Register Allocation Via Clique Separators

Rajiv Gupta
Philips Laboratories
345 Scarborough Road
Briarcliff Manor, NY 10510

Mary Lou Soffa*
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Tim Steele
AT&T Bell Laboratories
75 Foundation Avenue
Ward Hill, MA 01830

Abstract - Although graph coloring is widely recognized as an effective technique for global register allocation, the overhead can be quite high, not only in execution time but also in memory, as the size of the interference graph needed in coloring can become quite large. In this paper, we present an algorithm based upon a result by R. Tarjan regarding the colorability of graphs which are decomposable using clique separators, that improves on the overhead of coloring. The algorithm first partitions program code into code segments using the notion of clique separators. The interference graphs for the code partitions are next constructed one at a time and colored independently. The colorings for the partitions are combined to obtain a register allocation for the program code. The technique presented is both efficient in space and time because the graph for only a single code segment needs to be constructed and colored at any given point in time. The partitioning of a graph using clique separators increases the likelihood of obtaining a coloring without spilling and hence an efficient allocation of registers for the program. For straight line code an optimal allocation for the entire program code can be obtained from optimal allocations for individual code segments. In the presence of branches, optimal allocation along one execution path and a near optimal allocation along alternative paths can be potentially obtained. Since the algorithm is highly efficient, it eliminates the need for a local register allocation phase.

Keywords - spans, interference graph, clique separators, graph coloring, spill code, node priorities.

1. Introduction

The problem of global register allocation is commonly formulated as a graph coloring problem in which an assignment of a color to each node in a graph is made such that no two nodes directly connected by an edge have the same color [2,1,4]. For register allocation, an interference graph is constructed in which the nodes correspond to candidates for registers and edges connect nodes that must be assigned different registers. A coloring of this graph is equivalent to an assignment of registers. Since graph coloring is an NP-complete problem[7], polynomial time heuristics are used to obtain suboptimal colorings. The interference graph for an entire procedure is constructed before register allocation is carried out. Since the coloring algorithms can be expensive both in time and memory demands, global register allocation is sometimes preceded by a local allocation phase[4]. However, this approach may not be successful as it has been found that many of the basic blocks in a program are small and so most of the allocation is done during the global allocation phase[8].

In this paper we present a strategy which can be used to improve both the space and time efficiency of global register allocation based upon graph coloring. As a result, local register allocation is no longer needed. The strategy is based upon a result due to Tarjan regarding the colorability of a graph containing clique separators[9]. A clique separator is a completely connected subgraph whose removal disconnects the graph into at least two subgraphs. These subgraphs may be further decomposable into smaller subgraphs using clique separators. If each subgraph is colored using at most k colors, then the entire graph can be colored using k colors by combining the colorings of the subgraphs. The subgraphs resulting from the decomposition of an interference graph correspond to code segments in a program. Thus, clique separators partition a program into code segments for which the register allocation can be performed independently. We show that the partitioning of the program can be carried out by examining the code; hence the technique does not require construction of the entire interference graph and running an algorithm to find the clique separators. The interference

* This work was partially supported by National Science Foundation under Grant CCR-8801104 to the University of Pittsburgh.

graph for the partitions can be constructed one at a time, and some coloring heuristic (e.g., priority based coloring) can be used to color a partition. The colorings for the partitions are combined, resulting in a coloring of the entire program code.

The allocation using clique separators can be carried out efficiently because at a given point in time, only an interference graph for a single partition needs to be constructed. This reduces the space requirements. Furthermore, if the run-time complexity of the coloring heuristic is a polynomial of a degree greater than one in the number of nodes in the graph, the time spent in coloring reduces with the number of partitions. The efficiency of the algorithm not only makes it possible to eliminate the local allocation phase but may also enable the use of more expensive coloring heuristics. Furthermore, the strategy is suitable for parallel implementations, as the subgraphs can be colored in parallel. The above strategy is not only efficient but may generate superior allocations, for it provides the additional flexibility of changing previously made register assignments. When a code segment is partitioned into two parts by a clique separator, the nodes that form the clique separator are included in the interference graphs for both of the parts. Coloring of each of the subgraphs independently may result in the assignment of different colors to the nodes from the clique. A coloring for the combined subgraphs is obtained by renaming the colors. The flexibility provided by the renaming of colors may enable us to color graphs which may not have been colored if the heuristic was run on the entire graph.

Consider a situation where a piece of straight line code has been partitioned using clique separators. An optimal solution for the entire segment can be constructed from optimal solutions for each of the components. If the code contains branches, optimal allocation can potentially be achieved for one of the paths. As done in techniques such as trace scheduling[6], this path can be chosen to be the one that is most likely to be executed. Near optimal allocations may be obtained for the other paths. The allocation of the alternative paths can be done independently until they merge with a path for which allocation has already been carried out. At this point code for moving values among the registers may have to be generated.

The register allocation algorithm developed by Chi and Dietz[3] uses the notion of register cut points to divide a program into parts for which allocation can be carried out independently. A register cut point in a program is a point at which the optimal allocation of live variables to registers can be determined without actually applying a register allocation algorithm. For example, if there is a single variable live at a point in the program, then one of the registers will hold its value and the remaining must be empty. The notion of clique separators is much more general than register cut points. A code segment can be divided into subparts using clique separators even if optimal allocation at any point during the code segment is not known. Furthermore, the register allocation scheme proposed by Chi and Dietz applies to straight line code, for which register allocation is not NP-complete, and is expensive as it has a running time of $O(m^k n)$ where m is the number live variables at a point in the program, k is the number of registers available and n is the number of references to variable values.

In subsequent sections we first summarize Tarjan's result and define the representation of live ranges that is used in constructing an interference graph. An algorithm for partitioning straight line code by clique separators is presented. Next we show how partitioning is done in the presence of branches. When spilling is required, we describe how the colorings of partitions are combined to obtain the overall register allocation. An algorithm that uses separators and priority based coloring is given. The performance of this global register allocation algorithm is analyzed in terms of space and time complexity and the quality of register allocation. The issue of using heuristics for artificially creating partitions is discussed. These heuristics can be used if the clique separators found do not result in sufficiently small partitions. The implementation of the technique is briefly described. The results based on experimental studies are summarized.

2. Background

In this section we summarize Tarjan's result on partitioning a graph using clique separators and obtaining a coloring for the entire graph from the colorings of the subgraphs.

A clique separator is a completely connected subgraph whose removal disconnects the graph. For the graph shown in Fig. 1(i) the clique $CS = \{v_1, v_2, v_3\}$ is a separator as its removal results in disconnected subgraphs $S_1 = \{v_4, v_5\}$ and $S_2 = \{v_6, v_7, v_8, v_9\}$. The subgraphs which must be colored using k colors, if a k-coloring for the entire graph is to be found, are shown in Fig. 1(iii). These subgraphs are formed by including the members of the clique separator in each of the disconnected subgraphs (S_1 and S_2) shown in Fig. 1(ii). In Fig. 1(iii) colorings of the subgraphs using three colors are shown. These colorings are combined to obtain a 3-coloring for the entire graph shown in Fig. 1(iv). The combining process involves renaming of colors in one of the subgraphs so that both subgraphs use the same colors for the members of the clique. In this example the coloring was achieved by interchanging the use of colors c_1 and c_2 in the subgraph that includes S_1 and CS. The subgraphs resulting from a decomposition may be further decomposable using clique separators. A graph which cannot be decomposed any further is called an atom. In the above example S_1 is an atom and S_2 is not an atom, as it can be further decomposed by clique $\{v_6, v_7\}$.

The algorithm developed by Tarjan requires construction of the entire graph, following which the separators are identified and the graph decomposed. This approach is not useful for register allocation because it does not reduce the space complexity of the algorithm. In order to avoid this problem, clique separators in this paper are identified by examining the code instead of the interference graph of the entire code sequence.

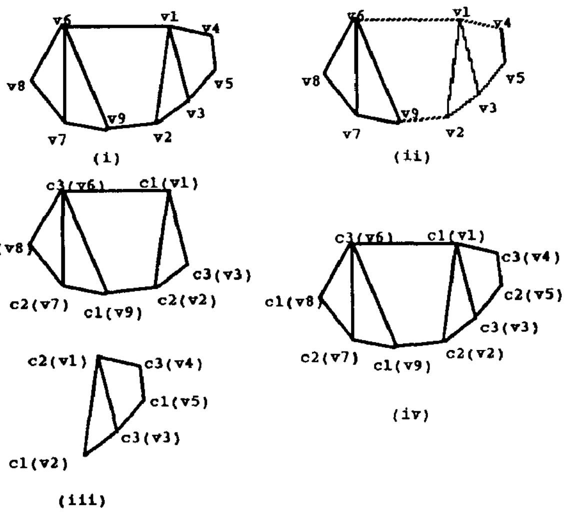


Fig. 1. Clique Separators

3. Live Ranges (Spans) and Interference Graphs

In earlier global register allocation algorithms, global register allocation is sometimes preceded by local register allocation. Thus, during the global phase, a live range of a variable is defined as an isolated group of contiguous basic blocks in the control flow graph in which the variable is defined and referenced. In this work, since there is no local allocation phase, the live ranges are defined as an isolated group of contiguous code statements (instead of basic blocks) in which the variable is defined and referenced. Thus, a variable may have multiple live ranges in a single basic block.

The nodes in an interference graph for a code segment correspond to spans or live ranges of variables. In straight line code, a span or a live range is a code sequence at the beginning of which a variable is defined and at the end of which this definition is last referenced. Thus, each definition of a variable gives rise to a span in the interference graph. Fig. 2(i) shows the spans resulting from definitions of variables. The spans that overlap (represent values that are simultaneously live at some program point) cannot be assigned the same register and thus are connected by an edge in the interference graph. For the example in Fig. 2(i), spans L_1 and L_2 are connected by an edge as they overlap. The interference graph is colored using register names as colors. Two spans are considered to overlap if the values they represent can be simultaneously live at some point in the program. If a span ends at an instruction and another starts at the same instruction, the two spans are not considered to overlap as the same register can be allocated to both.

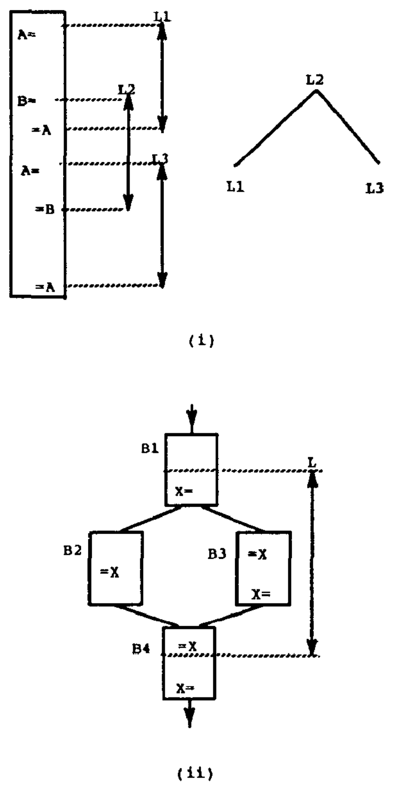


Fig. 2. Spans and Interference Graph

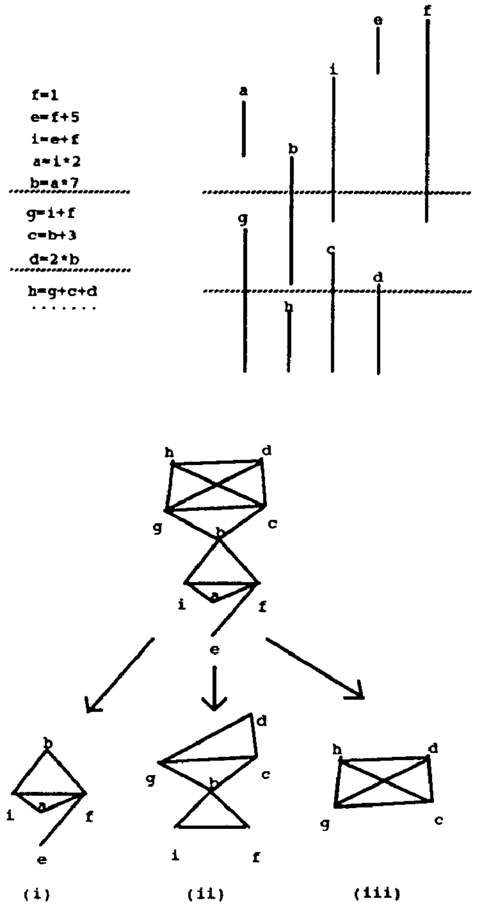
In code with branches, a span is the part of the code in the control flow graph in which a variable is defined and referenced and no other definition of the variable reaches a reference inside the span. The example in Fig. 2(ii) shows a live range of variable X that starts in basic block B_1 and ends in basic block B_4.

4. Partitioning Straight Line Code

We consider the problem of partitioning straight line code into code segments for which register allocation can be carried out independently. The interference graphs for the code segments are the same as the subgraphs that would result upon decomposition of the interference graph for the entire code segment using *certain* clique separators. After the register allocation for each code segment has been carried out, the register allocation for the entire code segment can be found by renaming the registers assigned to the spans that appear in more than one interference graph. Tarjan's result guarantees that if each code segment has been allocated at most k registers then the overall allocation will require at most k registers.

In Fig. 3 we show a set of overlapping spans for a sample piece of code. At any point in time, there are several overlapping spans, represented by nodes of a clique in the interference graph. Each of these cliques is a separator because its removal from the interference graph results in subgraphs consisting of spans that end before the clique and spans that start after the clique. Furthermore, these subgraphs are not connected by an edge as the live ranges from these subgraphs do not overlap. The nodes forming a clique separator are included in both the subgraphs into which it divides the interference graph. If we divide the code at each of the possible separators, each resulting partition will contain a single statement. The interference graph corresponding to a partition will contain all values live at that point. Since a span can appear in any number of these subgraphs, this partitioning of the subgraph into subparts does not result in proportionally smaller subgraphs. As a result, partitioning using all clique separators will make register allocation more expensive.

The above problem can be avoided by chosing the cliques carefully. The maximum number of cliques, chosen as separators, in which a span can occur can be fixed to a small constant (say c). Thus, the maximum number of subgraphs in which a span can occur is $c+1$. If the entire graph containing n vertices is divided into m subgraphs, then each subgraph on an average will contain $(c+1)n/m$ nodes. Assuming m is large, the subgraphs will be significantly smaller than the interference graph for the entire program.



```
f=1
e=f+5
i=e+f
a=i*2
b=a*7
g=i+f
c=b+3
d=2*b
h=g+c+d
.......
```

Fig. 3. Clique Separators

In this section we present an algorithm that choses separators such that no span is present in more than two of the chosen clique separators (i.e., $c = 2$). The example presented in Fig. 3 contains two such separators, consisting of spans {b,i,f} and {g,c,d}. These separators divide the code segment into three parts; hence the interference graph is divided into the three subgraphs shown in Fig. 3. The interference graph for a single code segment represents only the information regarding the spans that are live during that period. Thus, the spans that end before the code segment and the spans that begin after the code segment are excluded from the graph. The members of the separator are included in both of the subgraphs that result from it. In the example shown {b,i,f} is included in the first and second subgraph while {g,c,d} is included in the second and third subgraph. The subgraphs are colored separately and their colorings combined to obtain the coloring for the entire graph.

```
Partition
  {
   $PRE = POST = CLIQUE = \phi$
   while there are more instructions to scan
     {
      Let $s_{start}$ denote the span that starts at the
        current instruction
         $CLIQUE = CLIQUE \cup \{s_{start}\}$
         $POST = POST - \{s_{start}\}$
         $\forall$ spans $s_i$, st $s_i$ has not yet started
             and $s_{start}$ overlaps $s_i$
         $POST = POST \cup \{s_i\}$
      $\forall$ spans $s_{end}$ that end at the current instruction
        {
         $PRE = PRE \cup \{s_{end}\}$
         $CLIQUE = CLIQUE - \{s_{end}\}$
        }
      $\forall$ spans $s_i$ that no longer overlap
         a member of $CLIQUE$
           $PRE = PRE - \{s_i\}$
      Check ( $CLIQUE$ )
     }
  }

Check ( $CLIQUE$ )
  {
   If $CLIQUE$ can be partitioned into disjoint sets
      $CLIQUE_{PRE}$ and $CLIQUE_{POST}$
   such that
      $PRE \neq \phi$; $POST \neq \phi$;
      no span from $PRE$ overlaps a span from $CLIQUE_{POST}$
      no span from $POST$ overlaps a span from $CLIQUE_{PRE}$
   then set $CLIQUE$ is chosen as a separator
  }
```

Fig. 4. Finding Separators

In order to identify the separators, we scan the code from beginning to end, constructing and updating three sets, namely *PRE*, *POST*, and *CLIQUE*. By examining the sets we can determine whether the clique at that point in the program should be chosen as a separator or not. The set *CLIQUE* contains the members of the current clique, the set *PRE* contains the spans that have already ended but overlap one of the members of *CLIQUE*, and the set *POST* contains the spans that have not yet begun but overlap members of the set *CLIQUE*. Thus, in the example above, at the point at which the separator {b,i,f} occurs, the three sets contain the following: *PRE*={a,e}, *POST*={g,c} and *CLIQUE*={b,i,f}. The clique separator formed by members of *CLIQUE* is chosen if and only if it can be divided into disjoint sets $CLIQUE_{PRE}$ and $CLIQUE_{POST}$, such that spans from *PRE* do not overlap spans from $CLIQUE_{POST}$, spans from *POST* do not overlap spans from $CLIQUE_{PRE}$ and the sets *PRE* and *POST* are non-empty. For the clique {b,i,f}, the set $CLIQUE_{PRE}$ is {i,f} and the set $CLIQUE_{POST}$ is {b}. The above condition ensures that no span appears in more than two consecutive separators. Furthermore, in chosing a separator, sets *PRE* or *POST* are non-empty to ensure that the interference graph for a code segment contains at least one node that is not present in the subgraphs preceding and succeeding it. The algorithm that constructs the sets and checks for separators is summarized in Fig. 4. The partitioning of code shown in Fig. 3 was generated using this algorithm.

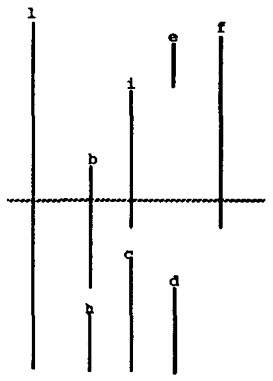


Fig. 5. Choosing Additional Separators

If a live range, *l*, extends along a long sequence of code, no clique separators will be selected by the algorithm in Fig. 4. Live range *l* will belong to every *CLIQUE* set making it impossible to construct sets $CLIQUE_{PRE}$ and $CLIQUE_{POST}$ that satisfy the conditions necessary for selecting a clique. In this situation, if the live range *l* does not already belong to $(c+1)$ partitions, a clique separator that divides the sequence of code and satisfies the criteria that no live range belongs to more than $(c+1)$ partitions may exist. This is demonstrated by the example in Fig. 5. After the algorithm in Fig. 4 has been applied, any remaining long sequences of code can be further partitioned in the above manner. The condition for choosing such separators is as follows:

```
Check ( CLIQUE )
  {
  If CLIQUE can be partitioned into sets
      $CLIQUE_{PRE}$, $CLIQUE_{POST}$, and REM
  such that
      no span from PRE overlaps a span from $CLIQUE_{POST}$
      no span from POST overlaps a span from $CLIQUE_{PRE}$
      no span from REM belongs to 2 chosen separators
  then set CLIQUE is chosen as a separator
  }
```

5. Code with Branching

Next we show how separators can be located and register allocation carried out in the presence of branches. Consider the control flow graph shown in Fig. 6(i) which has both convergence and divergence of flow. We start by locating separator S_1 in basic block B_1. If no more separators are found in B_1, we identify the separators along the paths from basic block B_1 to B_2 and B_3 independently. As shown in Fig. 6(i), these basic blocks may not contain separators and the next separator may be in basic block B_4 after the two paths have merged. Separators S_1 and S_2 partition the program into three parts P_1, P_2 and P_3, and register allocation for these is performed one at a time.

For the same flow graph, Fig. 6(ii) shows the situation in which the basic blocks B_2 and B_3 contain separators S_2 and S_3. These are found by examining the two paths independently for separators. Upon convergence of flow, a clique in the block immediately following the merge may be a separator for one of the paths and not for the other. The path more likely to be executed can be chosen and the separator along that path can be found. In this example the graph has been partitioned into five parts. The partition P_2 contains code from basic blocks B_1, B_2 and B_3. The reason for constructing a combined graph is to ensure that both B_2 and B_3 use the same registers for the values being passed from B_1. Interference graphs for all five of the regions are constructed and allocation is performed independently. Thus, spans common among adjacent partitions may have been assigned different registers. Renaming of colors assigned to members of the clique separator S_2 during the coloring of partition P_2 is carried out so that they are the same as the colors that were used during coloring of partition P_1. Similarly, renaming is applied to partition P_3 to match the colors it used for members of S_2 with the colors that were used by P_2. At this point the colors used in partition P_4 have to be renamed to match the colors used in P_2 and P_3. Such a renaming may not exist. Thus, copy code that moves the values into appropriate registers is inserted along the path from B_3 to B_4. The allocation can be done in such a manner that the copy code is inserted along a path that is less likely to be executed. Optimal colorings for individual regions will result in optimal allocation along one path. The allocation along the other path is near optimal as the usage of registers is optimal except for the copying code that has been inserted.

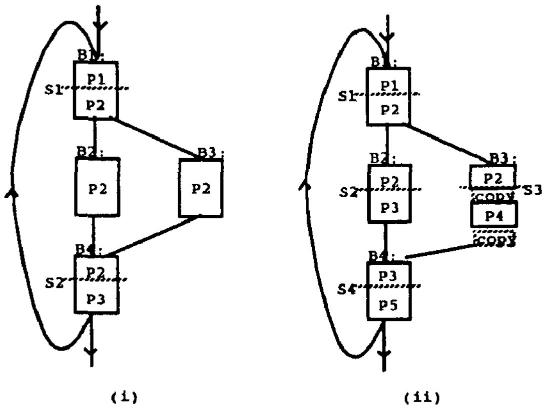


Fig. 6. Separators in Presence of Branching

When performing renaming of colors for partition P_4 in the example in Fig. 6(ii), an attempt is made to assign registers to the span in a manner that does not require introduction of copying code. For example, if priority based coloring is used, copy code is only introduced if the register required to eliminate the copy code has been allocated to a span with higher priority. If this is not the case, the register is freed and assigned to the span with higher priority. The free registers are assigned to spans with lower priority, and values corresponding to spans with lower priority are spilled if needed.

6. Combining Colorings in Presence of Spilling/Splitting

In the previous sections we discussed combining of colorings obtained for individual partitions, so as to obtain an overall coloring. The spans that form a clique separator are present in the interference graphs of the code partition preceding and succeeding the clique. Thus, they may be assigned different registers. For straight line code, Tarjan's result allows renaming of registers in one of the segments so that the same registers are used in both code segments. In the presence of branches, the partitions preceding and succeeding a code segment may already have been assigned registers that cannot be renamed. In this situation, code to transfer values from one register to another may have to be introduced.

In the discussion above, the issue of spilling the values into memory and the splitting of spans was not considered. If the number of registers available is less than the number of live values, then the register allocation algorithm must choose the values to be held in registers and spill the remaining values into memory. A live range must be split into subranges which are assigned different registers, and code to transfer the value from one register to another must be introduced.

Next we show that the combining of colorings for partitions can be carried out even in presence of spilling and splitting of spans that are members of clique separators. We assume a priority based heuristic to determine the spans that should be spilled. Let us consider a situation in which during coloring of one subgraph, a span from the clique is split, while it is not split when the other subgraph is colored. A span split during coloring of any one of the subgraphs is treated as being split for both the subgraphs. Splitting a span in one subgraph does not disrupt the coloring in the other subgraph. This can be explained by considering the following cases:

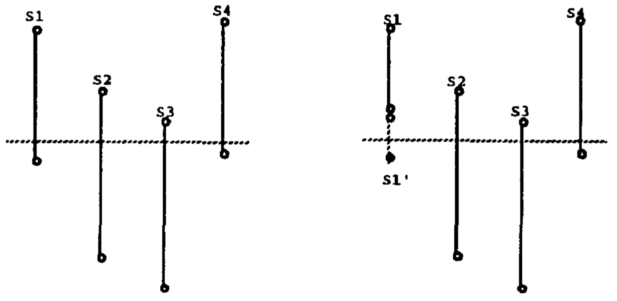


Fig. 7(i). Combining Colorings in Presence of Spilling

Case (i): Consider the situation in which a span from the clique is split because the size of the clique is greater than the number of registers available. In this case, the part of the span spilled will be the same irrespective of the subgraph chosen for coloring first. This is because the priorities of the spans are used to decide upon the span that should be spilled. As a result the spilling of a member of the clique does not effect the colorings of the individual partitions. In the example shown below, if we assume that there are three registers available and span S_1 has the lowest priority, then coloring of code partitions preceding and succeeding the separator require that span S_1 be split in the manner shown.

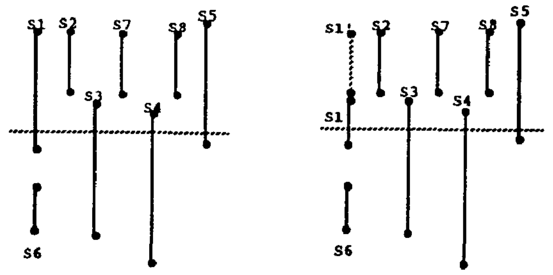


Fig. 7(ii). Combining Colorings in Presence of Spilling

Case (ii): Now let us consider the case in which there are enough registers available to assign to all spans in the clique. If a span from the clique is split while coloring the first subgraph, then it must be the case that a span from set *PRE* with higher priority was assigned that register. We know that spans from *POST* cannot overlap spans from *PRE*. Thus, the span overlapping a member of *PRE* created by splitting a member of the clique, can be ignored during the coloring of the subgraph formed by the *POST* set. Using a similar argument, it can be shown that if a span from the clique is split while the second subgraph is being colored, it will not require recoloring of the first subgraph.

The example in Fig. 7(ii) demonstrates this case. Let us assume that there are four registers available and that span S_1 has the lowest priority. When the partition preceding the separator is colored, part of span S_1 is spilled. The part of the span spilled does not overlap any other spans in the code partition following the separator. As a result no changes are required for the coloring that may have been performed for the second partition. In this example span S_1' created by splitting S_1 was spilled. However, there are cases in which after splitting S_1' may be assigned a register. In either case this does not effect the coloring of partition following the separator.

7. Computing Node Priorities

The priority of a node or live range is measured in terms of the savings in execution time that are incurred by assigning a register to it. These priorities are maintained to guide the coloring of the interference graph. Since, during coloring spans can be split, the priorities of spans are updated and that of newly created spans computed. Let *LODSAV* be the savings in execution time for each reference to a variable in a register instead of memory, *STRSAV* be the savings for each definition of a variable assigned to a register as compared to storing the value in memory, and *MOVCOST* be the cost of moving a value between a register and memory. The net savings in execution time, *NETSAV*, if a live range is assigned to a register, is given by:

$$NETSAV = LODSAV^{*}\text{u} + STRSAV^{*}\text{d} - MOVCOST^{*}\text{n}$$

where u is the number of uses, d is the number of definitions, and n is the number of loads and stores that must be performed to ensure that the value is in the register during the period a register is allocated to it and in memory otherwise.

In order to compute the value of n we must determine whether the value of the variable is to be loaded into the register, or whether it is already in the register at the start of the live range. Similarly if the value is live at the end of the live range assigned to a register, the value must be saved in memory. In addition, the separators also must be examined to determine whether there is a possibility of copy code being generated. For example, consider a variable that has definitions in basic block B_2 and B_3 of Fig. 6 and these definitions are live in basic block B_4. If the choice of separators as shown in Fig. 6(i) is made, no copy code may be needed. On the other hand if the separators in Fig. 6(ii) are chosen, copy code may be needed. Thus, additional operations are needed to move the values among the registers. It should be noted that the copy code can only be introduced if the paths merge.

The live range of a variable can extend over several basic blocks. A variable referenced inside a loop body is likely to be referenced more often, and hence assigning a register to such a variable is likely to result in greater savings. Thus, the total savings resulting from assigning a register to a live range is normalized with respect to the loop-nesting depth of the basic blocks in which they are referenced as follows:

$$TOTALSAV = \sum_{i \in lr}(NETSAV_i \times w_i)$$

where w_i is the loop-nesting depth of basic block i in the flow graph and $NETSAV_i$ is the savings accrued by assigning the value to a register in basic block i.

8. Algorithm for Allocating Registers

An overall algorithm for global register allocation using clique separators and based upon priorities, is summarized in Fig. 8. In this algorithm, the program is partitioned into code segments and an interference graph for a single partition is constructed and colored. The node priorities are maintained globally and recomputed as spans are split in a manner similar to priority based coloring[4]. During coloring of a partition, only priorities of those nodes that belong to the current partition are needed. Thus, while choosing a node to be colored, only the priorities of nodes in the current graph need to be examined. This reduces the running time of the algorithm. One by one the subgraphs are constructed and colored. After a subgraph has been colored, its colors are renamed to match the colors of the adjacent partitions that have already been colored. Copy code is introduced if needed. The nodes in the graph which have fewer neighbors than the number of registers are colored last as they can be colored no matter what colors are assigned to their neighbors. These nodes are referred to as the unconstrained set of nodes and the remaining nodes are called the constrained set of nodes in the algorithm. Over-allocation of registers is prevented by assigning registers to only those spans for which *TOTALSAV* is positive.

```
Allocate
 {
  Partition the program by identifying the separators
  Determine the order for processing the partitions

  loop
   {
     Construct the interference graph for the partition to
     be processed next.

     Put the live ranges whose number of neighbors is less
     than the number of registers in the unconstrained set.
     The rest of the nodes are put in the constrained set
     of nodes.

     Repeat the following steps till all constrained nodes
     have been processed.
     {
       For each live range lr
       {
         If lr has less colored neighbors than the total number
         of colors then assuming that it can be assigned
         a register compute TOTALSAV.

         If lr has more colored neighbors than the total
         number of colors then split the live range lr.
         Construct lr_1 such that it is the largest part of lr
         that can be colored. Update the graph and sets
         unconstrained and constrained. Recompute
         TOTALSAV for the nodes added to constrained pool.
       }
       Choose the live range from the constrained pool with
       highest TOTALSAV and assign a color to it. If several
       live ranges have same priority choose the one which is
       the shortest.
     }
     Assign colors to unconstrained nodes. The nodes are
     assigned colors if doing so results in savings.

     Combine the results with already processed partitions.
     This involves renaming of colors for the current partition
     and possibly generation of copy code.
   }
  until all partitions have been processed
 }
```

Fig. 8. Priority Based Register Allocation

In the algorithm presented the priority of a node, which is the value *TOTALSAV*, is not normalized by the length of the live range. In the algorithm developed by Chow and Hennessy, the priority is normalized by the length of the live range, because the global allocation phase is preceded by the local register allocation phase. During global allocation the unallocated variables have occurrence frequencies that do not differ greatly, as the local allocation is based upon the occurrence frequencies of variables. The adjustment of the priority by the live range length is needed as a longer range occupies the register for a longer period of time. However, in the above algorithm there is no local allocation phase and hence the priorities are not normalized. If several live spans have the same priority the shortest span is colored first.

9. Performance

The strategy presented makes the coloring process efficient which justifies the elimination of the local register allocation phase. This also opens up possibilities for using more expensive heuristics that may result in improved performance. In the analysis below it is assumed that the number of live ranges (n) remains constant although splitting of live ranges increases the number of spans. This assumption greatly simplifies the analysis.

Space Complexity: The space complexity of the modified priority based coloring algorithm is $O(n^2/m^2)$, where n is the number of live ranges and m is the number of partitions into which the program is divided.

Proof: The space complexity of the coloring heuristic when applied to an x node interference graph is $O(x^2)$, as there can be at most $x(x-1)$ edges in the graph. Since only the interference graph for a single code partition, consisting of $O(n/m)$ nodes, is constructed at any given point in time, the space required by the algorithm is $O(n^2/m^2)$. □

Run-time Complexity: The run-time complexity of the modified priority based coloring algorithm is $O(n^2/m)$, where n is the number of spans and m is the number of partitions into which the program is divided.

Proof: The run-time complexity of the coloring heuristic when applied to an interference graph with x nodes is $O(x^2)$, for in each iteration of the loop, one live range is chosen, and we may have to perform x iterations. The time complexity of processing a single code partition is $O(n^2/m^2)$ as its interference graph contains $O(n/m)$ live ranges. Since there are m partitions to process, the run-time complexity of the modified priority based coloring algorithm is $O(n^2/m)$. □

The computation of live ranges entails additional overhead that has not been considered in the above analysis. In the modified priority based register allocation algorithm presented in this paper, the live ranges must be computed to locate the separators. However, in Chow and Hennessy's algorithm some of the overhead in computing the live ranges is avoided by assuming that all uses and definitions of a variable are part of a single span. A span is split into smaller live ranges only if it cannot be colored.

The partitioning of a graph into smaller subgraphs increases the likelihood of finding a coloring. This can be demonstrated using the simple example given below. Let us assume that three registers are to be assigned to A, B, C and D for which the interference graph is shown in Fig. 9. It is possible that the heuristic may color A with R_1 and B with R_2 in which case the graph cannot be colored. On the other hand if the graph is split into two subgraphs using the clique separator {C,D} the resulting subgraphs can be colored using three colors. The colorings of the subgraphs can be combined to obtain a coloring for the entire graph as shown in Fig. 9(ii).

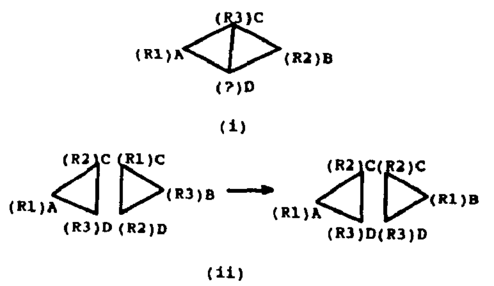


Fig. 9. Reassigning Registers to Subgraphs

From the above example it is clear that the cause for improved performance is the flexibility provided by the ability to change register assignments when the assignments for individual subgraphs are combined to obtain the assignment for the entire graph. The following result shows that optimality is preserved when the results are combined.

Claim: An optimal solution can be constructed from optimal solutions for individual partitions in straight line code.

Proof: This results follows quite easily from Tarjan's result which states that if colorings for each of the subgraphs can be carried out using k-colors, the entire graph can be colored in k colors. The coloring for the combination of two subgraphs is obtained by changing the coloring of one of the subgraphs. This is to make sure that the members of the clique are assigned the same color in both the subgraphs. Changing the names of the colors does not change the number of colors used to color a subgraph; therefore the optimality of the solutions is preserved when they are combined. □

The above results show the advantages of using the strategy based upon clique separators. The following result shows that the strategy described is not likely to result in solutions worse than the solutions generated by the Chow and Hennessy's algorithm. In other words, applying the coloring heuristic to the subgraphs is not likely to result in worse solutions than an algorithm that constructs the entire interference graph before coloring it.

Claim: The strategy based upon clique separators will never split or spill a live range with higher priority in order to avoid splitting or spilling of another node with lower priority.

Proof: In order to prove the above result we consider the following cases:

Straight Line Code: When a partition from a straight line code is being processed the nodes are colored in decreasing order of priority. The inability to color a node is not due to nodes with lower priority from the same partition, because these nodes have not yet been colored. However, nodes with lower priority from other partitions may have already been colored. A low priority node from another partition cannot be the cause for the inability to color a node with higher priority because the coloring of nodes in different partitions is done independently of each other. Thus, if a live range is spilled or split, it could not have been avoided by splitting or spilling nodes with priority lower than this live range.

Code with Branches: In code containing branches, the colorings of partitions found independently cannot always be combined so that they use the same colors for the spans that are shared by more than one partition. Copying code is introduced to ensure that values are in appropriate registers. Introduction of copy code is equivalent to splitting of live ranges. In order to prove our claim we must show that a span with higher priority is not split in preference to a span with lower range. However, as explained in section 5, when combining colorings an attempt is made to use the same colors for spans with higher priority. □

10. Creating Additional Partitions

The shorter the spans the more likely it is that the code will contain separators. If the spans in the set *CLIQUE* are long, they are likely to overlap spans from both *PRE* and *POST*. Thus, it may not be possible to partition *CLIQUE* into sets $CLIQUE_{PRE}$ and $CLIQUE_{POST}$. If enough partitions do not exist, clique separators can be created using the following approaches.

Renaming

Long spans are created by global variables that are used throughout the computation. In order to create shorter spans the definition of the spans can be modified as follows. A span starts when a variable is given a new definition and ends at the point where that definition is last used. This definition for a span differs from the definition used by Chow and Hennessy. In their work a span could contain multiple definitions of the same variable. The modified definition will result in shorter spans which increases the likelihood of finding separators in the interference graph. If the renaming optimization has already been performed, the code will automatically yield shorter spans with single definitions[5].

Heuristics

Live ranges that extend through long sequences of code can also result due to variables that are defined once but used throughout the program. Such live ranges cannot be split by renaming. In order to increase the likelihood of finding separators, such live ranges should be identified and handled in the following manner:

(i) The live range can be assigned a register and removed from consideration during the rest of the allocation process. Live ranges due to variables that are frequently referenced throughout the program should be treated in this manner. (See Figure 10i.)

(ii) The live range can be removed from consideration during the register allocation process. After register allocation any unassigned registers can be allocated to this live range. The live ranges that are referenced infrequently should be processed in this manner. (See Figure 10i.)

(iii) The live range can be made shorter by removing sequences of basic blocks in which the variable is not referenced. This will result in shorter multiple live ranges. The live ranges that do not fall in the first two categories can be processed in this manner. (See Figure 10ii.)

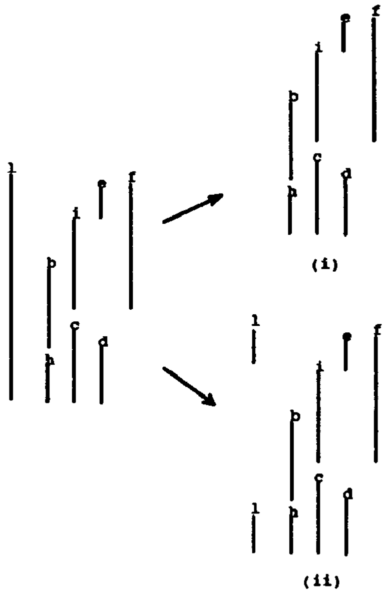


Fig. 10. Creating Additional Separators

The solutions described above require examining the live ranges and splitting them if enough separators are not chosen by the algorithms discussed. The next two solutions are simpler and may be used as the separators are being examined. After the current partition becomes larger than some specified size, and no separator has yet been found, the following approaches can be used to create a separator.

Giving up Efficiency

As mentioned in section 4, code along an execution path has a separator at the end of each statement. However, for efficiency reasons separators are chosen in a manner that ensures that no span belongs to more than two chosen clique separators along an execution path (i.e., $c=2$). This limits the number of subgraphs in which a span can appear and hence the size of the interference graph for each partition. The partitions which are large can be further subdivided by relaxing the constraint on the value of c. The spans can be allowed to appear in a larger number of chosen clique separators until the subgraphs obtained are sufficiently small. Since not all the live ranges appear in two clique separators, it is acceptable for a few of them appear in more than two clique separators.

Splitting the Spans

If a long segment of code does not have separators, they can be created by splitting the spans. Splitting of a span into multiple spans allows assignment of different registers to the newly created spans as well as spilling some of the spans.

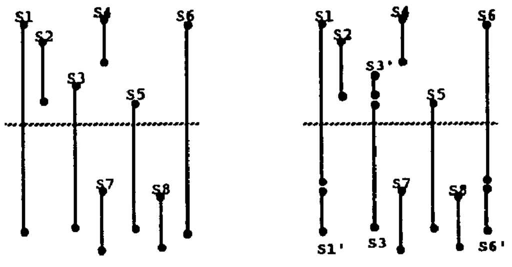


Fig. 11. Splitting Spans to Create Separators

The example in Fig. 11 shows how a clique can be turned into a separator by splitting the spans. If we consider the sets $CLIQUE=\{S_1,S_3,S_5,S_6\}$, $PRE=\{S_2,S_4\}$ and $POST=\{S_7,S_8\}$, we observe the clique is not a separator. However, if spans S_1, S_3 and S_6 are split resulting in sets $CLIQUE=\{S_1,S_3,S_5,S_6\}$, $PRE=\{S_2,S_4,S_3'\}$ and $POST=\{S_1',S_7,S_8,S_6'\}$, then the clique does form a separator. Heuristics can be used to decide which span to split and where to split it. For example, the spans with lower priority can be chosen for splitting and, if possible, a span can be split into two parts such that one of them uses the value scarcely and another uses the value heavily. At the points where a span is split, instructions to load the value into a register or store it into memory may have to be inserted.

11. Implementation

The algorithms described in this paper have been implemented as part of a compiler for a subset of Ada. There are two main differences between the implementation and the manner in which the algorithms were described in this paper. Firstly, the interference graphs for the code partitions are constructed and colored as the separators are detected instead of finding all the separators before starting the allocation. Secondly, the register assignments performed in one partition are propagated to other partitions that contain the same live ranges but have not been colored yet. This eliminates the need for an explicit phase that combines the colorings for individual partitions by renaming colors.

Studies are currently being conducted to experimentally compare the performance of the clique separator approach to register allocation with the approach of coloring the entire graph. Results of experiments to investigate the space efficiency of the approach for a sample of small programs are given in the following table. The programs considered included a money changer, integer matrix multiplication, sieve, bubble sort and towers of hanoi program. The programs, with the exception of the towers program, contained one procedure. The towers program had two procedures, the results of which are presented separately. In the study, no registers were spilled, with sixteen being the maximum number of registers used by either scheme. In the table, the first column gives the size of the graph (the number of nodes) using the coloring scheme developed by Chow and Hennessy. The number of separators found is given in the second column. The next three columns give the maximum, minimum and average graph sizes used in the clique separator approach. From these results, it is clear that the clique separator approach considerably reduces the size of the graphs that need to be colored and thus the space requirements of coloring.

Program	Original	#Sep	Max	Min	Ave
Changer	36	2	15	11	13
Intmm	74	2	41	21	28
Sieve	27	4	12	4	9
Bubble	54	3	16	15	16
Towers-P1	34	3	14	11	12
Towers-P2	45	3	20	6	15

Importantly, in all cases, the number of registers used by the clique separator scheme was either equal to or less than the number of registers used by the Chow and Hennessy coloring technique. For example, in the Sieve program, 13 registers were needed when coloring the entire graph but only 10 registers were used by utilizing separators. Future investigations will include experiments to determine the time efficiency of the clique separator approach and the effects of spilling on the performance of the scheme. As the separator approach is a more complicated approach than coloring the entire graph, we will also investigate the overhead of running the clique separator algorithm.

References

1. G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 17, no. 6, pp. 98-105, June, 1982.

2. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation via Coloring," *Computer Languages*, vol. 6, no. 1, pp. 47-57, 1981.

3. C.H. Chi and H.G. Dietz, "Register Allocation for GaAs Computer Systems," *21st Annual Hawaii International Conference on System Sciences*, vol. I, pp. 266-274, Jan., 1988.

4. F. Chow and J. Hennessy, "Register Allocation by Priority-based Coloring," *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 19, no. 6, pp. 222-232, June, 1984.

5. R. Cytron and J. Ferrante, "What's In a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation," *Proc. International Conf. on Parallel Processing*, pp. 19-27, August, 1987.

6. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers*, vol. 7, no. C-30, pp. 478-490, July, 1981.

7. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.

8. J.R. Larus and P.N. Hilfinger, "Register Allocation in the SPUR Lisp Compiler," *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pp. 255-263, 1986.

9. R.E. Tarjan, "Decomposition by Clique Separators," *Discrete Math.*, vol. 55, pp. 221-231, 1985.