1983

A Technique to Reduce Procedure Call Overhead in Block Structured Languages

Thomas P. Murtagh

Report Number:
83-453

https://docs.lib.purdue.edu/cstech/372

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
A Technique to Reduce Procedure Call Overhead in Block Structured Languages

by

Thomas P. Murtagh

Computer Science Department
Purdue University
West Lafayette, IN 47907

Abstract: The conventional storage allocation scheme for block structured languages requires the allocation of stack space and the building of a display with each procedure call. This report describes a technique for analyzing the call graph of a program in a block structured language that makes it possible to eliminate these operations from many call sequences, even in the presence of recursion.
1. Introduction

Current programming methodologies \(^4,11\) recommend that programs be decomposed into fairly small procedures that each perform simple abstract operations. This approach to program design makes the efficiency with which transfers can be made between procedures critical \(^7\). Unfortunately, a considerable amount of overhead is associated with procedure calls in current implementations of languages like Algol and Pascal, even on machine architectures designed to support such languages \(^5,6\). This report describes a new approach to memory management for procedure activation records that may significantly reduce this overhead.

The code generated for a procedure call by a typical translator of a language like Algol includes instructions to:

1. Save register values and other components of the caller's state,
2. Calculate the values of the actual parameters,
3. Allocate space on a stack for the local variables and parameters of the called procedure, and
4. Build a "display" of pointers to the stack segments containing non-local variables that may be referenced by the called procedure \(^2\).

Operations 3 and 4 are included in call sequences to support a language feature that is not used by all procedures -- the ability to make recursive calls. If none of the procedures in an Algol program are called recursively, a translator could allocate all variables for the program statically. This would make it possible to eliminate operations 3 and 4 from all call sequences for the program.

Even if some of the procedures in a program are called recursively, it is often possible to eliminate operations 3 and 4 from many of the call sequences generated by a translator. For example, suppose the main procedure in a program, \(P\), contains declarations for two procedures \(Q\) and \(R\) such that \(P\) may call \(Q\), \(Q\) may call \(R\), \(R\) may call \(P\), and no other calls are possible. Because all three of these procedures may be called recursively, this looks like an example in which dynamic allocation is required with each procedure call. If, however, a translator generates code to allocate the local variables for all three procedures each time \(P\) is called, then calls can be made to \(Q\) and \(R\) without allocation. This will reduce the execution time of the program, since the allocation of space for all three procedures takes no more time
than the allocation of space for P alone.

The significance of this example is that such optimizations of stack space allocation can be made automatically. Walter 8 refers to the property that makes such optimizations possible as "relative non recursiveness" and provides a method for recognizing cases of it based on the static nesting of procedures. It is possible, however, to do better by directly analyzing a program's call graph. The key is to recognize groups of procedures in which the task of allocating space for all of them can be shifted to just one of them. The next section presents a technique for recognizing such groups of procedures. Section three explains how to perform storage allocation using this information. Section four presents algorithms that make a further optimization when procedures are recognized to be absolutely nonrecursive. Sections five and six explain how these optimizations of stack space allocation affect the building of displays. Finally, section seven summarizes the techniques presented.

2. Intervals in a Call Graph

A call graph for a program P is a directed graph with one node for each procedure in P and an edge between two nodes Q and R if the procedure corresponding to Q may call the procedure corresponding to R. The node corresponding to the main program is called the initial node. We assume that all nodes in the call graph are reachable from the initial node. The word may in this description of a call graph is critical. It is impossible to determine whether one procedure will call another, even under very weak assumptions about the language involved 9,10. Therefore, rather than working with a precise call graph, we must work with an approximation. Weih 9 and Walter 8 present algorithms for computing such approximations.

An interval I in a directed graph G is defined to be a connected set of nodes in G such that:

1. There is a node h in I called the header which is contained in every path from a node not in I to a node in I.
2. I - {h} is acyclic

The notion of an interval was first used by Cocke and Allen 3 to recognize loops in program flow graphs, but it is also exactly what we need to recognize groups of relatively nonrecursive procedures in a call graph. If storage is allocated for all of the procedures in an interval each time the header is called, then
calls to all of the other procedures in the interval can be made without doing any storage allocation. Furthermore, a call graph can be partitioned into maximal intervals by a simple algorithm. One version of this algorithm is shown in Fig. 1. A more detailed version of the algorithm that runs in time proportional to the number of edges in the graph can be found in 1.

3. Memory Allocation for Intervals

When a call is made to the header of an interval, stack space should be allocated for all the procedures in the interval. This does not, however, mean that the amount of space allocated must equal the sum of the amounts required for the individual procedures. For example, if the subgraph for an interval takes the form shown below,

the space allocated for B can overlap that for C. In this case, the space allocated for the interval need only equal the sum of the size of A and the larger of B and C.

Let the size of a path in a call graph be the sum of the sizes of the activation records for the procedures on the path. Then, the space allocated for an interval need only be as large as the size of the largest acyclic path in the interval. This can be accomplished by allocating space for each procedure in the interval at a displacement equal to the size of the largest acyclic path from the header to the procedure. Thus, the displacement for the header is zero and the displacement for any other procedure is equal to the largest of the displacements at which the spaces allocated for its predecessors end. These displacements can be computed easily during the construction of the interval partition, because a node is added to an interval only after all of its predecessors have been added.
% Given a call graph G with initial node n0, this procedure % will set INTS equal to the set of sets of vertices % that form the maximal interval partition of the call % graph.

% % H is the set of vertices known to be the heads of intervals % that have not yet been considered.
% % DONE is the set of vertices that have been placed in intervals % % S[I] denotes the set of successors of nodes in I

INTS := {}  
H := \{n0\}  
DONE := {}  

while H \neq {} do  
  x := any element of H  
  call MAXI(x)  
  DONE := DONE U I  
  H := (H - \{x\}) U ( S[I] - I - DONE )  
  INTS := INTS U \{I\}  
end

where

procedure MAXI(h) begin  
  I := \{h\}  
  while there exists x in (S[I] - I) such that P(x) \subseteq I  
    do  
      I := I U \{x\}  
    od
end

Figure 1. The Allen-Cocke Interval Partition Algorithm.
The space allocated in this way is the minimum that must be allocated when the header is called. In some instances, however, some of it may be wasted. Consider the call graph in Fig. 2. A, B, C, D and E form one maximal interval. F alone forms another. Assuming that each procedure takes one unit of space, 4 units must be allocated for the first interval whenever A is called. If the path (A,C,D,E) is followed, then all of this space is actually required. If the path (A,B,E) is followed, however, only three units are actually used. Worse yet, if C calls F, only two units are actually used. Fortunately, in this last case, the unused units can be recovered by simply allocating space for F immediately after the space for C instead of at the end of the space for the interval. Thus, as a rule, space for a new interval should be allocated at the end of the space allocated for the procedure that called its header, rather than at the end of the space for the interval containing the caller.

4. Static Allocation

The call graph of a program is a representation of the relation "A may call B directly." The transitive closure of this relation is the relation "A may precede B in a sequence of calls." Using the transitive
closures, one can determine that a procedure is nonrecursive by asking whether "A may precede A in a sequence of calls." This information is stronger than the fact that a group of procedures forms an interval in the call graph. It implies that space for a procedure can be allocated statically.

In a conventional implementation of an Algol-like language, it would be advantageous to statically allocate activation records for all nonrecursive procedures detected in this way. With an interval based allocation scheme, static allocation of a procedure is only worthwhile if all of the procedures in the same interval are nonrecursive. This is because the main savings obtained through static allocation is the cost of stack space allocation, which is zero for all procedures except headers in the interval based scheme. The advantage of limiting static allocation to procedures that fall in intervals that are nonrecursive is that it is generally less expensive to determine which intervals are nonrecursive than to determine which procedures are nonrecursive. It can be done by computing the transitive closure of the derived call graph defined as follows:

1. For each interval in the call graph there is one node in the derived call graph.
2. If A and B are two intervals, there is an edge from A to B in the derived call graph if and only if there is a path from some node in A to the header of B in the call graph.

In most cases, the derived call graph will contain fewer nodes than the call graph. Therefore, computing its transitive closure will not be as expensive as computing the transitive closure of the full call graph.

When nonrecursive intervals are allocated statically, the space used need not equal the sum of the amounts of space needed for the individual intervals. Instead, space for many intervals can be overlapped by allocating each nonrecursive interval at a displacement equal to the maximum of the ending addresses of the storage segments allocated for procedures in nonrecursive intervals that precede its header. These displacements can be computed by first constructing a graph with:

1. one node for each procedure in a nonrecursive interval.
2. an edge from A to B if there is a path from A to B in G such that all vertices on the path except A and B fall in recursive intervals.

and then visiting the nodes of the graph in topological sort order setting each nodes allocation displacement equal to the maximum of the ending displacements of its predecessors.
In the case of a program in which all procedures are nonrecursive, this scheme will lead to the static allocation of space for all procedures. This is a satisfying result, since it implies that this scheme leads to the best possible allocation of storage in what is probably the most common case it will encounter. On the other hand, the savings obtained by static allocation in this case are insignificant. The use of intervals will have already eliminated all but one allocation of stack space in such a program.

Static allocation is more significant in the case of a program in which a nonrecursive procedure is called from recursive procedures in different intervals. For example, in a compiler using a recursive descent parser, many of the recursive routines in the parser would call some sort of 'get_lexeme' routine, which is likely to be nonrecursive. If the parser procedures that call 'get_lexeme' fall in distinct intervals, then 'get_lexeme' will be the header of a separate interval. Therefore, if nonrecursive intervals are not allocated statically, every call to 'get_lexeme' will involve stack space allocation. This overhead can be eliminated by using static allocation.

5. Display Sharing

The notion of an interval in a call graph also provides a way to reduce the overhead associated with display building. Recall that the display associated with a procedure, P, is just a sequence of pointers to instances of the procedures that statically surround P. The first elements of this sequence point to the outermost surrounding procedures. The last element points to the immediately surrounding procedure.

Now, suppose that the language being considered does not allow procedure variables or parameters of type procedure. Then, whenever a procedure is called, its display can be constructed from its caller's display be either a) adding a pointer to the caller if the procedure is defined within the body of the caller or b) removing zero or more pointers from the caller's display. This implies that the display associated with any procedure in an interval must consist of some prefix of the display associated with the interval's header followed by zero or more pointers to other procedures in the same interval. If storage for all of the procedures in an interval is allocated in one block, as suggested above, this makes it possible for all the procedures in an interval to share the display built for the header. To do this the compiler would handle non-local references in two distinct ways. If a reference would have been handled through a display
pointer that points to another element of the interval I, then the referenced variable can be found at a known displacement from the beginning of the stack segment allocated for the entire interval I. Thus, a single pointer to the beginning of the space for I can replace all such display pointers. All the other display pointers can be found in the display built for the header, which can also be found at a known displacement from the beginning of the stack segment for I.

To simplify the preceding explanation of display sharing, we have assumed that the language involved allowed neither procedure variables nor procedure parameters. We will continue to assume the absence of procedure variables, since the presence of both procedure variables and the ability to nest procedure definitions in a language would require the retention of activation records or some other mechanism to avoid dangling references. Procedure parameters, on the other hand, are found in two of the best known languages to which these techniques could otherwise be applied, Algol 60 and Pascal.

The problem with procedure parameters is that the display associated with a procedure called through such a parameter is determined at the time at which the parameter is bound rather than at the time the call is made. If a call is made in an interval using a procedure parameter that was bound outside of the interval, the display associated with the called procedure may contain pointers to procedures that are neither members of the interval nor pointed to by the header's display. This, unfortunately, means the called procedure and any procedures it calls cannot safely share the header's display.

This problem can be circumvented by refining the maximal interval partition of the call graph to obtain intervals in which the use of procedure parameters is appropriately restricted. We define a simply scoped interval to be an interval I with header h such that the set of formal parameter names that may be bound to any procedure in I - {h} is a subset of the formal parameter names declared by the procedures in I - {h}. The following lemma implies that the procedures in a simply scoped interval can all share the display built for the header if storage for all the procedures in the interval is allocated together.

Lemma: If I is a simply scoped interval in a call graph with header h, then the display associated with any instance of a procedure Q in I at runtime will consist of some prefix of the display associated with the most recently created active instance of h followed by zero or more pointers to the most recently created active instances of other procedures in I that fall on acyclic paths from h to Q.
The lemma can be proven by induction on the length of the longest acyclic path from the header to a particular procedure in I, which we call the distance from h to the procedure. The displays of all procedures at distance zero from the header clearly have the required property. Now, assume that the displays of all procedures at a distance less than n from the header have the desired property. If P is a procedure at distance n from the header, then any procedure Q that calls P must be at a distance less than n from h. Therefore, by assumption, the display of Q has the desired form. But, unless Q calls P through a formal parameter name, the display of P will be formed either by adding a pointer to Q to the display of Q or by removing a suffix from the display of Q. In both cases the resulting display will have the required form.

If a procedure calls P through a formal parameter name, the display associated with P is not determined at the point of call. It is determined at the points at which P is bound to formal parameters. So, consider all calls of the form

\[ R( \ldots P \ldots ) \]

The display to be associated with P when called through this formal parameter binding is formed from the display of the procedure Q in which the call to R occurs by either adding a pointer to Q's activation record or removing a suffix from Q's display. We will say that Q precedes P in I if Q precedes P on some acyclic path from h to P. If Q precedes P in I, then the resulting display must have the desired form, since the distance from h to Q must be less than n. If Q does not precede P in I, the form of the display associated with P does not matter, because the binding made can never be used to call P. This is because no parameter, X, defined by a procedure that does not precede P in I can be used to call P or to make a binding between P and any parameter, Y, defined in a procedure that precedes P in I. To call P using X, one would have to reference X in the caller. But the caller must precede P in I. This implies that its distance from h must be less than n and, by our assumption, that the only pointers to procedures in I on its display are pointers to procedures that precede P in I. Therefore, it cannot reference X or any other identifier defined in a procedure in I that does not precede P in I. Similarly, to make a binding between P and Y using X, X would have to be referenced by the procedure that called the procedure that
defined Y. But, such a procedure must precede P in L. Therefore, by the same argument as above, X cannot be referenced by any such procedure.

The intervals computed by the algorithm in Fig. 1 may not be simply scoped. Each of these intervals, however, can be partitioned into a set of simply scoped intervals. Fig. 3 shows an algorithm that simultaneously partitions a call graph into maximal intervals and partitions each maximal interval into simply scoped intervals. Given this partition, the procedures in a program are split into three classes: 1) headers of maximal intervals; 2) headers of simply scoped intervals and 3) members of intervals other than the headers. Calls made to the first type of procedure require both stack space allocation and display building. Calls made to the second require only display building. Calls made to all other procedures need neither.

The algorithm assumes that the set of procedure values that can be associated with any procedure parameter, pp, is known and can be referenced as val(pp). This is reasonable since an algorithm to determine the call graph must determine the values that can be associated with procedure parameters. It assumes that for each node x, count[x] is the in-degree of x in the call graph and rcount[x] is a count of the number of procedure parameters that may refer to x. After the algorithm is run, head[x] will point to the header of the maximal interval to which x belongs, rhead[x] will point to the header of the simply scoped subinterval to which x belongs. During the algorithm, head[x] is used to point to the header of the maximal interval in which the first successor of x was found, rhead[x] is used to point to the header of the simply scoped interval in which the first successor of x was found, and firstref[x] is used to point to the header of the simply scoped interval in which the first procedure that declares a procedure parameter that might be bound to x is found. The algorithm operates by placing each node in a doubly linked list of potential header of maximal intervals, H, as soon as any predecessor of the node is placed in an interval. If all of the predecessors of a node are placed in the same maximal interval, the node is removed from the list of headers and placed in a list of headers for simply scoped intervals or in the current simply scoped interval, depending on whether or not all of its predecessors and all of the procedures that declare parameters that might refer to it belong to the current simply scoped interval. This algorithm may not
head[s] := s
H := {s}
while H := {} do
   select and delete a node h from the front of H
   rH := {h}
   head[h] := h
   while rH := {} do
      select an delete a node rh from the front of rH
      place rh in add_to_rI
      rhead[rh] := rh
      while add_to_rI := {} do
         select and delete a node x from add_to_rI
         for each procedure parameter pp defined by x do
            for each y in val(pp) do
               if rhead[y] = 0 then
                  firstref[y] := rh
               fi
               count[y] := count[y] - 1
            od
         od
         for each (x,y)
            count[y] := count[y] - 1
            if head[y] = 0 then
               head[y] := h
               rhead[y] := rh
               add y to H and set where[y] equal to
               its position in H
            end
            if head[y] = h and count[y] = 0
            then
               remove y from H (at position where[y])
               if count[y] = 0 & rhead[y] = rh
               & firstref[y] = rh
               then
                  add y to add_to_rI
               else
                  add y to rH
               end if
            end
         end for
      end while
   end while
end while

Figure 3.
produce maximal simply scoped intervals when applied to programs that bind procedures to parameters that are never used. This does not seem important, however, since such bindings probably indicate a programming error.

6. Display Compression

In addition to eliminating the need for many displays, allocation by intervals allows the displays that remain to be compressed. In discussing the technique for sharing displays, we have already explained that a pointer to the beginning of the space for an interval that contains a given procedure is as good as a pointer to the procedure itself. Using this observation, the pointers to procedures usually found in displays can be replaced by pointers to intervals. If a procedure's display would have contained two or more pointers to procedures in the same simply scoped interval, they can all be replaced by the same interval pointer. In this case, the display can be compressed by replacing all pointers to procedures in a given simply scoped interval by one pointer to the interval (or to the maximal interval that contains it).

Similarly, all display pointers to procedures in nonrecursive intervals can be replaced by a single global pointer to the beginning of the area in which space for these procedures is statically allocated. In some cases, the elimination of such pointers will completely eliminate the need to build a display for an interval.

7. Summary

We have presented a technique for organizing the activation records of procedures in block structured languages that can significantly reduce the overhead associated with stack space allocation. In addition we have shown how this technique can lead to a reduction of the overhead involved in building displays in languages that allow nested procedure definitions.

We are confident that these techniques would be of significant value, because we believe that relatively few programs depend heavily upon recursion. Some experimentation is needed, however, to determine how useful these techniques are when applied to programs that are recursive.
References


