1977 ACM Student Award
Paper: First Place

An Analysis of Inline Substitution for a Structured Programming Language

Robert W. Scheifler
Massachusetts Institute of Technology

An optimization technique known as inline substitution is analyzed. The optimization consists of replacing a procedure invocation by a modified copy of the procedure body. The general problem of using inline substitution to minimize execution time subject to size constraints is formulated, and an approximate algorithmic solution is proposed. The algorithm depends on run-time statistics about the program to be optimized. Preliminary results for the CLU structured programming language indicate that, in programs with a low degree of recursion, over 90 percent of all procedure calls can be eliminated, with little increase in the size of compiled code and a small savings in execution time. Other conclusions based on these results are also presented.

Key Words and Phrases: inline substitution, open coding, open compilation, program optimization, compilers, structured programming languages, run-time statistics
CR Categories: 4.12

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract N00014-75-C-0661. Author's address: Programming Methodology Group, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

1. Introduction

In recent years various studies have been undertaken to determine the efficacy of program optimization [6, 10, 15, 16]. Results from these studies generally indicate that current optimization techniques are indeed worthwhile for conventional algebraic and general-purpose languages [6, 15]. With the advent of structured programming and structured programming languages, it is important to determine which existing techniques are more effective, and which are less effective, when applied to structured programs. This paper analyzes the efficacy of a technique known as inline substitution. CLU [7], a structured programming language supporting data abstractions [8] currently under development at M.I.T., serves as the test vehicle.

In the remainder of this section, we discuss how inline substitution directly affects program size and execution time and how use of the technique can affect other optimizations. Section 2 considers the problem of using inline substitution to minimize execution time subject to size constraints and develops an approximate algorithmic solution. The algorithm has been implemented, and preliminary results are given in Section 3, with conclusions based on these results presented in Section 4.

1.1 Inline Substitution

Simply stated, inline substitution consists of replacing a *procedure invocation* by a modified copy of the procedure body; i.e. the body is substituted for the invocation. By invocation we mean a statement or expression that, when executed, *calls* an explicitly named procedure, and we include the notions of subroutine, subprogram, and function under the general term procedure. In literature dealing with conventional languages, this technique is sometimes known as open coding or open compilation of subroutines [3, 14].

Exactly how the body is modified depends on the particular language used. Generally the body must be enclosed in some construct, such as a **begin-end** block, to provide for local definitions of formal arguments and other variables. In addition, argument passing must be made explicit. Each call-by-value formal is assigned the corresponding actual argument; each instance of a call-by-name formal is replaced by the corresponding actual; and so on. Conflicts between free names of the actual arguments and local names of the body are resolved by a systematic renaming, as are conflicts between free names of the modified body and those names whose scope includes the invocation.

Figure 1 presents a sample substitution in CLU. We note that a CLU variable is not an object that holds a value, but simply a name used to denote an object. The assignment $x := y$ makes the variable x denote the same object that y currently denotes. Any change in the state of that object will be visible through both variables. Further, arguments are passed by assignment; in the example, one can think of the array argument as being passed by reference and the integer arguments as being passed by value.

It is not hard to see that the expansion shown is semantically equivalent to the given invocation. The meaning of a subroutine call is often described [11] and even formally defined [9] by the *copy rule*, which is the very process of inline substitution. However, the source code to source code transformation stated above fails to work correctly in many instances for a wide variety of languages. Most block-structured languages only allow blocks as statements; so the expansion of a function call within an expression is impossible without complex rewriting rules. If a return from a procedure is allowed at any point in the procedure body, then there must be a comparable exit mechanism for **begin-end** blocks; otherwise many subroutine calls could not be expanded. Of course, optimization is not usually done on source code, but rather on some intermediate representation between source and object code. Some optimizers, though, convert this representation back to source code [15], and there are definite problems in incorporating inline substitution into such optimizers.

Several optimizations are possible in the expanded invocation shown. Each of the first three assignments could be removed by replacing all occurrences of the left-hand side by the right-hand side in the rest of the block, and the conditional test could be eliminated if $n > 1$ were known to hold. Certain simple optimizations can be incorporated into the substitution algorithm, but this is unnecessary if other techniques, that in part achieve the same results, are also employed.

Fig. 1. Sample substitution. Top: part of a quicksort program [4]. Bottom: an expansion of *qsort(L, 1, n)*. *Partition(A, b, e, v)* rearranges the elements of A with indexes between b and e and returns m such that $(\forall i, b \le i \le e)[i \le m \leftrightarrow A[i] \le v]$ holds.

```
qsort = proc (A: array[int], b: int, e: int);
  if e > b
    then
      v: int := (A[b] + A[e])/2;
      m: int := partition(A, b, e, v);
      qsort(A, b, m);
      qsort(A, m + 1, e);
    end;
  end qsort;

begin
  A: array[int] := L;
  b: int := 1;
  e: int := n;
  if e > b
    then
      v: int := (A[b] + A[e])/ 2;
      m: int := partition(A, b, e, v);
      qsort(A, b, m);
      qsort(A, m + 1, e);
    end;
  end;
```

1.2 Direct Cost Effect

Conceptually we can think of a procedure invocation as being implemented by a single mechanism, which we shall refer to as the *call* mechanism. Part of the size change resulting from a substitution is the difference between the size of the expanded invocation and the size of that part of the call mechanism originally present in the code. In addition, when the last remaining invocation of a procedure is expanded, it may be possible to discard the procedure; this is also counted in the size change. Although optimizers tend to concentrate on reducing execution time, we consider both time and space to be scarce in a computer system. The efficacy of inline substitution thus depends to a large extent on the average size change, as well as the average execution time of the call mechanism.

Consider a program composed of just a few large procedures. When an invocation is expanded there is a substantial increase in program size, provided more than one invocation of the procedure exists. The execution time of the expanded body will be very large compared to that of the call mechanism, and so the time savings will only begin to offset the size increase when the invocation occurs inside an inner loop of the program, i.e. in a section of code that represents a major fraction of the total program execution time. Hence inline substitution would not appear to be generally useful for such a program. There is some empirical evidence to support this statement. Knuth, for example, uses the technique only twice in his well-known study of Fortran programs [6] and (perhaps arbitrarily) only during the "best conceivable" "anything goes" phase of optimization. Descriptions of many optimizers make no mention of the technique.

A program written in a structured programming language is constructed by repeated division of a problem into subproblems, each expressed as a procedure. As a result, structured programs tend to be composed of a collection of many small procedures. Introducing abstract data types [8] into the language increases the likelihood that programs will contain a large number of small procedures. An abstract data type consists of a set of data objects and a set of operations for creating and manipulating the objects. Each operation is implemented by a separate procedure; in practice many of these procedures are very small.

Since procedures in structured programs are small, we may assume that the time to execute a procedure body is correspondingly small. Hence the call mechanism may represent a considerable amount of overhead in both time and space. At some point it becomes cost effective to perform inline substitution to reduce this overhead. For some procedures the call mechanism may be as large as the expanded invocation, and total cost will decrease when the substitution is made. If all invocations of a procedure are expanded, the procedure itself may no longer be needed, further reducing total cost. Thus, for structured programming languages, inline substitution may significantly improve execution time with only a small increase in size.

1.3 Effect on Other Optimizations

Of course, inline substitution is not expected to be the only practical optimization technique for structured programming languages. Yet the success of intraprocedural optimizations may depend crucially on its use. In small procedures one would expect to find constants as arguments to procedures more often than as right-hand sides of assignments; hence there will be fewer opportunities to propagate constants than in large procedures, and so the probability of finding dead code is lower. One would also expect fewer discoveries of redundant computations, partly because they often will be separated by procedure boundaries, and partly because a programmer is less likely to generate them, there being less code to comprehend at once. Moreover, large portions of loop bodies may be written as separate procedures, making some loop optimizations less likely.

This kind of situation has been analyzed by Zelkowitz and Bail [16] using the structured programming language SIMPL. Two global and two local optimization techniques were used, with the result that almost no optimization was possible. Palm [10] implemented various local and global optimizations for C, a machine-oriented general-purpose language. The test programs were composed of many small procedures, and again only marginal improvements in execution time were gained. Palm gives several other reasons for this, but suggests that perhaps inline substitution is the key.

When several optimization techniques are used, the order of optimization is important. We believe inline substitution should be performed first. Theoretically this does little harm, since an optimization that is possible before inline substitution is almost always possible afterwards.[1] When an invocation is expanded, the fact that it was once an invocation can be remembered, and information (e.g. specifications) about the called procedure can be used directly without trying to derive that information from the expanded body. Of course, extra bookkeeping is required to remember such data, and the multiple copies of procedure bodies produced by substitution will mean redundant computation in later optimizations. However, the alternative, using other techniques both before and after inline substitution, is probably still more expensive.

Various optimization techniques, including inline substitution, have been proposed for languages like CLU by Atkinson [2]. He speculates that if the rest of the optimizer can be made "smart" enough, the dominant effect of inline substitution may well be the benefit afforded to other optimizations, this benefit ultimately producing greater improvements than those obtained directly from use of the technique. The remainder of this paper, however, it limited to an analysis of inline substitution in the absence of other optimizations.

[1] A space-increasing optimization is one kind of optimization that might not be possible, if there are space constraints such as those developed in the next section.

2. Problem Formulation and Solution

We would like to be able to incorporate inline substitution into an optimizer in such a way that little if any decision-making input is required from the user. We believe that the user is responsible for the *logical* structure of a program, not the physical structure. Restricting our attention to inline substitution as an isolated technique, it is reasonable to ask if there is an optimal way to use the technique.

In practice it is not possible to simply expand all procedure invocations. Not only will such a process never terminate for recursive procedures, but usually there will be some constraint placed on the total program size. There also may be a limit on the size of individual procedures. For instance, the compiler used in our study could not compile extremely large procedures. Moreover, it is not always desirable to optimize an entire program. One might choose some subset of all procedures and only consider expanding *internal* invocations: those that call from a procedure in the subset to a procedure in the subset. The problem is thus one of constrained minimization:

Substitution Problem. Given a program, a subset of all invocations, a maximum program size, and a maximum procedure size, find a sequence of substitutions that minimizes the expected program execution time.

We note that in reality there are absolute bounds on the sizes of programs and procedures, though the problem statement does not impose such constraints. Further, in practice, the absolute maximum procedure size is used for every program. We shall return to these points later.

Assuming there is a way to compute the time saved by any sequence, there clearly is a solution to the Substitution Problem. For any given (nontrivial) program there are only a finite number of sequences that will not violate the size constraints, and all such sequences can be generated systematically. However, as the number of invocations increases, there will be a combinatorial explosion in the number of possible sequences, and such a scheme will require exponential (in the number of invocations) time to solve the Substitution Problem.

2.1 Complexity of the Problem

Unfortunately we must conjecture that the Substitution Problem is not tractable, that any solution will require exponential time for some set of programs. To support this belief we give an efficient reduction, suggested by Rivest [12], of the Knapsack Problem to the Substitution Problem; this reduction will show that the Substitution Problem is at least as hard as the Knapsack

Problem. The Knapsack Problem is known to be NP-complete[2] and so is thought to have no tractable solution:

Knapsack Problem. Given a sequence of positive integers $S = i_1, \ldots, i_n$ and an integer k, is there a subsequence that sums to exactly k?

The reduction is based on two almost trivial assumptions about the Substitution Problem. First, we assume it is possible to construct an invocation with any integral execution time *overhead.* By (execution time) overhead we mean the total overhead for all executions of the given invocation. For the purpose at hand, we can choose a fixed call mechanism to be used for all invocations and define the per call overhead of this mechanism to be one unit of time. By placing an invocation in a loop we can get any integral overhead. The second assumption is that it is possible to construct a procedure of any integral size less than the maximum. Again, we can define the size of a particular block of code to be one unit of space and build all procedure bodies with sequences of this block.

For each integer i_j in $S = i_1, \ldots, i_n$, we construct procedures P_j and R_j. P_j contains two invocations of R_j, but only one is included in the set of invocations that can be expanded. We define the overhead of this one invocation to be i_j units of time. R_j contains no invocations, and the size increase resulting from its substitution is defined to be i_j units of space.

We define the maximum procedure size so that all substitutions are possible, but we constrain the maximum program size increase to k. By construction, the execution time saved by any sequence of substitutions will be exactly the same as the resultant size increase, and so the maximum time savings possible is k. If there is a subsequence of S that sums to k, then there is a sequence of substitutions that will obtain the maximum time savings. Conversely, if there is a sequence that obtains the maximum time savings, then there is a subsequence of S that sums to k. Hence we can solve the Knapsack Problem by solving the Substitution Problem. Any tractable solution to the Substitution Problem is thus a tractable solution to the Knapsack Problem, for we can certainly perform the above construction in polynomial time.

The reduction we have given depends on the fact that the maximum procedure size and maximum program size can be made arbitrarily large. If there are absolute bounds on these sizes, it may be possible to solve the Substitution Problem in polynomial time. However, as long as the bounds are reasonably large, the polynomial will be of very high degree, and so the problem is still essentially intractable.

In view of the above, it is reasonable to take an engineering approach and make certain approximations in a manner that will arrive, we hope, at a near-optimal solution. To begin, we need some measure of size and some measure of the execution time overhead for any given invocation. The size change resulting from a substitution is easy to compute, but we need a method for determining how the overhead of an invocation changes as other invocations are expanded. We make several practical approximations to fulfill these needs and then return to the actual problem solution at the end of this section.[3]

2.2 Size Approximation

There are a number of possible size measures, such as the number of symbols in the code being optimized, the number of symbols in the object code, or the number of machine words occupied by the object code. This last measure is perhaps the best, and often can be determined exactly during the optimization phase. However, there are situations where this is not convenient. For example, it may be very expensive to calculate object code size from the representation being optimized.

A very simple approximation of object code size can be made: an operator expression with n operands occupies $n + 1$ words plus the words occupied by each operand that is an expression. For example, the expression $a/(b + c)$ would occupy six words. In the usual tree representation, this corresponds to summing the number of leaves less one with twice the number of nodes. Special cases might be made of certain operators, e.g. those known to map into single machine instructions. Control structures can be treated in a similar manner. An **if-then-else** statement might occupy two words for branch instructions plus the words occupied by the predicate and clauses. The actual approximation seems unimportant as long as it gives a good indication of relative size; the size constraints will probably be estimates, and they can easily be expressed as changes relative to initial sizes.

2.3 Run-Time Statistics

To determine the execution-time overhead of an invocation we need to know to overhead *per call* and the expected number of executions of the invocation. The overhead per call can be calculated by examining the code for the invocation. Although one might attempt to derive the expected number of executions from a static analysis of the program, we prefer a more accurate approach. The program is run with various sets of input data and statistics are gathered. These statistics are used to calculate the expected overhead of each invocation.

The calculations can be simplified by assuming that the overhead per call is the same for *every* invocation. This is probably a good approximation in general. The method used to alter the environment and transfer control will usually be the same for each invocation.

[2] Nondeterministic polynomial time complete. See, for example, [1].

[3] Many of the ideas in the rest of this section were first proposed by Atkinson [2].

Procedures in many languages, especially languages supporting data abstractions, have few formal arguments and fewer return values, and, in practice, transmission types are chosen in a way that allows each argument to be passed in about the same amount of time; so the time spent in data transmission is essentially constant. Thus we can simply use the expected number of executions of an invocation as the invocation's execution time overhead.

Suppose the body of a procedure P contains an invocation $S(x)$. When an invocation $P(y)$ is expanded, a new $S(x)$ will be created and must be assigned an expected number of executions. The ideal way to determine this number is to keep a statistic on how many times the old $S(x)$ executes as a result of P's being called from $P(y)$. Such a two-level history of control flow would entail multiple counters for each invocation. Further, a three-level history would be required to expand the new $S(x)$ if S had an invocation $R(z)$, and so on.

As a practical matter, the cost of gathering statistics should be low; it is probably too expensive, in both time and space, to keep even a two-level history. To avoid a multilevel history, we choose to make the following assumption about control flow:

Constant Ratios Assumption. For any procedure body and any invocation contained therein, the expected number of executions of the invocation per execution of the body is constant.

Although this assumption is very strong, it is the most reasonable one we can think of that enables us to get by with just a one-level history. Exactly how this assumption is used will be developed further below.

In addition to keeping statistics for invocations, it often will be necessary to count the number of times each procedure is called. In some cases a procedure may be called via some mechanism other than an explicit invocation of that procedure. For example, the language might allow calls through procedure variables. More importantly, if the invocations considered for expansion form a small subset of all invocations, the cost of keeping statistics for all invocations can be prohibitively greater than the cost of keeping statistics only for invocations in the subset, additionally counting the number of times each procedure is called.

2.4 Calculating Expected Numbers

Figure 2 gives a tabular form of the example to be used in deriving how new expected numbers are calculated and old expected numbers are modified when a nonrecursive substitution is performed. In the Before table, procedures P, S, and R are called p, s, and r times, respectively. P has an invocation of S with $s1$ executions, and S has an invocation of R with $r1$ executions. The After table shows what happens when P's invocation of S is expanded. The invocation of S disappears, and the expansion creates a new invocation of R, in the body of P, with some number of executions $r2$.

Fig. 2. Nonrecursive substitution.

Before expanding $S(. . .)$:	After:
P has p calls	P has p calls
$S(. . .)$ has $S1$ executions	$R(. . .)$ has $r2$ executions
S has s calls	S has s' calls
$R(. . .)$ has $r1$ executions	$R(. . .)$ has $r1'$ executions
R has r calls	R has r calls

$$\begin{aligned} r2 &= s1 * (r1/s) \\ s' &= s - s1 \\ r1' &= r1 - r2 \end{aligned}$$

Fig. 3. Recursive substitution.

Before expanding $P(. . .)$:	After:
P has p calls	P has p' calls
$P(. . .)$ has $p1$ executions	$P(. . .)$ has $p2$ executions
	$R(. . .)$ has $r2$ executions
$R(. . .)$ has $r1$ executions	$R(. . .)$ has $r1'$ executions
R has r calls	R has r calls

$$\begin{aligned} p' &= p - p1' \\ p2 &= p1' * (p1/p) \\ r2 &= p1' * (r1/p) \\ r1' &= r1 - r2 \\ p1' &= p * p1/(p + p1) \end{aligned}$$

Clearly the number of calls to S is reduced by the number of executions of the invocation being expanded, so that $s' = s - s1$. Since S itself is not called as often, the number of executions of S's invocation of R also decreases. This decrease must be compensated by the number of executions of the new invocation of R, for certainly the substitution should not alter the number of times R is called. Thus $r1' = r1 - r2$. The number of calls to P should likewise remain the same. Last, the Constant Ratios Assumption implies that $r1'/s' = r1/s$. We can substitute for $r1'$ and s' in this equality and find that $r2 = s1 * (r1/s)$.

Figure 3 depicts a situation before and after the expansion of a recursive invocation. In the After table, the invocation of P and the invocation of R with $r2$ executions are created by the expansion. To do the derivation we also need the number of executions, $p1'$, of the expanded invocation.

As in the nonrecursive case, the number of calls to R must remain the same; so $r1' = r1 - r2$. The executions of the old recursive invocation must go somewhere; some reappear as executions of the new recursive invocation, and the rest must remain executions of the expanded invocation itself, which makes $p1' = p1 - p2$. Thus the number of calls to P decreases by the number of executions of the expanded invocation, or $p' = p - p1'$.

All that remains is to decide how to apply the Constant Ratios Assumption. The assumption concerns the body of a procedure, not the name. It states that, for any execution of the body of a procedure, the number of times an invocation contained in that body is expected to execute is independent of the control flow history prior to the execution of the body. Therefore the assumption must also apply to any copy of the body that is substituted for an invocation, i.e. the expected number of executions of an invocation contained in a copy, per execution of the copy, will be the same as for the original body of the procedure. In the current example this means that $p2/p1' = p1/p$ and that $r2/p1' = r1/p$, which leads to the solution in Figure 3.

2.5 Order of Substitution

The algorithm we choose has three distinct phases. At each step in the first phase, every substitution that will result in a nonpositive *basic* size change is performed, independent of the time saved; the process repeats until no further substitutions are possible by this rule. The basic size change for a substitution is the change in the size of the procedure body containing the invocation. The reduction in program size possible if the called procedure can be removed is *not* counted in the basic size change, but it *is* counted in determining the program size after the substitution. We use the basic size change in an attempt to keep procedure sizes small. As a heuristic, we would rather substitute a small frequently called procedure often than expand one invocation of a very large procedure within that small procedure.

The second phase is the heart of the algorithm. At each step, the invocation with the highest ratio of expected executions to basic size change, whose expansion will not violate the procedure size constraint, is expanded. Any invocations created by the substitution are available at the next step. The order of substitution can be computed very cheaply by using this rule, but there are no mathematical guarantees that the result will be near-optimal.[4] The hope, of course, is that good results are obtainable for this specific application and that more expensive calculations are not required. Substitutions continue according to this rule until the maximum program size is just exceeded. We prefer not to complicate the rule to avoid violating the size constraint; the maximum is, after all, only approximate, and in general the additional increase will be small.

Once the second phase is complete, it still may be possible to expand some invocations without increasing the total program size. When a nonrecursive procedure has exactly one invocation to it remaining in the entire program and the procedure is known not to be called by any other mechanism, expansion of the invocation also allows the procedure to be removed, and little if any total size increase is involved. All such expansions that do not violate the procedure size constraint are performed in the third and final phase.

3. Preliminary Results

At the time of this work, the CLU compiler was a high-level to high-level language translator, producing object code in MDL [5], a Lisp-like language currently used as the base language for the M.I.T. Dynamic Modeling System. Inline substitution is often impossible in CLU, as well as in the compiler's internal representation, but the transformation is always possible for the MDL code produced by the compiler. For this reason, the substitution algorithm was implemented for MDL, with one extension. If, when expanding an invocation, an actual argument is a constant or a variable and the corresponding formal is not assigned to in the procedure body, the actual-to-formal assignment is eliminated by replacing all occurrences of the formal by the actual in the expanded body.

Four programs were tested: (1) a "programmed" space war game written in MDL, but containing many small procedures, (2) the part of the MDL compiler that orders a set of procedures for compilation, (3) a simple infix-to-postfix expression translator written in CLU, and (4) the CLU compiler, which is written largely in CLU. For each of the first three programs, all invocations were included for possible expansion. Two separate tests were made with the CLU compiler. First the most often used system support routines, written in MDL, were optimized. That is, only invocations internal to these procedures were considered. In the second test, the part of the compiler dealing with declarations, written in CLU, was optimized along with these original support routines.

For Programs 1, 2, and 3, the maximum program size was set at twice the size of the original program. For Program 4 the maximum was such that the subset of procedures being optimized could double in size. The maximum procedure size was the same for every test. To determine the accuracy of the size approximation, both the MDL code and the object code from the MDL compiler were measured before and after optimization. After substitutions were complete, the accuracy of the remaining expected numbers was checked by gathering statistics for the optimized program. Execution time was measured both before and after optimization. A summary of the results appears in Table I. For each test, the table shows the various measurements prior to inline substitution and the actual changes produced by optimization. The change in the number of calls and the change in program size, as calculated by the algorithm, are also shown. We now discuss these results.

3.1 Size Constraints

The procedure size constraint turned out to be necessary. Initially, an essentially infinite size was allowed, with the result that some procedures ended up too large for the MDL compiler to handle. However, the constraint was not an important factor in the final outcome. The reductions in the expected number of calls under the relaxed constraint were only marginally better than those shown, with essentially the same expected changes in total size.

[4] This is not an ϵ-approximate algorithm as defined by Sahni [13].

Table I. Summary of Results.

		Procs	Invs	Calls	Time	Object	Source	Lines
1.	Initial:	77	234	316528	347.5	12357	4687	822
	Change:	−64%	−65%	−92%	−28%	−1%	100%	
	Expected:			−92%		−100%		
2.	Initial:	15	28	9645	8.3	2903	1515	214
	Change:	−53%	89%	−50%	−5%	87%	256%	
	Expected:			−57%		167%		
3.	Initial:	15	13	769	1.50	1734	917	128
	Change:	±93%	−100%	−100%	−17%	−67%	−22%	
	Expected:			−100%		−2%		
4a.	Initial:	79	142	448979	1248	10981	3966	525
	Change:	0%	−22%	−100%	−13%	9%	34%	
	Expected:			−100%		44%		
4b.	Initial:	167	441	779390	1281	31020	16585	2745
	Change:	−25%	−1%	−97%	−7%	22%	86%	
	Expected:			−97%		105%		

Subs:	1	2	3	4a	4b
	158	23	13	38	234

Procs	= number of procedures
Invs	= number of internal invocations
Calls	= number of internal calls
Time	= execution time of compiled code in CPU seconds
Object	= size of object code in machine words
Source	= size of MDL source code in machine words (5 chars/word)
Lines	= number of source text lines
Subs	= number of substitutions performed

In four of five tests the vast majority of calls were eliminated without exceeding the maximum program size. Program 2 was the exception. For this program, 93 percent of all calls were from recursive invocations of a single procedure. Size grew very quickly as the recursive invocations were expanded, but the expected number of calls decreased rather slowly. Yet, even for this program, the program size constraint did not significantly affect the number of calls eliminated. Allowing *all* possible substitutions (by using a larger maximum program size) only reduced the expected number of calls another 15 percent, with an expected total size increase of 266 percent.

3.2 Expected Numbers

Perhaps the most striking aspect of Program 1 was that the expected numbers calculated by the algorithm agreed exactly, for all invocations of all procedures, with the numbers obtained by rerunning the optimized program. The reason is that calculations are only approximate when the Constant Ratios Assumption is used; none of the substitutions for Program 1 required the assumption. The assumption is not used when substituting a body containing either no invocations or only invocations with zero expected numbers, nor when expanding the last remaining invocation of a procedure.

Approximate calculations were required for four substitutions in Program 2, involving a total of 25 invocations. Five of these invocations were eventually expanded exactly, leaving 20 approximate numbers, most of which did not match the actual numbers very closely. All calculations for Programs 3 and 4(a) were exact, and in both cases all calls were eliminated. Fifteen substitutions in Program 4(b) used approximate calculations, leaving 84 invocations with approximate numbers. Only the total calls to each procedure were counted in rerunning this program; so the accuracy of the individual numbers is not known.

The results indicate that the Constant Ratios Assumption may be a poor model of control flow. However, the results also show that the use of a multilevel history as a more accurate model is not warranted. It seems that in most cases virtually all calls can be eliminated with only a small increase in size. Program 2 was an exception, but in such a highly recursive program it is difficult to imagine doing much better by using any algorithm. Hence using the even simpler algorithm of expanding the most deeply nested invocations first, which does not require use of run-time statistics, may be adequate for many programs. However, we believe there are a large number of practical programs for which our algorithm performs significantly better than this simpler algorithm, and it is easy to contrive example programs; so we believe the proposed solution is superior as a general-purpose algorithm.

3.3 Size

As the results show, there is a substantial space overhead involved with procedure invocations in MDL. There is also a significant space overhead associated with each procedure, and the reduction in the total number of procedures was an important factor in the size determination.

Program 2 was again the exception. Some attention was certainly paid to efficiency when the program was written, but more generally MDL programmers are well aware of the costs of procedures and procedure calls and design their programs accordingly. In this particular program there were few procedures besides the recursive ones, with no good way of avoiding the recursion.

The size approximation used was not a good indication of the actual size of compiled code, but, as we have stated, this had little impact on the final results. If even moderate increases in program size cannot be tolerated, a much more precise determination of size must be used, such as directly calculating the size of compiled code.

3.4 Execution Time

The average execution time saved per call was about 350 microseconds for Programs 1, 3, and 4(a), which agrees with the measured time for a normal procedure call in MDL. The average time saved for Program 2 was only about 83 microseconds. The main reason for this disparity is that a significantly faster call mechanism is used for recursive invocations than for nonrecursive invocations. However, the time for a recursive call is actually somewhat faster than this measured average. Inline substitution turned some recursive invocations into *non*recursive invocations, which had a canceling effect on the average time.

The average time saved for Program 4(b), 119 microseconds, is a little harder to explain. All of the testing was done on a time-sharing system with a paged memory system, and the time measurements include paging time. Further, primary memory is composed of memories of various speeds. These are not obstacles in timing most programs, but reliable timings are difficult for very large programs such as the CLU compiler. The size increase for this test apparently caused a significant increase in the amont of paging necessary.

It is possible (and standard practice for permanent MDL programs) to replace the slow calling sequence for nonrecursive invocations with one equivalent to that for recursive invocations. If this had been done for these programs, the reductions in execution time would have been more nearly the same and on the whole smaller. As the MDL compiler does no global optimization, we can only speculate about the benefits of inline substitution to other optimizations.

4. Conclusion

It appears that, in practice, the cost of structuring a program as a collection of many small procedures is not large, and in most cases this cost can be essentially eliminated by means of inline substitution. For structured programs with a low degree of recursion, the judicious use of inline substitution can eliminate almost all procedure calls with little or no increase in the size of compiled code. For recursive programs, a large number of calls can be eliminated, but at the expense of a rather substantial size increase. The execution time saved directly by inline substitution is small, even for fairly inefficient procedure call mechanisms; however, the enlarged context made available to other techniques may lead to much more optimization than would otherwise be possible. The algorithm we have presented seems sufficient to obtain these kinds of results in general, even though our model of control flow is probably not realistic. Further, we believe that any general algorithm for inline substitution must be at least as complex as the one presented if equivalent results are to be obtained.

Acknowledgments. I give my deepest thanks to Russ Atkinson for many suggestions and much advice and to Barbara Liskov for help with various portions of this work.

References

1. Aho, A.V. et al. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974, pp. 372-374.
2. Atkinson, R.R. Optimization techniques for a structured programming language. S.M. Th., M.I.T., Cambridge, Mass., 1976.
3. Cocke, J., and Schwartz, J.T. *Programming Languages and Their Compilers: Preliminary Notes.* New York U. Press, New York, 1970, pp. 480-494.
4. Hoare, C.A.R., Quicksort. *Comput. J. 5*, 1 (April 1962), 10-15.
5. Galley, S.W., and Pfister, G. *The MDL Primer and Manual.* SYS.11.01, Lab. Comptr. Sci. M.I.T., Cambridge, Mass., 1975.
6. Knuth, D.E. An empirical study of FORTRAN programs. *Software – Practice and Experience 1*, 2 (April-June 1971), 105-133.
7. Liskov, B.H., et al. Abstraction mechanisms in CLU. *Computation Structures Group Memo 144-1.* Lab. Comptr. Sci., M.I.T., Cambridge, Mass., Jan. 1977.
8. Liskov, B.H., and Zilles, S. Programming with abstract data types. Proc. ACM SIGPLAN Conf. on Very High Level Languages, *SIGPLAN Notices 9*, 4 (April 1974), 50-59.
9. Naur, P., Ed. Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6*, 1 (Jan. 1963), 11-13.
10. Palm, R.C. A portable optimizer for the language C. S.M. Th., M.I.T., Cambridge, Mass., 1975.
11. Pratt, T.W. *Programming Languages: Design and Implementation.* Prentice-Hall, Englewood Cliffs, N.J., 1975, pp. 147-151.
12. Rivest, R.L. Private communication.
13. Sahni, S. Approximate algorithms for the 0/1 knapsack problem. *J. ACM 22*, 1 (Jan. 1975), 115-124.
14. Schaeffer, M. *A Mathematical Theory of Global Program Optimization.* Prentice-Hall, Englewood Cliffs, N.J., 1973, pp. 150-155.
15. Schneck, P.B., and Angel, E. A FORTRAN to FORTRAN optimizing compiler. *Comput. J. 16*, 4 (Nov. 1973), 322-330.
16. Zelkowitz, M.V., and Bail, W.G. Optimization of structured programs. *Software – Practice and Experience 4*, 1 (Jan.-March 1974), 51-57.