of most schemes is to keep a program's working set of pages [Den80] in physical memory. Hence, a program's working set size can be adopted as a bound on its resident set size, and its paging behavior can be characterized by a work measure based on the number of hits and misses to its resident set.

Using this idea, a paging behavior measure which represents the amount of paging work done on behalf of a program was developed. Reflecting possible relative costs of resident set hits and misses, the *paging work measure* is computed on a program's address trace as the sum of the number of resident set hits multiplied by one work unit and the number of resident set misses multiplied by one hundred work units.[5] Since a program's working set size varies during its execution and since a virtual memory scheme may not always be successful at keeping a program's working set in physical memory, the paging work measure is figured for several maximum resident set size values.

The paging work measure values presented in this paper were calculated with respect to a paging system with a page size of 4096 (2^{12}) bytes, which falls in the range of typical page sizes in available systems [PeS86, Tan87]. The chosen page size is consistent with the cache size specified in the previous section, since certain aspects of cache look-up using the low-order 12 bits of an address (8 bits for the set number and 4 bits for the byte within the line) could be performed on the hypothetical machine in parallel with virtual address translation. The average working set size, the associated standard deviation, the minimum working set size, and the maximum working set size were computed with respect to windows of 10,000 references [MaD74, PeS86] in a test program's trace, and are presented in Table 5.6. A paging system which bounds the maximum size of a program's resident set, removing the least-recently-used

[5] A two order of magnitude difference between the cost of a resident set hit and of a miss implies that some misses do not engender secondary storage device accesses, i.e., that for some page faults, the referenced page is available in main memory, perhaps in a buffer associated with the secondary storage device or on the free list.

	Average Working Set Size	Standard Deviation	Minimum Working Set Size	Maximum Working Set Size
compact:				
Noninlined	9	1	3	15
Intramodule Inlined	9	1	3	16
extract:				
Noninlined	10	5	6	21
Intramodule Inlined	12	6	6	25
grep:				
Noninlined	5	1	3	12
Intramodule Inlined	6	1	4	12
mincost:				
Noninlined	9	2	5	13
Intramodule Inlined	10	2	5	13
nroff:				
Noninlined	17	3	3	28
Intramodule Inlined	19	4	3	32
Intermodule Inlined	23	5	3	44
yacc:				
Noninlined	13	6	3	29
Intramodule Inlined	13	6	3	34
Intermodule Inlined	13	6	3	31

Table 5.6: Working Set Size Quantities Found for Reference Traces

page from the resident set whenever a page fault occurs and the size of the set is at its specified limit, was simulated.

The number of resident set hits and misses found for noninlined and inlined versions of several test programs are listed in Table 5.7,[6] and the paging work measure values are enumerated in Table 5.8. The quantities were computed with respect to five limits on the resident set size: the program's minimum working set size, the difference of its average working set size and one standard deviation, its average working set size, the sum of its average working set size and one standard deviation, and the program's maximum working set size.[7] Table 5.9 provides the ratios of the inlined paging work measures to the noninlined. In a typical operating situation, in which the maximum resident set size for each version of

[6]A program's total number of resident set references, given in Table 5.7, is greater than the number of references in its address trace, given in Table 5.3, when any of its references cross a page boundary.

[7]For the noninlined version of *extract*, the average working set size minus one standard deviation was less than the minimum working set size, and for the intramodule inlined version of *extract*, the average working set size minus one standard deviation was equal to the minimum working set size.

the test programs might be equal to that version's average working set size, the programs' inlined versions showed somewhat better paging behavior, in most instances, than their noninlined versions.[8] When the maximum resident set size for each version was fixed at its maximum working set size, in a setting representative of an unloaded system or one with a generous supply of physical memory, inlined versions usually demonstrated slightly better paging behavior as well. In addition, when each version's maximum resident set size was set to its minimum working set size, an assignment intended to suggest the versions' paging behavior under the memory constraints of a heavily loaded system or one with a severe shortage of physical memory, the majority of the test programs' inlined versions had lower paging work measure values than their noninlined counterparts. For this last case, since all of the test programs but one had identical minimum working set size values for their noninlined and inlined versions, nearly every program had the same stringent resident set size restriction applied to its noninlined and inlined versions.

	Minimum Working Set Size		Ave. Working Set Size - Std. Deviation		Average Working Set Size		Ave. Working Set Size + Std. Deviation		Maximum Working Set Size		Number of Resident Set References
	Hits	Misses	Hits	Misses	Hits	Misses	Hits	Misses	Hits	Misses	
compact:											
Noninlined	45,958,174	1,574,487	47,510,557	22,104	47,532,611	50	47,532,617	44	47,532,642	19	47,532,661
Intramodule Inlined	45,046,420	1,425,875	46,450,161	22,134	46,472,227	68	46,472,242	53	46,472,275	20	46,472,295
extract:											
Noninlined	14,988,613	36,473	14,970,906	54,180	15,015,174	9,912	15,023,903	1,183	15,025,037	49	15,025,086
Intramodule Inlined	14,566,666	37,683	14,566,666	37,683	14,594,226	10,123	14,600,510	3,839	14,604,239	110	14,604,349
grep:											
Noninlined	10,171,455	511,620	10,657,784	25,291	10,682,849	226	10,682,928	147	10,683,063	12	10,683,075
Intramodule Inlined	10,706,423	74,856	10,747,848	33,431	10,781,110	169	10,781,200	79	10,781,267	12	10,781,279
mincost:											
Noninlined	85,162,983	1,242,570	86,177,609	227,944	86,375,314	30,239	86,404,725	828	86,405,251	302	86,405,553
Intramodule Inlined	68,797,764	1,372,934	70,011,416	159,282	70,167,366	3,332	70,170,145	553	70,170,396	302	70,170,698
nroff:											
Noninlined	54,328,135	4,184,109	58,275,546	236,698	58,498,591	13,653	58,506,769	5,475	58,511,644	600	58,512,244
Intramodule Inlined	52,866,295	4,119,771	56,881,881	104,185	56,967,576	18,490	56,977,092	8,974	56,984,193	1,873	56,986,066
Intermodule Inlined	51,386,096	4,753,948	56,016,542	123,502	56,114,163	25,881	56,125,485	14,559	56,137,032	3,012	56,140,044
yacc:											
Noninlined	52,187,176	7,090,161	58,910,865	366,472	59,236,782	40,555	59,268,121	9,216	59,277,032	305	59,277,337
Intramodule Inlined	50,109,827	8,226,143	57,943,846	392,124	58,289,495	46,475	58,322,698	13,272	58,335,845	125	58,335,970
Intermodule Inlined	52,149,809	6,190,477	58,006,725	333,561	58,285,325	54,961	58,328,913	11,373	58,340,105	181	58,340,286

Table 5.7: Number of Resident Set Hits and Misses Found for Reference Traces

[8]When the paging work measure is computed using a three or four order of magnitude difference between the cost of a resident set hit and of a miss [PaH90], the inlined versions of these programs display virtually the same paging behavior as the corresponding noninlined versions.

	Minimum Working Set Size	Ave. Working Set Size - Std. Deviation	Average Working Set Size	Ave. Working Set Size + Std. Deviation	Maximum Working Set Size
compact:					
Noninlined	203,406,874	49,720,957	47,537,611	47,537,017	47,534,542
Intramodule Inlined	187,633,920	48,663,561	46,479,027	46,477,542	46,474,275
extract:					
Noninlined	18,635,913	20,388,906	16,006,374	15,142,203	15,029,937
Intramodule Inlined	18,334,966	18,334,966	15,606,526	14,984,410	14,615,239
grep:					
Noninlined	61,333,455	13,186,884	10,705,449	10,697,628	10,684,263
Intramodule Inlined	18,192,023	14,090,948	10,798,010	10,789,100	10,782,467
mincost:					
Noninlined	209,419,983	108,972,009	89,399,214	86,487,525	86,435,451
Intramodule Inlined	206,091,164	85,939,616	70,500,566	70,225,445	70,200,596
nroff:					
Noninlined	472,739,035	81,945,346	59,863,891	59,054,269	58,571,644
Intramodule Inlined	464,843,395	67,300,381	58,816,576	57,874,492	57,171,493
Intermodule Inlined	526,780,896	68,366,742	58,702,263	57,581,385	56,438,232
yacc:					
Noninlined	761,203,276	95,558,065	63,292,282	60,189,721	59,307,532
Intramodule Inlined	872,724,127	97,156,246	62,936,995	59,649,898	58,348,345
Intermodule Inlined	671,197,509	91,362,825	63,781,425	59,466,213	58,358,205

Table 5.8: Paging Work Measure Values

	Minimum Working Set Size	Ave. Working Set Size - Std. Deviation	Average Working Set Size	Ave. Working Set Size + Std. Deviation	Maximum Working Set Size
compact:					
Intramodule Inlined	0.92	0.98	0.98	0.98	0.98
extract:					
Intramodule Inlined	0.98	0.90	0.98	0.99	0.97
grep:					
Intramodule Inlined	0.30	1.07	1.01	1.01	1.01
mincost:					
Intramodule Inlined	0.98	0.79	0.79	0.81	0.81
nroff:					
Intramodule Inlined	0.98	0.82	0.98	0.98	0.98
Intermodule Inlined	1.11	0.83	0.98	0.98	0.96
yacc:					
Intramodule Inlined	1.15	1.02	0.99	0.99	0.98
Intermodule Inlined	0.88	0.96	1.01	0.99	0.98

Table 5.9: Ratios of Inlined to Noninlined Paging Work Measure Values

CHAPTER 6

Conclusion

This chapter begins with a summary of the results presented in this dissertation. Some areas for related future work are then identified.

6.1. Summary of Results

Equations were formulated to represent the execution time performance of noninlined and inlined versions of a program. The accuracy of the equations' description of inlined program execution time behavior was demonstrated for a set of test programs on four computer systems. Using the equations, the influence of various factors on the speed of inlined code was examined.

The increased size of inlined code, built as discussed in Section 3.2, did not affect its execution time on the demand-paged virtual memory machines employed in this research. Inlined program speed was accurately described on these machines by equations that did not include any factors representing the size growth associated with inlining. Subprogram inliners written for such machines need not take code size growth into account when choosing call sites to inline. Also, inlining can be used to subsume interprocedural optimization on such machines. Inlining was found to slightly improve the caching and paging behavior of test programs. This observation was based on new measures, computed on program address traces, that were designed for comparing caching and paging behavior across versions of a program.

Register save and restore time was shown to sometimes have a negative impact on the performance of inlined code, causing an inlined version of a program to run slower than the noninlined version in some cases. The importance of ensuring that variables assigned to registers in the noninlined version of a program are not displaced to memory locations in an inlined version was demonstrated. A subprogram

inliner could benefit by being accompanied by a compiler which employs the *shrink-wrapping* or *smarter caller* scheme for placing register save and restore operations at desirable points in the code and which allocates registers using variable usage counts, register coloring, and live range splitting.

Information acquired in developing INLINER, the source-to-source C function inlining tool that supports much of the work in this dissertation, was presented. INLINER's transformations for C statements were outlined, including mappings for iterative statements with inlineable calls in their test or update expressions, return statements in functions being inlined, and statements combining inlineable calls with nested function calls, comma operators, or short-circuit operators. The effect of inlining on C data declarations was also explained, the most important topics being the renaming of data items and the merging of data from multiple input modules into a single output module. In addition, inlining issues related to C constructs were noted. Of particular interest was the inlining problem posed by call sites at which more actual parameters are supplied than the callee is declared to accept. Furthermore, an optimization that supplements inlining's effects was devised, in which a caller's expression or return statement is moved into and possibly duplicated in an inlined callee's body so as to eliminate both the local variable which would replace the inlined function's return value and the assignments to that local. Moreover, in connection with implementing INLINER, a method for promoting its efficiency was formulated, in which subprograms are processed in "callee precedes caller" sorted order.

Machine designers should be aware that inlining can increase the typical distance to a branch target address in the object code. Compiler writers need to recognize that inlining can augment the data manipulated during code optimization.

6.2. Areas for Future Work

Three areas for future work are addressed in the following paragraphs. One area is the inlining of calls to recursive subprograms. Another is the performance of inlining in conjunction with particular global optimizations. The third is experimentation with a register allocator intended for inlined code.

The inlining of calls to recursive subprograms should prove more fruitful than is currently recognized. Scheifler's inlining system would repeatedly substitute a recursive subprogram inline until a specified size limit was reached, at which point calls to the original copy of the subprogram were produced. He found that [Sch77]:

> Size grew very quickly as the recursive invocations were expanded, but the expected number of calls decreased rather slowly.

The effect of Scheifler's inlining transformation for a recursive subprogram on the volume of executed calls is represented by the dynamic call graphs given in Figure 6.1. The number of recursive calls eliminated by inlining is equal to the depth to which the recursive subprogram is inlined into its caller. Alternatively, in a manner analogous to loop unrolling, the recursive subprogram could be inlined into its own callable body, as shown in Figure 6.2. For this method, inlining diminishes the number of recursive calls by a factor of the depth to which the recursive subprogram is inlined. INLINER could be modified to perform inline expansion of recursive calls in the proposed fashion, and empirical study could be used to verify the technique's effectiveness.

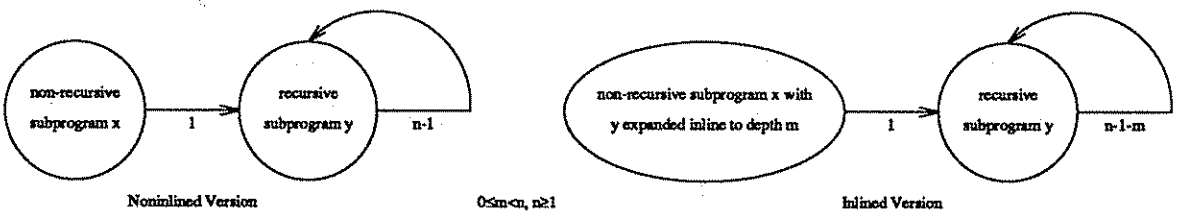


Figure 6.1: Dynamic Call Graphs for Program w. Recursion (Scheifler's Method)

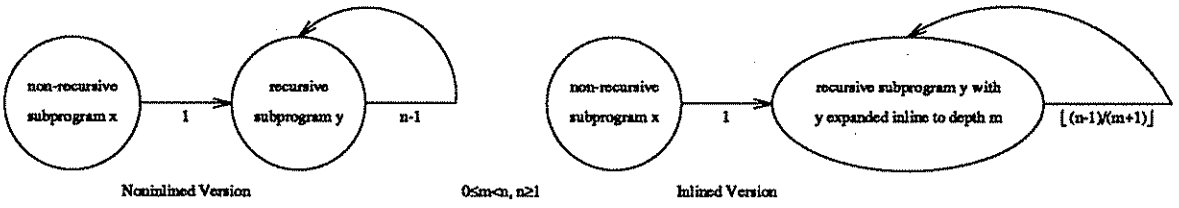


Figure 6.2: Dynamic Call Graphs for Program w. Recursion (Proposed Method)

The interaction of subprogram inlining with certain global optimizations could be examined. Optimizations such as loop invariant code motion and strength reduction may affect the *Secondary Inlining Effects Time* term of the execution time behavior equations. The union of inlining and selected global optimizations is expected to produce good code, but may only be worthwhile for some software applications. Richardson and Ganapathi have recorded large increases in compile time when inlining is combined with their global optimizer [RiG89b].

Experimentation with a register allocator suited to inlined code would be interesting. As previously discussed, such a register allocator would employ the *shrink-wrapping* or *smarter caller* technique for placing register save and restore operations at desirable points in the code and would allocate registers using variable usage counts, register coloring, and live range splitting. These sophisticated techniques can be of modest benefit, compared with naive approaches, when applied to noninlined code. However, it is expected that particular ways in which inlined code differs from noninlined, such as the increased number of local variables found in subprograms in inlined code and the presence in inlined code of both many local variables with independent lifetimes (because the variables were drawn from code in different branches of the noninlined program's call graph) and many local variables with nested lifetimes (because the variables were derived from code in the same branch of the noninlined program's call graph), would allow the techniques to yield more substantial improvements for inlined code than simpler schemes. Also, the use of these register allocation techniques tends to be expensive, but the structure of inlined code, in which certain variable lifetime information can be determined for a block of code based on whether or not that block was derived from an inlined subprogram body, may support the development of methods for utilizing these techniques which will ameliorate the costs somewhat. Such methods may be akin to Wall's call graph analysis approach to link-time global register allocation in noninlined code [Wal86].

APPENDIX A

Data Used in Validating the Execution Time Behavior Equations

The experimental validation, described in Section 3.2, of the accuracy of the projection of inlined program execution time by the execution time behavior equations necessitated determining values for the factors in the *Call Overhead Time* terms of the noninlined and inlined instantiations of the equations. The measures for the time cost factors in the *Call Overhead Time* terms were collected for computer systems of interest. The quantities for the frequency count factors in the *Call Overhead Time* terms were gathered for test programs' noninlined and inlined versions. This appendix explains how the needed information was obtained, and presents the data acquired.

The time cost factors gauged were:

Time Cost for each Call and Return Operation Pair
Time Cost for Removing the Parameters from the Stack after Returning from each Call
Time Cost for Adjusting the Stack to Accommodate Stack-allocated Locals
Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site
Time Cost for Saving and Restoring each Register

These costs were found on the four computer systems listed in Table 2.1 by timing the requisite machine operations produced by each target C compiler.[1] Assembly language programs, composed of the appropriate instructions duplicated a large number of times in the body of a loop, were created. Each of the programs was processed by the UNIX *strip* utility, to remove from its linked object file information not essential to its execution, and was timed via the average of five runs through UNIX *ksh time*. The machine-specific sections of this appendix present the assembly language programs constructed, and discuss the time cost value computations. Table A.1 summarizes the time cost data given in those sections.

[1]Timings collected for MC68020 instructions differ by only about 11% from the average of two kinds of timings published for MC68020 instructions [Mot85]: timings that apply when the instructions are found in the cache and timings that apply when they must be fetched from main memory. Published timings are not available for VAX-8600, Clipper, and Convex instructions.

	Time Cost for each Call & Return Operation Pair	Time Cost for Removing Parameters from the Stack	Number of Parameters Passed	Time Cost for Adjusting the Stack to accommodate Stack Locals	Size in Bytes of Stack Locals	Parameter Passing Time - Time for Handling at Inlined Site	Method for Parameter Passing	Time Cost for Saving & Restoring each Register	Application of Costs to Registers
VAX-8600	3.824×10^{-6}	0.0	any	1.920×10^{-7} 2.880×10^{-7}	1-63 >63	3.25×10^{-7}	Stack	6.600×10^{-7} 8.400×10^{-7} 1.056×10^{-6} 1.096×10^{-6} 1.344×10^{-6} 1.480×10^{-6}	for 1 reg for 2 regs for 3 regs for 4 regs for 5 regs for 6 regs
MC68020	1.240×10^{-6}	1.120×10^{-7} 2.260×10^{-7}	1,2 >2	0.0	any	3.20×10^{-8}	Stack	5.200×10^{-7} 9.280×10^{-7} 2.680×10^{-7}	for 1 reg for >1 reg per reg over 1
Clipper	1.160×10^{-6}	1.600×10^{-7} 3.000×10^{-7}	3,4,5 >5	3.200×10^{-7} 6.000×10^{-7}	1-15 >15	8.87×10^{-8} 4.59×10^{-7}	Register Stack	5.80×10^{-7} 1.00×10^{-6} 2.06×10^{-6} 3.60×10^{-7} 1.60×10^{-6} 2.34×10^{-6} 3.16×10^{-6} 3.92×10^{-6}	for 1 reg for 2 regs for >2 regs per reg over 3 for 1 fl. pt. reg for 2 fl. pt. regs for 3 fl. pt. regs for 4 fl. pt. regs
Convex	3.57×10^{-6}	1.080×10^{-7} 2.040×10^{-7}	1[2] >1	1.080×10^{-7} 1.960×10^{-7}	1-7 >7	1.82×10^{-7}	Stack	6.720×10^{-7}	per reg

Table A.1: Time Costs (in Seconds) of Factors in Execution Time Behavior Equations

The quantity *Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site* was the only time cost value that was not straightforward and fairly inexpensive to measure. Each actual parameter's storage class and type determine its passing cost, and noting the class and type of every parameter passed during the execution of a program would be costly. Instead, only the count of passed parameters was accumulated, and a fixed distribution of the parameters across common classes and types was assumed. For the VAX-8600, MC68020, and Convex, 32-bit register, constant, and stack-resident data items were considered equally likely, and for the Clipper, whose C compiler performs more extensive register allocation than the other systems' C compilers, 32-bit register data items were expected in one-half the cases, 32-bit constant data items in one-third, and 32-bit stack-resident data

[2]On the Convex, every call produced by the compiler has at least one parameter, which is the number of other parameters being passed for the call.

items in one-sixth.[3] Using these assumptions, the average cost of passing a parameter was calculated. Inlining transforms each parameter passing operation at an inlined call site either into a copy of the actual parameter's value to a local temporary, introduced to replace what was the corresponding formal parameter, or into no operation at all when the actual parameter is copy-propagated. The local temporary substituted for the formal parameter may be of stack-resident or register class. To avoid the overhead of recording the class of each such temporary at every executed inlined call site, the temporary was presumed to always be of stack-resident class on the VAX-8600, MC68020, and Convex, and to be of stack-resident class one-third of the time and of register class two-thirds of the time on the Clipper. Using the assumptions about the distribution of actual parameters' classes and types and substituted local temporaries' classes, the average cost of copying an actual parameter value to a local temporary was computed. Determining the number of actual parameters that were copied to temporaries and the number that were copy-propagated at executed inlined call sites would be expensive. Investigation using a subset of the test programs listed in Table 2.2 indicated that copy propagation occurred in about one-third of the executed instances. Hence, *Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site* was estimated to be the average cost of passing a parameter minus two-thirds the average cost of copying it.

The frequency count factors gathered for noninlined and inlined versions of test programs were:

Count of Call and Return Operation Pairs Executed
Count of Parameter-Passing Call Operations Executed
Count of Calls Executed to Subprograms with Stack-allocated Locals
Count of Parameters Passed during the Program's Execution
Count of Registers Saved and Restored during the Program's Execution

These frequency counts were determined separately on the four computer systems under consideration. There are several reasons the counts were tabulated with respect to each of the systems individually. First, inlined programs' frequency counts differed across the computer systems. Inlining was constrained to ensure that variables assigned to registers in noninlined code were also assigned to registers in inlined code, and each system has a different number of available registers which are allocated according to its

[3]Note that the incidence of parameters of type double is taken to be negligible; this is true for the set of test programs used.

compiler's policies, so the call sites expanded inline for a program varied with respect to the target system. Second, both noninlined and inlined programs' frequency counts depended on the runtime library of the computer system on which the programs were built. The programs' call graphs had slight distinctions across computer systems, because the systems' runtime libraries were not constructed identically. Furthermore, small deviations in the values returned by the different systems' runtime library routines, such as random number generators and mathematical functions, sometimes affected the programs' execution behavior. Third, the structure of each computer system's time costs influenced how frequency counts were gathered on that system. Almost every time cost hinged on characteristics of the data to which the corresponding machine-level operation was to be applied, as the information in Table A.1 illustrates. In the execution time behavior equations, frequency counts are multiplied by their associated time cost factors. The frequency counts were collected in accordance with the relevant data qualities.

The frequency count factors were calculated for noninlined and inlined versions of the test programs listed in Table 2.2 using several tools. A tool similar to the UNIX *prof* utility was used to tally the *Count of Call and Return Operation Pairs Executed*. The other four frequency counts were computed as the product of these dynamic call count values and certain static information. The static number of parameter-passing calls and of parameters passed was output by INLINER, and the static number of subprograms with stack-allocated locals and of register save and restore operations was gathered by inspecting each program's assembly code. The resulting frequency counts are enumerated in the machine-specific sections of this appendix.

A.1. Measurements on the VAX-8600

This section presents the time costs and frequency counts collected for solving the execution time behavior equations for the VAX-8600 [Leo87]. The time costs were measured for the appropriate machine instructions produced by the Portable C compiler with its optimizer enabled. The assembly language used in the exposition and figures is the dialect accepted by the *Berkeley VAX/UNIX Assembler* [ReH83].

The assembly language instructions comprising the call and return operation pair are:

```
calls       nn,subr
ret
```

where *nn* is the total size in longwords[4] of the parameters passed to the subprogram. The program in Figure A.1 incorporates these instructions; it contains 5000 *calls* instructions within a loop that is traversed 50 times, so that a total of 250,000 call and return operation pairs are performed. The program's execution time averaged 0.956 seconds, which divided by 250,000 yields 3.824×10^{-6} seconds as the *Time Cost for each Call and Return Operation Pair*. Table A.2 presents the quantities gathered for *Count of Call and Return Operation Pairs Executed*, along with the ratio of the number of call and return operation pairs executed in a program's inlined version to the number executed in its noninlined version.

No instructions in addition to those comprising the call and return operation pair are required to remove parameters from the stack after returning from a call. This operation is included in the *ret* instruction, based on the value of the *nn* operand in the corresponding *calls* instruction. Hence, the *Time Cost for Removing the Parameters from the Stack after Returning from each Call* is zero, and the *Count*

```
        .text
#
        .align      1
        .globl      _f
_f:     .word       0x0
        ret
#
        .align      1
        .globl      _main
_main:  .word       0x0
        clrl        r0
loop:
        calls       $0,_f           # duplicated 5000 times
        incl        r0
        cmpl        r0,$50
        jlss        loop
        ret
```

Figure A.1: VAX-8600 Program to Measure Call and Return Operation Pair Time Cost

[4]A longword on the VAX-8600 is 4 bytes wide.

	Noninlined	Intramodule Inlined				Intermodule Inlined	
	Count	Count	Ratio	Count	Ratio	Count	Ratio
cache	5,279,792			2,498,649	0.47	1,231,676	0.23
calls	408,636			368,244	0.90	368,244	0.90
compact	85,007	30	0.00				
diff	820,237			76,555	0.09	54,293	0.07
dry	850,039	36	0.00				
extract	32,165	5,866	0.18				
grep	12,452	12,442	1.00				
inltest	861,363			424,784	0.49	317,470	0.37
invert	454,107			83,043	0.18	53,115	0.12
mincost	972,755	36,269	0.04				
nroff	610,121			293,548	0.48	339,423	0.56
sed	659,183			536,795	0.81	183,638	0.28
sort	855,068	311,119	0.36				
tsp	13,451,757	6,261,878	0.47				
vpcc	3,169,611			1,819,041	0.57	1,390,863	0.44
whetstone	230,850	77,936	0.34				
yacc	98,631			52,950	0.54	48,087	0.49
AVE			0.30		0.50		0.38

Table A.2: VAX-8600 Call and Return Operation Pairs Executed

of Parameter-Passing Call Operations Executed was not garnered.

The assembly language instruction to adjust the stack to accommodate a subprogram's stack-allocated local variables is:

```
subl2       nn,sp
```

if the stack-allocated local variables occupy between 1 and 63 bytes, and is:

```
movab       -nn(sp),sp
```

if the stack-allocated local variables occupy more than 63 bytes, where *nn* is the size in bytes of the space consumed by the stack-allocated local variables. Two versions of the program in Figure A.2, with *<local-stack-adjustment>* set in one version to *subl2 $2,sp* and in the other version to *movab -64(sp),sp*, were used to time the instructions. The short stack adjustment version averaged 0.096 seconds of execution time and the long stack adjustment version averaged 0.144 seconds. Dividing the values by 500,000 (100 repetitions of 5000 stack adjustment instructions), the *Time Cost for Adjusting the Stack to Accommodate Stack-allocated Locals* is 1.920×10^{-7} seconds, for each subprogram with between 1 and 63 bytes of stack-allocated local variables, and is 2.880×10^{-7} seconds, for each subprogram with more than 63

```
        .text
#
        .align      1
        .globl      _main
_main:  .word       0x0
        clrl        r0
loop:
        <local-stack-adjustment>    # duplicated 5000 times
        incl        r0
        cmpl        r0,$100
        jlss        loop
        ret
```

Figure A.2: VAX-8600 Program to Measure Local Var. Stack Adjustment Time Cost

	1 ≤ Size in Bytes of Stack-allocated Locals ≤ 63			Size in Bytes of Stack-allocated Locals > 63		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	599,962	503,410	0	378,534	2	1
calls	38,790	12,302	12,302	0	0	0
compact	37,206	0		37,040	1	
diff	748,583	16,003	16,003	1,615	1,615	1,615
dry	300,000	0		1	1	
extract	10,394	1,485		1,480	1,357	
grep	12,358	12,358		0	0	
inltest	244,428	147,793	149,541	2,417	2,353	3,745
invert	24,980	36,755	6,827	2	1	1
mincost	298,486	0		0	1	
nroff	25,083	98,271	279,348	241	3	241
sed	609,954	536,519	183,362	0	0	0
sort	310,701	310,697		1	1	
tsp	5,846,581	0		3	1	
vpcc	654,362	628,914	490,586	3,553	85,384	95,641
whetstone	91,303	1		0	0	
yacc	20,559	17,628	16,456	0	1	1

Table A.3: VAX-8600 Calls Executed to Subprograms with Stack-allocated Locals

bytes of stack-allocated local variables. Table A.3 contains the data found for *Count of Calls Executed to Subprograms with Stack-allocated Locals.*

As discussed in the introductory section of this appendix, parameters are assumed to be passed by the following operations:

```
pushl       reg             # pass 32-bit register data item
pushl       nn              # pass 32-bit constant data item
pushl       -mm(fp)         # pass 32-bit stack-resident data item
```

each occurring with equal likelihood. Three versions of the program in Figure A.3, with *<parameter-*

```
        .text
#
        .align      1
        .globl      _main
_main:  .word       0x0
        movl        $20000,r1
        clrl        r0
loop:
        <parameter-passing-instr>   # duplicated 5000 times
        addl2       r1,sp
        incl        r0
        cmpl        r0,$100
        jlss        loop
        ret
```

Figure A.3: VAX-8600 Program to Measure Parameter Passing Time Cost

passing-instr> set in each respectively to *pushl r11*, *pushl $10*, and *pushl -4(fp)*, were created for timing these parameter passing instructions. The instruction times in seconds were 5.88×10^{-7}, 6.00×10^{-7}, and 7.00×10^{-7}, so the average time to pass a parameter is 6.29×10^{-7} seconds. These parameter passing operations are replaced at inlined call sites, when actual parameter values are not copy-propagated, with the appropriate copy operations from the list below:

```
movl        reg,-mm(fp)      # copy 32-bit register data item
movl        nn,-mm(fp)       # copy 32-bit constant data item
movl        -pp(fp),-mm(fp)  # copy 32-bit stack-resident data item
```

assuming the local temporary replacing the formal parameter is stack-resident. Another three versions of the program in Figure A.3 found the times in seconds of the copy operations to be 3.68×10^{-7}, 4.52×10^{-7}, and 5.48×10^{-7}, so the average time to copy a parameter is 4.56×10^{-7} seconds. Computing *Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site* as explained in the introductory section of this appendix, i.e., as the average cost of passing a parameter minus two-thirds the average cost of copying it, the result is 3.25×10^{-7} seconds. Table A.4 contains the values gathered for *Count of Parameters Passed during the Program's Execution*.

Register save and restore operations are performed by the *calls* and *ret* instructions respectively, on the registers specified in a word-width mask located at the beginning of a callee. Of the machine's sixteen registers, six (r11 down through r6) are, by convention, saved and restored if used by a callee. These registers are always allocated by the compiler in descending numerical order, so if a subprogram

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	5,953,446	1,991,450	0
calls	769,406	729,755	729,755
compact	127,907	2	
diff	2,302,842	73,962	29,448
dry	1,550,004	0	
extract	66,155	7,298	
grep	24,726	24,716	
inltest	942,685	338,388	235,424
invert	664,237	97,064	13,656
mincost	2,239,078	2	
nroff	401,983	153,920	189,535
sed	1,195,430	1,024,088	317,775
sort	1,140,854	596,900	
tsp	21,899,871	2	
vpcc	4,121,497	2,279,036	1,615,007
whetstone	271,170	0	
yacc	155,711	72,518	64,033

Table A.4: VAX-8600 Counts of Parameters Passed during the Programs' Execution

saves and restores any registers, its register mask will be one of only six values. To measure the time consumed by each of the six corresponding register save and restore operations, six variations of the program given in Figure A.1 were utilized. The first variation set the register mask to specify r11; the second to specify r11 and r10; the third to specify r11, r10, and r9; and so on. The overhead represented by the *Time Cost for each Call and Return Operation Pair* was eliminated from the results, and the values found for *Time Cost for Saving and Restoring each Register* are given in Table A.5. The *Count of Registers Saved and Restored during the Program's Execution* is given in Tables A.6 through A.8.

Registers Specified in Mask	Time Cost
r11	6.600×10^{-7}
r11,r10	8.400×10^{-7}
r11,r10,r9	1.056×10^{-6}
r11,r10,r9,r8	1.096×10^{-6}
r11,r10,r9,r8,r7	1.344×10^{-6}
r11,r10,r9,r8,r7,r6	1.480×10^{-6}

Table A.5: VAX-8600 Time Costs (in Secs.) for Saving & Restoring Sets of Registers

	One Register Saved and Restored			Two Registers Saved and Restored		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	1,135,599	186,548	0	17	1	0
calls	39,211	0	0	353,826	351,049	351,049
compact	37,039	0		0	0	
diff	32,005	22,261	0	10	0	0
dry	400,000	0		100,001	0	
extract	3,822	0		7,014	0	
grep	0	0		1	0	
inltest	117,238	14,804	9,791	315,097	32,380	19,862
invert	3,237	11,776	0	353,786	0	0
mincost	0	0		0	0	
nroff	137,369	15,786	2,817	169,620	1,789	906
sed	48,952	0	0	73,434	0	0
sort	543,914	0		14,265	14,256	
tsp	0	0		0	0	
vpcc	319,824	49,841	19,841	488,563	332,301	190,148
whetstone	0	0		0	0	
yacc	15,842	4	0	31,568	246	244

Table A.6: VAX-8600 One & Two Register Save & Restore Operations Executed

	Three Registers Saved and Restored			Four Registers Saved and Restored		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	378,533	0	0	0	0	0
calls	10,467	279	279	1,861	12,063	12,063
compact	0	0		37,206	1	
diff	745,092	16,002	16,002	5,105	5,107	5,107
dry	50,002	0		0	0	
extract	5,862	1,646		0	0	
grep	12,359	1		0	0	
inltest	11,453	2,575	2,860	2,370	117,462	6,846
invert	14,695	0	0	11,118	18,603	6,827
mincost	1	0		0	0	
nroff	192,471	60,490	22,026	0	29,052	50,450
sed	487,572	3	3	0	487,567	85,461
sort	296,437	24,482		0	0	
tsp	46	0		0	1	
vpcc	269,019	77,541	15,241	292,836	59,639	52,717
whetstone	0	0		0	0	
yacc	14,209	4,373	3,923	1	15,121	11,146

Table A.7: VAX-8600 Three & Four Register Save & Restore Operations Executed

	Five Registers Saved and Restored			Six Registers Saved and Restored		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	0	1	0	0	1	1
calls	227	0	0	0	2,089	2,089
compact	0	0		0	0	
diff	0	0	0	0	2	2
dry	0	0		0	1	
extract	0	0		0	1,196	
grep	0	12,357		0	0	
inltest	0	7,506	123,871	0	18,124	19,698
invert	0	1	1	0	0	0
mincost	0	1		0	0	
nroff	0	53,023	153,675	0	118,703	106,611
sed	0	0	0	0	48,952	97,901
sort	0	7		0	271,953	
tsp	0	0		0	0	
vpcc	13,428	124,054	22,319	105,220	414,450	532,876
whetstone	0	0		0	0	
yacc	23,153	22,811	22,500	199	495	804

Table A.8: VAX-8600 Five & Six Register Save & Restore Operations Executed

A.2. Measurements on the MC68020

This section presents the time costs and frequency counts collected for solving the execution time behavior equations for the MC68020 [Mot85]. The time costs were measured for the appropriate machine instructions produced by the Sun-3 C compiler with its optimizer enabled. The assembly language used in the exposition and figures is the dialect accepted by the *Sun Microsystems Assembler* [McT83]. The shared libraries available on the SunOS were not utilized.

The assembly language instructions comprising the call and return operation pair[5] are:

```
jbsr        subr
link        a6,nn
unlk        a6
rts
```

where *nn* is the total size in bytes of the subprogram's stack-allocated local variables. The program in Figure A.4 incorporates these instructions; it contains 5000 *jbsr* instructions within a loop that is

[5]The assembler maps the *jbsr* instruction into either *jsr addr:l* or *bsrX offset*, which have the same timings.

```
        .text
#
        .globl      _f
_f:
        link        a6,#0
        unlk        a6
        rts
#
        .globl      _main
_main:
        link        a6,#0
        movw        #49,d0
_loop:
        jbsr        _f              # duplicated 5000 times
        dbra        d0,_loop
        unlk        a6
        rts
```

Figure A.4: MC68020 Program to Measure Call and Return Operation Pair Time Cost

traversed 50 times, so that a total of 250,000 call and return operation pairs are performed. The program's execution time averaged 0.310 seconds, which divided by 250,000 yields 1.240×10^{-6} seconds as the *Time Cost for each Call and Return Operation Pair*. Table A.9 presents the quantities gathered for

	Noninlined	Intramodule Inlined				Intermodule Inlined	
	Count	Count	Ratio	Count	Ratio	Count	Ratio
cache	4,806,579			2,025,436	0.42	758,463	0.16
calls	424,343			129,820	0.31	129,820	0.31
compact	84,912	19	0.00				
diff	820,125			56,951	0.07	34,688	0.04
dry	850,021	50,018	0.06				
extract	32,080	6,313	0.20				
grep	12,441	12,431	1.00				
inltest	853,257			416,447	0.49	312,799	0.37
invert	491,777			88,288	0.18	58,401	0.12
mincost	971,910	35,424	0.04				
nroff	610,102			361,256	0.59	307,572	0.50
sed	659,151			610,193	0.93	208,086	0.32
sort	854,950	286,519	0.34				
tsp	43,280,451	36,090,572	0.83				
vpcc	3,132,192			1,693,196	0.54	1,295,896	0.41
whetstone	943,278	880,264	0.93				
yacc	98,606			46,429	0.47	45,991	0.47
AVE			0.43		0.44		0.30

Table A.9: MC68020 Call and Return Operation Pairs Executed

Count of Call and Return Operation Pairs Executed, along with the ratio of the number of call and return operation pairs executed in a program's inlined version to the number executed in its noninlined version.[6]

The assembly language instruction to remove parameters from the stack after returning from each call is:

```
addqw       #nn,sp
```

if 1 or 2 parameters are passed (assuming the parameters are 4 bytes each), and is:

```
lea         sp@(nn),sp
```

if more than 2 parameters are passed, where *nn* is the size in bytes of the parameters passed. Two versions of the program in Figure A.5, with *<parm-stack-adjustment>* set in one version to *addqw #8,sp* and in the other version to *lea sp@(0xc),sp*, were used to time the instructions. The short stack adjustment version averaged 0.056 seconds of execution time and the long stack adjustment version averaged 0.113 seconds. Dividing the values by 500,000 (100 repetitions of 5000 stack adjustment instructions), the *Time Cost for Removing the Parameters from the Stack after Returning from each Call* is 1.120×10^{-7} seconds, for each subprogram call with 1 or 2 parameters, and is 2.260×10^{-7} seconds, for each subprogram call with more than 2 parameters. Table A.10 contains the data found for *Count of Parameters Passed during the Program's Execution.*

```
        .text
#
        .globl      _main
_main:
        link        a6,#0
        movw        #99,d0
_loop:
        <parm-stack-adjustment>     # duplicated 5000 times
        dbra        d0,_loop
        unlk        a6
        rts
```

Figure A.5: MC68020 Program to Measure Parameter Stack Adjustment Time Cost

[6]The call counts given in this table differ slightly from those given in Table 2.3, because of an upgrade to the system's runtime library.

	1 ≤ Number of Parameters Passed ≤ 2			Number of Parameters Passed > 2		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	3,725,106	1,424,422	378,532	701,152	221,076	0
calls	419,557	125,962	125,962	1,169	1,169	1,169
compact	74,162	1		10,732	0	
diff	50,098	22,264	3	736,952	1,615	1,615
dry	550,002	50,000		200,000	0	
extract	19,674	1,889		9,181	1,485	
grep	12,368	12,358		0	0	
inltest	621,638	241,583	156,845	33,599	19,660	18,055
invert	422,063	18,576	12,241	23,552	23,552	0
mincost	658,136	1		278,351	0	
nroff	339,503	152,869	132,371	1,031	0	0
sed	658,905	609,952	207,845	3	0	0
sort	854,635	286,209		0	0	
tsp	4,105,602	46		3,084,323	0	
vpcc	1,768,222	758,910	447,060	410,914	270,243	210,133
whetstone	1,400	0		10	0	
yacc	80,521	32,368	32,367	3,637	433	0

Table A.10: MC68020 Counts of Parameter-Passing Call Operations Executed

No instructions in addition to those comprising the call and return operation pair are required to adjust the stack to accommodate a subprogram's stack-allocated local variables. This operation is included in the *link* instruction, based on the value of its *nn* operand. Hence, the *Time Cost for Adjusting the Stack to Accommodate Stack-allocated Locals* is zero, and the *Count of Calls Executed to Subprograms with Stack-allocated Locals* was not garnered.

As discussed in the introductory section of this appendix, parameters are assumed to be passed by the operations in the list below:

```
movl    reg,sp@-            # pass 32-bit register data item
pea     nn                  # pass 32-bit constant data item
movl    a6@(mm),sp@-        # pass 32-bit stack-resident data item
```

each occurring with equal likelihood. Three versions of the program in Figure A.6, with *<parameter-passing-instr>* set in each respectively to *movl d7,sp@-*, *pea 0x5*, and *movl a6@(-4),sp@-*, were created for timing these parameter passing instructions. The instruction times in seconds were 2.24×10^{-7}, 3.20×10^{-7}, and 3.68×10^{-7}, so the average time to pass a parameter is 3.04×10^{-7} seconds. These parameter passing operations are replaced at inlined call sites, when actual parameter values are not copy-

```
        .text
#
        .globl      _main
_main:
        link        a6,#-8
        movl        #10000,d1
        movw        #99,d0
_loop:
        <parameter-passing-instr>   # duplicated 2500 times
        addl        d1,sp
        dbra        d0,_loop
        unlk        a6
        rts
```

Figure A.6: MC68020 Program to Measure Parameter Passing Time Cost

propagated, with the appropriate copy operations from the following list:[7]

```
movl    reg,a6@(mm)         # copy 32-bit register data item
moveq   nn,d1               # copy 32-bit constant data item
movl    d1,a6@(mm)          # copy 32-bit constant data item, con't
movl    a6@(pp),a6@(mm)     # copy 32-bit stack-resident data item
```

assuming the local temporary replacing the formal parameter is stack-resident. Another three versions of the program in Figure A.6 found the times in seconds of the copy operations to be 3.20×10^{-7}, 4.40×10^{-7}, and 4.64×10^{-7}, so the average time to copy a parameter is 4.08×10^{-7} seconds. Computing *Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site* as specified in the introductory section of this appendix, i.e., as the average cost of passing a parameter minus two-thirds the average cost of copying it, the result is 3.20×10^{-8} seconds. Table A.11 contains the values gathered for *Count of Parameters Passed during the Program's Execution*.

Register save and restore operations are performed using the instructions:

```
movl    reg,sp@         # save single register
movl    a6@(-n),reg     # restore single register
```

[7]For the *moveq* instruction to be applicable, the value of the constant must be between -128 and 127 inclusive. This range is presupposed to be the appropriate one in most cases.

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	6,331,978	2,369,982	378,532
calls	801,202	253,382	253,382
compact	127,823	2	
diff	2,302,842	50,980	6,466
dry	1,550,004	100,000	
extract	66,155	8,079	
grep	24,726	24,716	
inltest	942,154	338,939	247,585
invert	718,143	102,395	19,069
mincost	2,239,078	2	
nroff	401,983	153,919	163,440
sed	1,195,430	1,097,518	293,305
sort	1,140,854	572,418	
tsp	21,899,916	47	
vpcc	4,121,730	2,097,863	1,443,623
whetstone	1,470	0	
yacc	155,711	59,528	58,227

Table A.11: MC68020 Counts of Parameters Passed during the Programs' Execution

when a single register is saved and restored, and using the instructions:

```
moveml  mask,sp@        # save multiple registers specified in mask
moveml  a6@(-n),mask    # restore multiple registers specified in mask
```

when multiple registers are saved and restored. The time cost for saving and restoring a single register was determined utilizing a version of the program given in Figure A.7, with *<link-instruction>* set to *link a6,#-4* and *<register-save-&-restore>* set to the pair *movl d7,sp@* and *movl a6(-4),d7*. Dividing the

```
        .text
#
        .globl      _main
_main:
        <link-instruction>
        movw        #99,d0
_loop:
        <register-save-&-restore>   # duplicated 2500 times
        dbra        d0,_loop
        unlk        a6
        rts
```

Figure A.7: MC68020 Program to Measure Reg. Save & Restore Operation Time Cost

version's average execution time (0.130 seconds) by 250,000, the *Time Cost for Saving and Restoring each Register* is found to be 5.200×10^{-7} seconds when a single register is saved and restored. The time cost for saving and restoring multiple registers can be partitioned into two components: the constant overhead associated with the *moveml* instructions and the varying expense that depends on the number of registers specified in the *mask* field. To determine the time costs of these components, two versions of the program in Figure A.7 were employed. One version saved and restored the two registers a5 and d7 by setting *<link-instruction>* to *link a6,#-8* and *<register-save-&-restore>* to the pair *moveml #8320,sp@* and *moveml a6(-8),#8320.*[8] The other version saved and restored the four registers a5, a4, d7, and d6 by setting *<link-instruction>* to *link a6,#-16* and *<register-save-&-restore>* to the pair *moveml #12480,sp@* and *moveml a6(-16),#12480.*[9] By subtracting the time consumed by each pair of multiple register save and restore operations in the first version from the time consumed by each pair of multiple register save and restore operations in the second version, and halving the result, the time cost for saving and restoring an individual register specified in the *mask* operand was found. Next, subtracting twice the time cost for saving and restoring an individual register in *mask* from the first version's time cost for each multiple register save and restore operation yielded the time cost for the constant overhead portion of the multiple register save and restore operation. Performing these measurements and computations, the *Time Cost for Saving and Restoring each Register* when multiple registers are saved and restored emerges as the sum of the following two values (expressed in seconds): the constant value 9.280×10^{-7} and the value 2.680×10^{-7} multiplied by the number of registers being saved and restored. The *Count of Registers Saved and Restored during the Program's Execution* is given in Tables A.12 and A.13.

[8]The decimal number 8320, which is equivalent to the hex number 2080, is the mask value that specifies a5 and d7.

[9]The decimal number 12480, which is equivalent to the hex number 30C0, is the mask value that specifies a5, a4, d7, and d6.

	Single Register Saves and Restores			Multiple Register Saves and Restores		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	1,135,599	186,548	0	757,082	378,535	378,533
calls	39,280	0	0	382,363	127,131	127,131
compact	0	0		74,162	1	
diff	32,005	22,261	0	750,207	1,619	1,618
dry	350,000	0		350,003	50,001	
extract	0	0		16,698	3,374	
grep	0	0		12,360	12,358	
inltest	117,244	14,736	4,352	328,918	177,586	181,711
invert	0	0	0	420,674	35,793	12,241
mincost	0	0		1	1	
nroff	137,128	15,786	2,817	362,331	330,784	301,836
sed	48,952	0	0	561,006	609,952	207,845
sort	543,914	0		310,702	286,216	
tsp	0	0		1,345,731	46	
vpcc	319,824	57,995	3,973	1,169,249	927,353	772,581
whetstone	0	0		0	0	
yacc	15,842	4	0	69,130	36,550	36,546

Table A.12: MC68020 Single & Multiple Reg Save & Restore Operations Executed

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	1,892,697	757,077	757,070
calls	779,871	319,230	319,230
compact	296,729	8	
diff	2,258,943	12,946	12,945
dry	750,009	300,008	
extract	39,258	18,474	
grep	37,080	61,789	
inltest	674,043	735,431	885,843
invert	878,111	120,572	38,140
mincost	3	6	
nroff	891,362	1,953,751	2,072,518
sed	1,609,586	2,488,751	978,223
sort	917,841	1,660,267	
tsp	2,692,703	97	
vpcc	3,893,195	4,767,413	4,798,013
whetstone	0	0	
yacc	231,997	195,693	195,985

Table A.13: MC68020 Registers Specified in Multiple Reg. Save & Restore Operations

A.3. Measurements on the Clipper

This section presents the time costs and frequency counts collected for solving the execution time behavior equations for the Clipper [Fai87]. The time costs were measured for the appropriate machine instructions produced by the Green Hills C compiler with its optimizer enabled. The assembly language used in the exposition and figures is the dialect accepted by the *Clipper Assembler* [Cli89].

The assembly language instructions comprising the call and return operation pair are:

```
call        sp,subr
ret         sp
```

The program in Figure A.8 incorporates these instructions; it contains 5000 *call* instructions within a loop that is traversed 50 times, so that a total of 250,000 call and return operation pairs are performed. The program's execution time averaged 0.290 seconds, which divided by 250,000 yields 1.160×10^{-6} seconds as the *Time Cost for each Call and Return Operation Pair*. Table A.14 presents the quantities gathered for *Count of Call and Return Operation Pairs Executed*, along with the ratio of the number of call and return operation pairs executed in a program's inlined version to the number executed in its noninlined version.

```
        .text
#
        .align      2
        .globl      _f
_f:     ret         sp
#
        .align      2
        .globl      _main
_main:  loadq       $0,r0
L7:
        call        sp,_f           # duplicated 5000 times
        addq        $1,r0
        cmpi        $50,r0
        bcgt        L7
        ret         sp
```

Figure A.8: Clipper Program to Measure Call and Return Operation Pair Time Cost

	Noninlined	Intramodule Inlined				Intermodule Inlined	
	Count	Count	Ratio	Count	Ratio	Count	Ratio
cache	4,426,614			1,645,504	0.37	440,036	0.10
calls	420,515			13,055	0.03	13,045	0.03
compact	84,917	37,125	0.44				
diff	787,323			3,761	0.01	739,098	0.94
dry	850,006	50,003	0.06				
extract	30,338	17,541	0.58				
grep	12,377	12,367	1.00				
inltest	808,065			263,483	0.33	259,189	0.32
invert	86,224			61,532	0.71	54,034	0.63
mincost	989,689	228,665	0.23				
nroff	609,592			279,264	0.46	257,726	0.42
sed	658,911			585,473	0.89	634,422	0.96
sort	854,649	854,622	1.00				
tsp	7,280,492	4,130,230	0.57				
vpcc	2,710,919			1,387,894	0.51	1,034,365	0.38
whetstone	152,916	2	0.00				
yacc	89,409			41,130	0.46	42,985	0.48
AVE			0.49		0.42		0.47

Table A.14: Clipper Call and Return Operation Pairs Executed

The Green Hills C compiler produces code to pass a subprogram's first two parameters in registers whenever possible, and parameters passed in registers need not be removed from the stack after returning from a call. For call sites at which 3, 4, or 5 parameters (assuming the parameters are 4 bytes each) are passed, the assembly language instruction to remove parameters from the stack after returning from a call is:

```
addq        $nn,sp
```

and for call sites at which more than 5 parameters are passed, the instruction is:

```
addi        $nn,sp
```

where *nn* is the size in bytes of the parameters passed on the stack. Two versions of the program in Figure A.9, with *<parm-stack-adjustment>* set in one version to *addq $4,sp* and in the other version to *addi $16,sp*, were used to time the instructions. The short stack adjustment version averaged 0.080 seconds of execution time and the long stack adjustment version averaged 0.150 seconds. Dividing the values by 500,000 (100 repetitions of 5000 stack adjustment instructions), the *Time Cost for Removing*

```
        .text
#
        .align      2
        .globl      _main
_main:  loadq       $0,r0
        movw        sp,r1
L7:
        <parm-stack-adjustment>     # duplicated 5000 times
        addq        $1,r0
        cmpi        $100,r0
        bcgt        L7
        movw        r1,sp
        ret         sp
```

Figure A.9: Clipper Program to Measure Parameter Stack Adjustment Time Cost

the Parameters from the Stack after Returning from each Call is 1.600×10^{-7} seconds, for each subprogram call with 3, 4, or 5 parameters, and is 3.000×10^{-7} seconds, for each subprogram call with more than 5 parameters. Table A.15 contains the data collected for *Count of Parameter-Passing Call Operations Executed*.

	3 ≤ Number of Parameters Passed ≤ 5			Number of Parameters Passed > 5		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	701,141	221,065	61,493	0	0	0
calls	1,167	1,167	1,167	0	0	0
compact	10,668	0		0	0	
diff	733,727	0	732,110	3,226	0	3,226
dry	200,000	0		0	0	
extract	9,181	8,930		0	0	
grep	0	0		0	0	
inltest	33,565	22,513	21,637	11	11	11
invert	23,552	23,552	23,552	0	0	0
mincost	176,963	55,729		117,205	117,205	
nroff	1,031	0	0	0	0	0
sed	3	0	0	0	0	0
sort	0	0		0	0	
tsp	1,757,862	1,361,262		1,362,461	1,362,461	
vpcc	282,373	269,318	253,274	128,541	93,468	81,818
whetstone	89,900	0		10	0	
yacc	3,637	4	2,251	0	0	0

Table A.15: Clipper Counts of Parameter-Passing Call Operations Executed

The assembly language code to adjust the stack to accommodate a subprogram's stack-allocated local variables is the pair:

```
subq        $nn,sp    # to reserve space for the locals
addq        $nn,sp    # to reclaim the locals' space
```

if the stack-allocated local variables occupy between 1 and 15 bytes, and is the pair:

```
subi        $nn,sp    # to reserve space for the locals
addi        $nn,sp    # to reclaim the locals' space
```

if the stack-allocated local variables occupy more than 15 bytes, where *nn* is the size in bytes of the space consumed by the stack-allocated local variables. The time costs for the *addq* and *addi* instructions were determined when measuring the *Time Cost for Removing the Parameters from the Stack after Returning from each Call*, using versions of the program in Figure A.9. Similar versions of that program were employed to time the *subq* and *subi* instructions, which were found to consume the same amount of time as *addq* and *addi*, respectively. Hence, the *Time Cost for Adjusting the Stack to Accommodate Stack-allocated Locals* is 3.200×10^{-7} seconds, for each subprogram with between 1 and 15 bytes of stack-allocated local variables, and is 6.000×10^{-7} seconds, for each subprogram with more than 15 bytes of stack-allocated local variables. Table A.16 contains the information garnered for *Count of Calls Executed to Subprograms with Stack-allocated Locals.*

As previously noted, the Green Hills C compiler produces code to pass a subprogram's first two parameters in registers whenever possible, and to pass its other parameters on the stack. The average cost of passing a parameter was figured separately for these transmission methods. As discussed in the introductory section of this appendix, parameters are considered to be passed in registers by the following operations:

```
movw        reg1,reg  # pass 32-bit register data item in a register
loadq/loadi $nn,reg   # pass 4-/32-bit constant data item in a register
loadw       m(sp),reg # pass 32-bit stack-resident data item in a register
```

with the first instruction occurring one-half the time, the second one-third, and the third one-sixth. Three versions of the program in Figure A.10, with *<parameter-passing-instr>* set in each respectively to two copies of *loadw 4(sp),r1*, the pair *loadq $1,r1* and *loadi $32,r1*, and two copies of *movw r2,r1*, were

	1 ≤ Size in Bytes of Stack-allocated Locals ≤ 15			Size in Bytes of Stack-allocated Locals > 15		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	2,272,372	412,964	378,541	378,535	2	61,494
calls	49,478	12,159	12,159	1,389	895	885
compact	0	0		74,248	37,124	
diff	34,353	3,758	3,759	1,621	2	3
dry	300,001	50,001		50,004	1	
extract	19,873	14,552		3,244	2,145	
grep	17	12,365		12,358	1	
initest	370,471	113,254	107,351	16,441	130,066	131,700
invert	64,306	61,067	54,029	3,416	464	4
mincost	157,099	28,209		56,761	28,553	
nroff	317,784	80,564	29,090	11,630	122,206	224,569
sed	48,959	1	0	536,520	585,471	634,421
sort	24,486	24,483		271,956	271,956	
tsp	2,800,173	45,255		1,213	1,213	
vpcc	1,584,996	411,472	365,777	352,617	401,659	385,843
whetstone	89,913	0		2	1	
yacc	15,014	1,939	1,624	7,688	18,223	18,143

Table A.16: Clipper Calls Executed to Subprograms with Stack-allocated Locals

```
        .text
#
        .align      2
        .globl      _main
_main:  loadq       $0,r0
L7:
        <parameter-passing-instr>   # duplicated 5000 times
        addq        $1,r0
        cmpi        $100,r0
        bcgt        L7
        ret         sp
```

Figure A.10: Clipper Program to Measure Parameter Passing Time Cost

created for timing these parameter passing instructions. The instruction times in seconds were 3.00×10^{-7}, 3.90×10^{-7}, and 6.30×10^{-7}, so the average time to pass a parameter in a register is 3.85×10^{-7} seconds, given the assumed distribution of actual parameters' classes and types. Parameters are considered to be passed on the stack by the following operations:

```
pushw       reg,sp     # pass 32-bit register data item on the stack
loadq/loadi $nn,reg    # pass 4-/32-bit constant data item on the stack
pushw       reg,sp     # pass 4-/32-bit constant data item, con't
loadw       m(sp),reg  # pass 32-bit stack-resident data item on the stack
pushw       reg,sp     # pass 32-bit stack-resident data item, con't
```

with the first operation occurring one-half the time, the second one-third, and the third one-sixth. The operation times in seconds were 6.00×10^{-7}, 7.50×10^{-7}, and 1.23×10^{-6}, so the average time to pass a parameter on the stack is 7.55×10^{-7} seconds, given the assumed distribution of actual parameters' classes and types. These register and stack parameter passing operations are replaced at inlined call sites, when actual parameter values are not copy-propagated, with appropriate copy operations to local temporaries introduced to substitute for the corresponding formal parameters. When these local temporaries are of register class, the appropriate copy operations are the same as those listed for passing parameters in registers, so the average time to copy a parameter to a register is 3.85×10^{-7} seconds. When these local temporaries are of stack-resident class, the appropriate copy operations are those in the list below.

```
storw       reg,m(sp)  # copy 32-bit register data item to the stack
loadq/loadi $nn,reg    # copy 4-/32-bit constant data item to the stack
storw       reg,m(sp)  # copy 4-/32-bit constant data item, con't
loadw       p(sp),reg  # copy 32-bit stack-resident data item to the stack
storw       reg,m(sp)  # copy 32-bit stack-resident data item, con't
```

The times in seconds of these copy operations were 3.20×10^{-7}, 7.30×10^{-7}, and 9.50×10^{-7}, so the average time to copy a parameter to the stack is 5.61×10^{-7} seconds, given the assumed distribution of actual parameters. Presuming, as mentioned in the introduction to this appendix, that substituted temporaries are of register class two-thirds of the time and of stack-resident class one-third of the time, the average time to copy a parameter is 4.44×10^{-7} seconds. Computing *Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site* as explained in the introductory section of this appendix, i.e., as the average cost of passing a parameter minus two-thirds the average cost of copying it, the result is 8.87×10^{-8} seconds for parameters passed in registers and 4.59×10^{-7} seconds for parameters passed on the stack. Table A.17 contains the values gathered for *Count of Parameters Passed during the Program's Execution.*

	Number Passed in Registers			Number Passed on the Stack		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	5,630,779	2,148,892	501,526	701,141	221,065	61,493
calls	800,104	25,143	25,133	1,167	1,167	1,167
compact	95,585	37,125		32,004	0	
diff	1,554,863	7,250	1,477,924	748,249	0	745,015
dry	1,300,005	100,001		250,000	0	
extract	56,434	31,990		10,917	10,415	
grep	24,734	24,724		0	0	
initest	978,409	298,278	286,983	45,367	29,214	29,357
invert	155,618	113,062	98,066	23,552	23,552	23,552
mincost	1,457,562	457,327		908,747	666,279	
nroff	400,956	138,202	112,096	1,031	0	0
sed	1,195,427	1,073,037	1,170,936	3	0	0
sort	1,140,855	1,140,839		0	0	
tsp	14,560,937	8,260,413		7,610,218	6,818,370	
vpcc	2,975,584	1,756,793	1,323,829	1,088,994	930,858	849,926
whetstone	181,220	0		89,950	0	
yacc	152,320	72,047	75,700	3,637	4	2,251

Table A.17: Clipper Counts of Parameters Passed during the Program's Execution

Register save and restore operations are performed using the instructions:

```
savewN                  # save general registers
restwN                  # restore general registers
```

for general registers, where *N* is a value between 0 and 14 which specifies that the general registers numbered *N* through 14 should be saved or restored, and using the instructions:

```
savedM                  # save floating point registers
restdM                  # restore floating point registers
```

for floating point registers, where *M* is a value between 0 and 7 which specifies that the floating point registers numbered *M* through 7 should be saved or restored. To make efficient use of these instructions, the Green Hills C compiler allocates registers in descending order of their numerical designation. Considering first the cost of saving and restoring general registers, it is observed that the instructions *savewN* and *restwN* are implemented in hardware only for values of *N* between 0 and 12 inclusive; when *N* is 13 or 14, the assembler maps the instructions to pushes and pops of the appropriate registers.[10] Hence, with

[10]The assembler maps *savew14* and *restw14* to *pushw r14,sp* and *popw r14,sp*, respectively, and maps *savew13* and *restw13* to the pair *pushw r14,sp; pushw r13,sp* and the pair *popw r13,sp; popw r14,sp*, respectively.

respect to general registers, the values collected for *Time Cost for Saving and Restoring each Register*, were measured for several cases. When a single register (always r14) is saved and restored, the time cost was found to be 5.80×10^{-7} seconds. When two registers (r14 and r13) are saved and restored, the time cost was found to be 1.00×10^{-6} seconds. The *Count of Registers Saved and Restored during the Program's Execution* values associated with these two time costs are given in Table A.18. For three registers, the time cost was 2.06×10^{-6} seconds, and for each register saved and restored in addition to three, the incremental cost was 3.60×10^{-7} seconds per register. Tables A.19 and A.20 contain the corresponding *Count of Registers Saved and Restored during the Program's Execution*. Considering next the cost of saving and restoring floating point registers, it is noted that the Green Hills C compiler treats floating point registers f0 through f3 as scratch registers, never saving and restoring their values. The quantities found for *Time Cost for Saving and Restoring each Register* with respect to the other floating point registers are given in Table A.21. Table A.22 contains the floating point *Count of Registers Saved and Restored during the Program's Execution.*

	One Register Saved and Restored			Two Registers Saved and Restored		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	1,555,040	577,010	0	158,260	34,419	0
calls	1,694	0	0	1	1	1
compact	85	0		0	0	
diff	22,253	0	0	18,074	3,492	3,492
dry	100,002	0		50,000	0	
extract	1,930	751		6,453	3,222	
grep	0	0		0	0	
inltest	201,551	12,493	10,446	33,486	24,161	14,308
invert	2,954	0	0	13,654	13,654	13,654
mincost	0	27,864		157,098	28,208	
nroff	145,359	1,039	1	97,026	39,271	27,152
sed	0	0	0	48,951	0	0
sort	8	0		1	0	
tsp	1,361,251	1,361,251		2,798,872	44,000	
vpcc	256,645	106,710	2	415,420	124,654	723
whetstone	1,400	0		0	0	
yacc	3,307	0	2,251	434	0	0

Table A.18: Clipper One & Two Register Save & Restore Operations Executed

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	791,178	468,981	61,494
calls	49,284	12,098	12,088
compact	74,163	37,124	
diff	746,714	2	735,339
dry	150,002	50,001	
extract	9,102	9,183	
grep	12,360	12,358	
inltest	127,347	142,980	152,954
invert	37,883	37,882	30,384
mincost	172,936	116,864	
nroff	30,084	140,848	227,586
sed	609,959	585,472	634,421
sort	310,703	310,702	
tsp	1,361,299	1,362,463	
vpcc	1,014,786	999,038	930,201
whetstone	3	1	
yacc	53,984	40,882	40,486

Table A.19: Clipper Three or More Register Save & Restore Operations Executed

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	761,682	191,582	368,964
calls	97,849	58,737	58,737
compact	444,978	222,744	
diff	1,485,511	12	1,464,643
dry	200,012	150,009	
extract	16,645	17,281	
grep	24,727	61,791	
inltest	73,492	79,824	260,239
invert	103,109	123,585	114,708
mincost	171,321	231,766	
nroff	92,332	446,526	870,873
sed	1,268,872	2,976,310	3,221,061
sort	1,741,803	1,741,802	
tsp	64	18	
vpcc	2,789,368	2,463,250	2,930,940
whetstone	7	3	
yacc	140,811	155,778	153,719

Table A.20: Clipper No. Registers in Excess of 3 Saved & Restored in Operation Pair

Set of Registers	Time Cost
f7	1.60×10^{-6}
f7,f6	2.34×10^{-6}
f7,f6,f5	3.16×10^{-6}
f7,f6,f5,f4	3.92×10^{-6}

Table A.21: Clipper Time Costs (in Secs.) for Saving & Restoring Floating Pt. Regs

	One Register			Two Registers			Three Registers			Four Registers		
	NI	Intra Inl	Inter Inl	NI	Intra Inl	Inter Inl	NI	Intra Inl	Inter Inl	NI	Intra Inl	Inter Inl
cache	1	1	1	0	0	0	0	0	0	0	0	0
calls	0	0	0	0	0	0	0	0	0	0	0	0
compact	0	0		0	0		0	0		0	0	
diff	0	0	0	0	0	0	0	0	0	0	0	0
dry	0	0		0	0		0	0		0	0	
extract	0	0		0	0		0	0		0	0	
grep	0	0		0	0		0	0		0	0	
inltest	0	0	0	0	0	0	0	0	0	0	0	0
invert	0	0	0	0	0	0	0	0	0	0	0	0
mincost	0	0		0	0		0	0		0	0	
nroff	0	0	0	0	0	0	0	0	0	0	0	0
sed	0	0	0	0	0	0	0	0	0	0	0	0
sort	0	0		0	0		0	0		0	0	
tsp	1	1		1,362,461	1,361,250		1,211	1,211		1,210	1,211	
vpcc	0	0	22	0	0	0	0	0	0	0	0	0
whetstone	1,401	0		89,901	0		1	1		0	0	
yacc	0	0	0	0	0	0	0	0	0	0	0	0

Table A.22: Clipper Floating Point Register Save & Restore Operations Executed

A.4. Measurements on the Convex

This section presents the time costs and frequency counts collected for solving the execution time behavior equations for the Convex [CON85]. The time costs were measured for the appropriate machine instructions produced by the CONVEX C compiler with its optimizer disabled.[11] The assembly language used in the exposition and figures is the dialect accepted by the *CONVEX Assembler*.

The assembly language instructions comprising the call and return operation pair are:

```
mov       sp,ap          # set argument pointer
pshea     #0             # push count of parameters being passed
calls     subr           # branch to subprogram
rtn                      # return from subprogram
add.w     #4,sp          # adjust stack to remove count & parameters
ld.w      12(fp),ap      # reset argument pointer
```

when no parameters are passed. The program in Figure A.11 incorporates these instructions (except for the stack adjustment which will be considered shortly); it contains 2500 call operations within a loop that is traversed 100 times, so that a total of 250,000 call and return operation pairs are performed. The program's execution time averaged 0.892 seconds, which divided by 250,000 yields 3.57×10^{-6} seconds as

```
        .text
        .globl    _f
_f:     rtn
        .globl    _main
_main:  sub.w     s7,s7
        mov       sp,a1
loop:
        mov       sp,ap          # four
        pshea     #0             # statements
        calls     _f             # duplicated
        ld.w      12(fp),ap      # 2500 times
        mov       a1,sp
        add.w     #1,s7
        le.w      #100,s7
        jmps.f    loop
        rtn
```

Figure A.11: Convex Program to Measure Call and Return Operation Pair Time Cost

[11]The CONVEX C compiler optimizer was disabled because its common-sequences-before-jumping optimization penalizes inlined code, as explained in Section 3.2.

the *Time Cost for each Call and Return Operation Pair*. Table A.23 presents the quantities gathered for *Count of Call and Return Operation Pairs Executed*, along with the ratio of the number of call and return operation pairs executed in a program's inlined version to the number executed in its noninlined version.

At every call site, the CONVEX C compiler produces code to transmit one more parameter than is passed in the corresponding source code.[12] The assembly language instruction to remove parameters from the stack after returning from each call is:

```
add.w       #nn,sp
```

where *nn* is the number of passed parameters, including the compiler-inserted one, times 4 (assuming the parameters are 4 bytes each). When only the compiler-inserted parameter is passed, a short variant of the instruction is used, but when other parameters are passed as well, a long variant of the instruction is employed. Two versions of the program in Figure A.12, with *<operation(s)-of-interest>* set in one version to *add.w #4,sp* and in the other version to *add.w #8,sp*, were used to time the instructions. The short

	Noninlined	Intramodule Inlined				Intermodule Inlined	
	Count	Count	Ratio	Count	Ratio	Count	Ratio
cache	4,428,056			2,025,454	0.46	758,481	0.17
calls	407,118			389,181	0.96	389,181	0.96
compact	85,000	37,229	0.44				
diff	820,213			802,397	0.98	786,389	0.96
dry	650,039	400,038	0.62				
extract	33,607	11,072	0.33				
grep	12,442	12,432	1.00				
inltest	800,082			412,454	0.52	331,300	0.41
invert	132,723			94,368	0.71	63,318	0.48
mincost	972,121	62,285	0.06				
nroff	610,110			452,494	0.74	498,545	0.82
sed	659,150			610,192	0.93	610,191	0.93
sort	855,095	311,157	0.36				
tsp	13,460,219	6,270,340	0.47				
vpcc	3,172,827			2,050,825	0.65	*	*
whetstone	272,750	119,836	0.44				
yacc	98,616			54,242	0.55	53,805	0.55
AVE			0.47		0.72		0.66

Table A.23: Convex Call and Return Operation Pairs Executed

[12]The value of the additional parameter is the number of parameters passed in the source code.

```
        .text
        .globl      _main
_main:  sub.w       s7,s7
loop:
        <operation(s)-of-interest>  # duplicated 5000 times
        add.w       #1,s7
        le.w        #100,s7
        jmps.f      loop
        rtn
```

Figure A.12: Convex Program to Measure Various Time Costs

stack adjustment version averaged 0.054 seconds of execution time and the long stack adjustment version averaged 0.102 seconds. Dividing the values by 500,000 (100 repetitions of 5000 stack adjustment instructions), the *Time Cost for Removing the Parameters from the Stack after Returning from each Call* is 1.080×10^{-7} seconds, for each subprogram call with one parameter passed (i.e., no parameters passed in the source code), and is 2.040×10^{-7} seconds, for each subprogram call with more than one parameter passed (i.e., at least one parameter passed in the source code). Table A.24 contains the data found for

	One Parameter Passed			More than One Parameter Passed		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	380,330	379,947	379,940	4,047,726	1,645,507	378,541
calls	3,601	3,601	3,601	403,517	385,580	385,580
compact	22	22		84,978	37,207	
diff	33,163	33,162	33,163	787,050	769,235	753,226
dry	150,037	50,036		500,002	350,002	
extract	4,752	4,468		28,855	6,604	
grep	74	74		12,368	12,358	
inltest	195,317	154,953	136,207	604,765	257,501	195,093
invert	56,497	56,490	56,490	76,226	37,878	6,828
mincost	35,634	35,634		936,487	26,651	
nroff	269,576	253,160	247,779	340,534	199,334	250,766
sed	242	240	240	658,908	609,952	609,951
sort	460	457		854,635	310,700	
tsp	6,270,339	6,270,339		7,189,880	1	
vpcc	991,671	702,876	*	2,181,156	1,347,949	*
whetstone	181,440	119,836		91,310	0	
yacc	14,458	14,449	14,446	84,158	39,793	39,359

Table A.24: Convex Counts of Parameter-Passing Call Operations Executed

Count of Parameters Passed during the Program's Execution.

The assembly language instruction to adjust the stack to accommodate a subprogram's stack-allocated local variables is:

```
sub.w       #nn,sp
```

where *nn* is the size in bytes of the space consumed by the stack-allocated local variables. If the stack-allocated local variables occupy between 1 and 7 bytes, then a short variant of the instruction is used, and if they occupy more than 7 bytes, then a long variant is employed. Two versions of the program in Figure A.12, with *<operation(s)-of-interest>* set in one version to *sub.w #4,sp* and in the other version to *sub.w #8,sp*, were used to time the instructions. The short stack adjustment version averaged 0.054 seconds of execution time and the long stack adjustment version averaged 0.098 seconds. Dividing the values by 500,000 (100 repetitions of 5000 stack adjustment instructions), the *Time Cost for Adjusting the Stack to Accommodate Stack-allocated Locals* is 1.080×10^{-7} seconds, for each subprogram with between 1 and 7 bytes of stack-allocated local variables, and is 1.960×10^{-7} seconds, for each subprogram with more than 7 bytes of stack-allocated local variables. Table A.25 contains the data found for *Count of Calls Executed to Subprograms with Stack-allocated Locals.*

As discussed in the introductory section of this appendix, parameters are assumed to be passed by the following operations:

```
psh.w       reg             # pass 32-bit register data item
pshea       nn              # pass 32-bit constant data item
ld.w        -mm(fp),s0      # pass 32-bit stack-resident data item
psh.w       s0              # pass 32-bit stack-resident, con't
```

each occurring with equal likelihood. Three versions of the program in Figure A.13, with *<parameter-passing-instr>* set in each respectively to *psh.w s7*, *pshea #60*, and *ld.w -4(fp),s0*,[13] were created for timing these parameter passing operations. The parameter passing operation times in seconds were 2.36×10^{-7}, 6.36×10^{-7}, and 6.68×10^{-7}, so the average time to pass a parameter is 5.13×10^{-7} seconds. These

[13]The code that saves and restores the value of the stack pointer was omitted from the version of the program containing the *ld.w -4(fp),s0* instruction, since the code was not needed with this instruction. The time cost for the instruction, which combined with this one passes a 32-bit stack-resident data item, was obtained from the first version of the program and was added to the time cost for this instruction to yield the total time cost to pass the data item.

	1 ≤ Size in Bytes of Stack-allocated Locals ≤ 7			Size in Bytes of Stack-allocated Locals > 7		
	Noninlined	Intramodule Inlined	Intermodule Inlined	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	386	2	0	1,356,653	881,945	378,534
calls	42,312	22,234	22,234	12,111	14,252	14,252
compact	0	0		74,246	37,207	
diff	28,513	22,261	0	755,309	746,975	753,227
dry	300,002	150,002		150,001	150,001	
extract	1,976	249		15,556	5,855	
grep	0	0		12,360	12,358	
inltest	292,909	43,732	22,560	137,021	155,363	161,230
invert	23,185	6,577	0	38,131	31,301	6,828
mincost	0	0		77,709	26,651	
nroff	135,322	26,487	20,159	363,962	411,668	464,327
sed	5	2	2	609,954	609,950	609,949
sort	8	0		310,704	310,709	
tsp	4,500,942	0		1,345,688	1	
vpcc	376,811	262,416	*	971,349	958,257	*
whetstone	1,400	0		89,903	1	
yacc	14,031	435	0	43,694	44,131	44,129

Table A.25: Convex Calls Executed to Subprograms with Stack-allocated Locals

```
        .text
        .globl      _main
_main:  sub.w       s7,s7
        mov         sp,a1              # save stack pointer value
loop:
        <parameter-passing-instr>  # duplicated 5000 times
        mov         a1,sp              # restore stack pointer value
        add.w       #1,s7
        le.w        #100,s7
        jmps.f      loop
        rtn
```

Figure A.13: Convex Program to Measure Parameter Passing Time Cost

parameter passing operations are replaced at inlined call sites, when actual parameter values are not copy-propagated, with the appropriate copy operations from the list below:

```
st.w        reg,-mm(fp)      # copy 32-bit register data item
ld.w        nn,s0            # copy 32-bit constant data item
st.w        s0,-mm(fp)       # copy 32-bit constant data item, con't
ld.w        -pp(fp),s0       # copy 32-bit stack-resident data item
st.w        s0,-mm(fp)       # copy 32-bit stack-resident, con't
```

assuming the local temporary replacing the formal parameter is stack-resident. The times in seconds of

the copy operations were 2.84×10^{-7}, 4.92×10^{-7}, and 7.16×10^{-7}, so the average time to copy a parameter is 4.97×10^{-7} seconds. Computing *Time Cost for Passing a Parameter – Time Cost for Handling a Parameter at an Inlined Call Site* as explained in the introductory section of this appendix, i.e., as the average cost of passing a parameter minus two-thirds the average cost of copying it, the result is 1.82×10^{-7} seconds. Table A.26 contains the values gathered for *Count of Parameters Passed during the Program's Execution.*

Register save and restore operations are performed using the instructions:[14]

```
st.w    reg,mm(fp)      # save register
ld.w    mm(fp),reg      # restore register
```

The time cost for saving and restoring a register was determined utilizing a version of the program given in Figure A.12, with *<operation(s)-of-interest>* set to the pair *st.w s7,-4(fp)* and *ld.w -4(fp),s7*. Dividing the version's average execution time (0.336 seconds) by 500,000, the *Time Cost for Saving and Restoring*

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	5,953,446	3,127,065	1,135,615
calls	766,907	748,910	748,910
compact	127,907	37,208	
diff	2,302,842	2,251,548	2,225,790
dry	1,050,004	750,004	
extract	66,155	17,768	
grep	24,726	24,716	
inltest	866,211	343,461	279,127
invert	169,175	99,308	13,656
mincost	2,239,078	159,902	
nroff	401,983	202,928	308,228
sed	1,195,430	1,097,518	1,097,517
sort	1,140,854	596,918	
tsp	21,899,871	2	
vpcc	4,124,760	2,680,051	*
whetstone	271,170	0	
yacc	155,711	73,510	72,209

Table A.26: Convex Counts of Parameters Passed during the Programs' Execution

[14]Unlike the other three computer systems considered, the Convex utilizes the caller-save convention for retaining register values across subprogram calls. The instructions given above are employed when saving and restoring a register containing a local variable. When saving and restoring a register containing a parameter value, the offset is expressed relative to *ap* rather than to *fp*.

each Register is found to be 6.720×10^{-7} seconds. The *Count of Registers Saved and Restored during the Program's Execution* is given in Table A.27.

	Noninlined	Intramodule Inlined	Intermodule Inlined
cache	2,561,192	3,500,599	1,464
calls	1,357,419	1,555,580	1,555,580
compact	191,436	148,860	
diff	2,353,991	3,093,941	2,384,085
dry	1,850,030	1,600,044	
extract	57	42,541	
grep	37,182	37,203	
inltest	1,035,436	1,438,127	1,225,792
invert	44,357	286,763	183,280
mincost	0	186,219	
nroff	1,336,895	1,388,214	1,701,807
sed	977,118	2,391,286	2,391,296
sort	2,302,513	719,714	
tsp	6	24,991,584	
vpcc	6,136,093	6,048,888	*
whetstone	0	0	
yacc	180,268	186,278	205,486

Table A.27: Convex Registers Saved & Restored during the Programs' Execution

References

[AHU74] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[AlJ88] R. Allen and S. Johnson, "Compiling C for Vectorization, Parallelization, and Inline Expansion", *SIGPLAN Notices 23*, 7 (1988), 241-249.

[BaT86] H. E. Bal and A. S. Tanenbaum, "Language- and Machine-Independent Global Optimization on Intermediate Code", *J. Computer Lang. 11*, 2 (1986), 105-121.

[Bal79] J. E. Ball, "Predicting the Effects of Optimization on a Procedure Body", *SIGPLAN Notices 14*, 8 (1979), 214-220.

[BCK89] P. Briggs, K. D. Cooper, K. Kennedy and L. Torczon, "Coloring Heuristics for Register Allocation", *SIGPLAN Notices 24*, 7 (1989), 275-284.

[CON85] CONVEX, *CONVEX Architecture Handbook*, Convex Computer Corporation, Richardson, TX, 1985.

[CAC81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, "Register Allocation via Coloring", *J. Computer Lang. 6* (1981), 47-57.

[Cha82] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring", *SIGPLAN Notices 17*, 6 (1982), 98-105.

[Cho83] F. C. Chow, "A Portable Machine-Independent Global Optimizer - Design and Measurements", Tech. Rep. 83-254, Stanford University, 1983.

[ChH84] F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring", *SIGPLAN Notices 19*, 6 (1984), 222-232.

[Cho88] F. C. Chow, "Minimizing Register Usage Penalty at Procedure Calls", *SIGPLAN Notices 23*, 7 (1988), 85-94.

[Cli89] Clipper Software, *CLIX System V Release 3.0 Reference Manual*, Intergraph Corporation, Huntsville, AL, 1989.

[CKT86] K. D. Cooper, K. Kennedy and L. Torczon, "The Impact of Interprocedural Analysis and Optimization in the R^n Programming Environment", *Trans. Prog. Lang and Systems 8*, 4 (1986), 491-523.

[CHT90] K. D. Cooper, M. Hall and L. Torczon, "An Experiment with Inline Substitution", Rice Computer Tech. Rep. 90-128, Rice University, 1990.

[DaH88] J. W. Davidson and A. M. Holler, "A Study of a C Function Inliner", *Software—Practice & Experience 18* (1988), 775-790.

[DaW89] J. W. Davidson and D. B. Whalley, "Methods for Saving and Restoring Register Values across Function Calls", Computer Science Tech. Rep. 89-11, University of Virginia, 1989.

[DaH89] J. W. Davidson and A. M. Holler, "Subprogram Inlining: A Study of its Effects on Program Execution Time", Computer Science Tech. Rep. 89-4, University of Virginia, 1989.

[Den80] P. J. Denning, "Working Sets Past and Present", *IEEE Trans. on Software Eng. SE-6*, 1 (1980), 64-84.

[DeS90] R. B. K. Dewar and M. Smosna, *Microprocessors: A Programmer's View*, McGraw Hill, New York, NY, 1990.

[Fai87] Fairchild, *Clipper 32-Bit Microprocessor User's Manual*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[Fre74] R. A. Freiburghouse, "Register Allocation via Usage Counts", *Comm. of the ACM 17*, 11 (1974), 638-642.

[Hal89] M. W. Hall, *Interprocedural Transformations in a Programming Environment*, Doctoral Thesis Proposal, Rice University, 1989.

[HaS84] S. P. Harbison and G. L. Steele, *C: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[Har77] W. Harrison, "A New Strategy for Code Generation - the General Purpose Optimizing Compiler", *4th ACM Symp. on Prin. of Prog. Lang.*, 1977, 29-37.

[HaG71] D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory", *IBM Systems Journal 10*, 3 (1971), 168-192.

[Hec77] M. S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York, NY, 1977.

[Hus82] C. A. Huson, "An In-line Subroutine Expander for Parafrase", Tech. Rep. UIUCDCS-R-82-1118, University of Illinois at Urbana-Champaign, 1982.

[HwC89a] W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling C Programs", *SIGPLAN Notices 24*, 7 (1989), 246-257.

[HwC89b] W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", *ACM SIGARCH Computer Architecture News 17*, 3 (1989), 242-251.

[JeW78] K. Jensen and N. Wirth, *PASCAL User Manual and Report*, Springer-Verlag, New York, NY, 1978.

[KeR78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

[Kuc78] D. J. Kuck, *The Structure of Computers and Computations*, John Wiley & Sons, New York, NY, 1978.

[LPI88] S. Laha, J. H. Patel and R. K. Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems", *IEEE Trans. on Computers 37*, 11 (1988), 1325-1336.

[LaH86] J. R. Larus and P. N. Hilfinger, "Register Allocation in the SPUR Lisp Compiler", *SIGPLAN Notices 21*, 7 (1986), 255-263.

[Leo87] T. E. Leonard, ed., *VAX Architecture Reference Manual*, DECbooks, Maynard, MA, 1987.

[LoM69] E. S. Lowry and C. W. Medlock, "Object Code Optimization", *Comm. of the ACM 12*, 1 (1969), 13-22.

[Mac84] M. D. MacLaren, "Inline Routines in VAXELN Pascal", *SIGPLAN Notices 19*, 6 (1984), 266-275.

[MaD74] S. E. Madnick and J. J. Donovan, *Operating Systems*, McGraw Hill, New York, NY, 1974.

[McT83] H. McGilton and R. Tuck, *Assembly Language Reference Manual for the Sun Workstation*, Sun Microsystems, Inc., Mountain View, CA, 1983.

[MMS79] J. G. Mitchell, W. Maybury and R. Sweet, "Mesa Language Manual", CSL-79-3, Xerox Palo Alto Research Center, 1979.

[Mot85] Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice Hall, Englewood Cliffs, NJ, 1985.

[PaS82] D. A. Patterson and C. H. Sequin, "A VLSI RISC", *Computer 15*, 9 (1982), 8-21.

[Pat85] D. A. Patterson, "Reduced Instruction Set Computers", *Comm. of the ACM 28*, 1 (1985), 8-21.

[PaH90] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[PeS86] J. L. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison Wesley, Reading, MA, 1986.

[Pra84] T. W. Pratt, *Programming Languages: Design and Implementation*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[ReH83] J. F. Reiser and R. R. Henry, *Berkeley VAX/UNIX Assembler Reference Manual*, Berkeley System Distribution, Berkeley, CA, 1983.

[RiG89a] S. Richardson and M. Ganapathi, "Code Optimization Across Procedures", *Computer 22*, 2 (1989), 42-50.

[RiG89b] S. Richardson and M. Ganapathi, "Interprocedural Analysis vs. Procedure Integration", *Inf. Proc. Letters 32* (1989), 137-142.

[Sch77] R. W. Scheifler, "An Analysis of Inline Substitution for a Structured Programming Language", *Comm. of the ACM 20*, 9 (1977), 647-654.

[Smi82] A. J. Smith, "Cache Memories", *Computing Surveys 14*, 3 (1982), 473-530.

[Smi87] A. J. Smith, "Line (Block) Size Choice for CPU Cache Memories", *IEEE Trans. on Computers 36*, 9 (1987), 1063-1074.

[Sto87] H. S. Stone, *High-Performance Computer Architecture*, Addison Wesley, Reading, MA, 1987.

[Sun86a] Sun Microsystems, *FORTRAN Programmer's Guide*, Sun Microsystems, Inc., Mountain View, CA, 1986.

[Sun86b] Sun Microsystems, *Pascal Programmer's Guide*, Sun Microsystems, Inc., Mountain View, CA, 1986.

[Tan87] A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[Uni83] United States Department of Defense, *Reference Manual for the Ada Programming Language, American National Standards Institute/MIL-STD-1815A-1983*, U.S. Government Printing Office, January 1983.

[Wal86] D. W. Wall, "Global Register Allocation at Link Time", *SIGPLAN Notices 21*, 7 (1986), 264-275.

[WRH71] W. A. Wulf, D. B. Russell and A. N. Habermann, "BLISS: A Language for Systems Programming", *Comm. of the ACM 14*, 12 (1971), 780-790.