Efficient Register Allocation via Coloring Using Clique Separators

RAJIV GUPTA, MARY LOU SOFFA, and DENISE OMBRES
University of Pittsburgh

Although graph coloring is widely recognized as an effective technique for register allocation, memory demands can become quite high for large interference graphs that are needed in coloring. In this paper we present an algorithm that uses the notion of clique separators to improve the space overhead of coloring. The algorithm, based on a result by R. Tarjan regarding the colorability of graphs, partitions program code into code segments using clique separators. The interference graphs for the code partitions are constructed one at a time and colored independently. The colorings for the partitions are combined to obtain a register allocation for the entire program. This approach can be used to perform register allocation in a space-efficient manner. For straight-line code (e.g., local register allocation), an optimal allocation can be obtained from optimal allocations for individual code partitions. Experimental results are presented demonstrating memory demand reductions for interference graphs when allocating registers using clique separators.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*hardware / software interfaces;* D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*

General Terms: Algorithms, Design, Languages, Performance

Additional Key Words and Phrases: Clique separators, graph coloring, interference graph, node priorities, spans, spill code

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 an interference graph is made such that no two nodes directly connected by an edge have the same color [Chaitin et al. 1981; Chaitin 1982; Chow and Hennessy 1984]. The nodes in an interference graph correspond to candidates for registers, and edges connect nodes that must be allocated different

This work was partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-9109089 to the University of Pittsburgh. A preliminary version of this paper appeared in the 1989 SIGPLAN Conference on Programming Language Design and Implementation.
Authors' current address: Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

registers. A coloring of this graph is equivalent to an assignment of registers. Existing techniques for register allocation can be divided into two categories: techniques for local register allocation and techniques for global register allocation. Local register allocation deals with the allocation of registers in straight-line code segments. The interference graphs for straight-line code belong to a class of graphs called interval graphs [Golumbic 1980]. Although graph coloring is an NP-complete problem for general graphs, interval graphs can be optimally colored in polynomial time [Garey and Johnson 1979]. Global register allocation deals with the allocation of registers in code containing branches. The interference graphs constructed during global register allocation are no longer interval graphs, and since general graph coloring is an NP-complete problem [Garey and Johnson 1979], polynomial time heuristics are used to obtain suboptimal colorings.

Due to the increased amount of analysis performed by modern optimizing compilers, interest in improving the space efficiency of such compilers is growing. As an example, sparse evaluation graphs are being used to limit the amount of data-flow information computed during program code optimization [Choi et al. 1990]. By avoiding the computation of intermediate results, the space requirement for saving data-flow information is reduced. In this paper we present an approach for reducing the space requirements during global register allocation. The size of an interference graph constructed for global register allocation can be large, and hence, it is desirable to develop techniques for limiting the memory demands of an interference graph. One approach that has been advocated performs inexpensive local register allocation before performing global register allocation using coloring [Chow and Hennessy 1984]. However, experience has shown that this strategy is usually not successful since very few candidate nodes are allocated registers during the local allocation phase, and hence most of the allocation is done during the global allocation phase [Larus and Hilfinger 1986]. Thus, to improve the efficiency of register allocation we must concentrate on improving the efficiency of global register allocation algorithms.

In this paper we present a technique that improves the space efficiency of graph-coloring-based global register allocation algorithms. The technique is based on a result due to Tarjan [1985] regarding the colorability of a graph by decomposition into subgraphs using clique separators, which states that if each subgraph can be colored using at most k colors, then the entire graph can be colored in k colors by combining the coloring of the subgraphs. In register allocation, the subgraphs resulting from a 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 the construction of the entire interference graph and then the execution of an algorithm on the graph to find the clique separators. The clique separators are found in the code by selecting traces (paths) through the control flow graph and finding separators along each trace. The interference graphs for the code partitions are constructed one at a time, and a coloring

heuristic (e.g., priority-based coloring) is used to color these subgraphs. Once a subgraph is colored, its storage can be reclaimed and used by another subgraph. The colorings for the subgraphs are combined, resulting in a coloring of the entire interference graph for the program.

Register allocation using clique separators is carried out efficiently because, at a given point during register allocation, only an interference graph for a single partition needs to be constructed. This reduces the space requirements for the interference graph. 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 on coloring reduces with the number of partitions. The time savings obtained during coloring partially offset the time spent on the detection of clique separators. The strategy is also suitable for parallel implementations, as the code in each trace can be decomposed in parallel and subgraphs can be colored in parallel. A parallel implementation will not provide any total savings in space, but will improve the run-time complexity of register allocation.

In the next section, we discuss background information, including a summary of Tarjan's results and a revised definition of name spans (or live ranges). An overall description of the technique is then presented in Section 3. In Section 4 we describe the partitioning technique using clique separators. In Section 5 two coloring algorithms that use separators are presented. The performance of a clique-based register allocation scheme is analyzed and an implementation of the technique and results based on experimental studies are presented in Section 6.

2. BACKGROUND

The technique for partitioning a program into code segments presented in this paper utilizes the notion of *clique separators* [Gavril 1977]. A clique separator is a completely connected subgraph whose removal disconnects the graph. The idea of decomposing a graph coloring problem using clique separators was first developed by Tarjan [1985]. Clique separators are used to decompose a graph into subgraphs that can be colored independently. A coloring for the entire graph is obtained from the colorings of the subgraphs. 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. For the graph shown in Figure 1a, 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 $S_2 = \{v_6, v_7, v_8, v_9\}$, given in Figure 1b. The subgraphs that must be colored using k colors, if a k-coloring for the entire graph is to be found, are shown in Figure 1c. 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 Figure 1b. In Figure 1c colorings of the subgraphs using three colors are given. These colorings are combined to obtain a 3-coloring for the entire graph shown in Figure 1d. 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 inter-

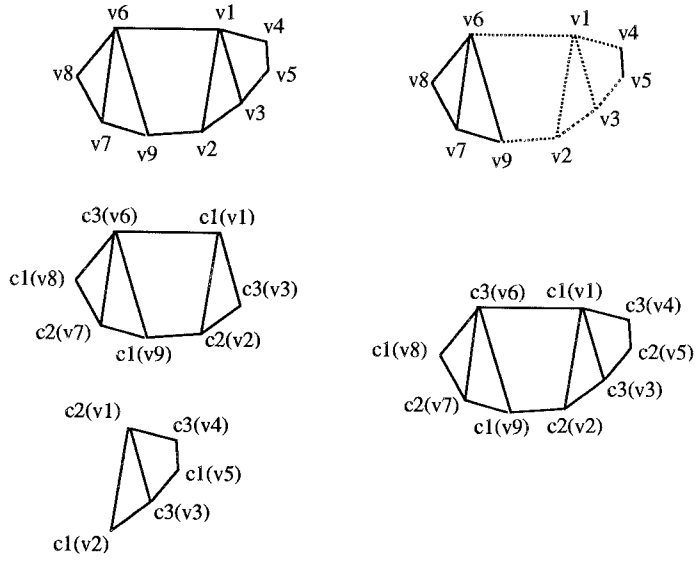

Fig. 1. Clique separators.

changing the use of colors c_1 and c_2 in the subgraphs that includes S_1 and CS. The subgraphs resulting from a decomposition may be further decomposable using clique separators. A graph that cannot be decomposed any further is called an *atom*. In the above example, S_1 is an atom, but 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 is decomposed. This approach is not useful for register allocation because it does not reduce the space complexity of the algorithm. To avoid this problem, clique separators in this paper are identified by examining the program code instead of the interference graph of the entire program code.

Another technique to divide a program into partitions for which register allocation can be carried out independently has been developed by Chi and Dietz [1988] using *cut points*. 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 use of clique separators to partition code is 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.

Before we can construct interference graphs, we must identify the name spans that are represented as nodes of the interference graph. In earlier global register allocation algorithms, a name span of a variable consisted of a group of basic blocks in the control flow graph. However, in this work name

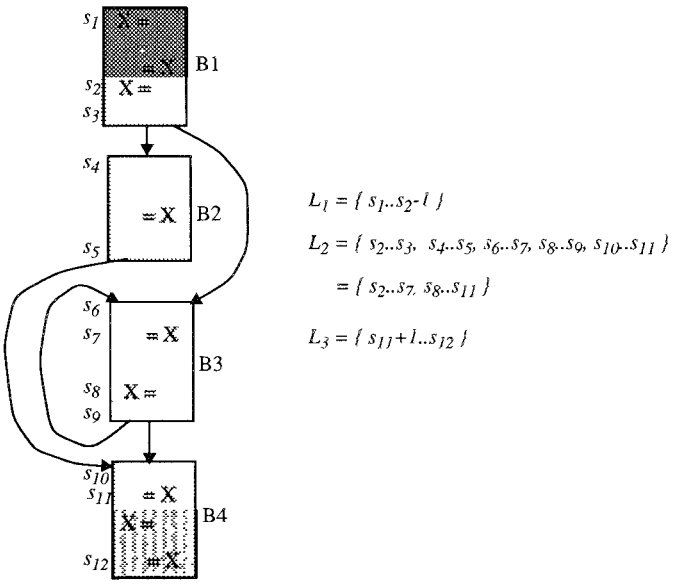

Fig. 2. Constructing spans.

spans are composed of a group of code statements that may include portions of basic blocks. This modification is essential because clique separators of interest may appear at any point in the code and not necessarily at basic block boundaries.

Definition. A span corresponding to a variable X is defined as an isolated group of contiguous code statements that satisfies the following conditions: (1) There is no use of X outside the span that is reachable from a definition of X inside the span, and (2) there is no use of X in the span that is reachable from a definition of X outside the span.

Consider the flow graph in Figure 2. In this example three spans are created for variable X. The definition of X in statement s_1 is used only by statement $s_2 - 1$, thus creating the span L_1. The definition of X in statement s_2 is used in basic blocks B2, B3, and B4, which causes groups of statements $s_2 \ldots s_3$, $s_4 \ldots s_5$, $s_6 \ldots s_7$, and $s_{10} \ldots s_{11}$ to be included in span L_2. Furthermore, since the use of X in statement s_{11} is also reachable from the definition of X in statement s_8, the statements $s_8 \ldots s_9$ are also included in L_2. The third span L_3 is composed of statements $s_{11} + 1 \ldots s_{12}$. If the code is stored in a linear code array and the code for basic blocks is stored in the order B1, B2, B3, and B4, then the span L_2 can be represented as consisting of statement groups $s_2 \ldots s_7$ and $s_8 \ldots s_{11}$.

3. OVERVIEW OF THE REGISTER ALLOCATION TECHNIQUE

A high-level algorithm for allocating registers using clique separators is given in Figure 3. The approach is independent of the coloring scheme used in the actual register allocation. A control flow graph representation of the program

```
Given: An control flow graph representation of a program.
Output: Register allocation for the program.

Compute the name spans.
PartitionList ← ∅
while the entire program has not been partitioned do
   Select a trace composed of basic blocks that have not been included in
     previous traces, giving preference to the blocks with higher execution counts.
   Partition the trace by identifying clique separators and add the
     partitions at the end of the PartitionList.
endwhile
for each partition P in PartitionList do
      Construct the interference graph for P.
      Color the graph using some graph coloring heuristic.
      Combine the colorings with already processed partitions which involves:
       renaming colors for the current partition; and
       possibly generation of copy code at merge and join points.
endfor
```

Fig. 3. Overview of register allocation using clique separators.

is assumed. Details about the individual steps in the algorithm are explained in subsequent sections.

A program is partitioned by selecting traces (paths) through the control flow graph and finding clique separators in each trace [Fisher 1981]. A trace consists of a sequence of basic blocks along an execution path. The statements belonging to a trace are examined in sequence, and clique separators in the trace are identified. After partitioning one trace, another trace is chosen, and the partitioning process is repeated until the entire program is partitioned. Each basic block is only included in one trace. When partitioning a trace, all live variables at the beginning of a trace are assumed to be defined at the start of the trace. The resulting partitions contain statements from basic blocks that are not only connected to each other, but also lie along an execution path. The selection of traces is typically based on the time a program spends in different parts of the program [Fisher 1981]. The traces constructed earlier in the selection process account for a greater percentage of the program's execution time than those selected later. The order in which the partitions are processed by the register allocation heuristic is the same as the order in which traces were found. This process of ordering results in the generation of better quality code for parts of the program where more execution time is spent. Branch and merge points of traces are handled during register allocation by incorporating live spans that overlap the traces.

Consider a situation where a segment of straight-line code has been partitioned using clique separators. An optimal register allocation for the entire segment can be constructed from optimal solutions for each of the partitions. This is a highly desirable characteristic since compilers for the current generation of superscalar processors, in an effort to exploit instruction-level parallelism, create large segments of straight-line code through program transformations such as loop unrolling [Dongarra and Jinds 1979]

and in-line expansion [MacLaren 1984]. If the code contains branches, better register allocation can be achieved for partitions processed earlier, at the expense of code quality for partitions processed later. Thus, code partitions resulting from traces with high probability of execution are processed before other partitions. As register allocation proceeds from one partition to the next partition, an attempt is made to assign the same register to a name span extending across multiple partitions. This goal is achieved by renaming colors assigned to a partition. If different registers must be assigned to different portions of a name span, code for moving values among the registers is generated. This copy code is typically introduced at branch or join points in the code. If a portion of a name span is not assigned any register (i.e., it is spilled), load and store instructions are also introduced.

4. PARTITIONING TRACES INTO SEGMENTS

First consider the problem of partitioning straight-line code into code segments for which register allocation can be carried out independently. At any given point in a code segment, there are several overlapping spans that are represented by nodes of a clique in the interference graph. The clique corresponding to any program point in the code segment represents a separator. This is because the removal of the clique 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 spans 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 cause register allocation to be more time expensive.

The above problem can be avoided by choosing 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 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. One approach for partitioning is to first partition a code segment assuming c has the value one. If a code partition is larger than the maximum acceptable size, we can further partition this code segment by increasing the value of c to two. We can continue to increase the value of c until all resulting partitions are sufficiently small. Our experience has shown that, if c is chosen to be one, very few separators are found. However, if c is chosen to be two, frequent separators are found, in practice. Thus, in this work we use an algorithm that

uses the value two for c. However, if the size of a partition reaches a certain maximum, the code is partitioned at that point.

The example presented in Figure 4 contains a separator that consists of the spans {b, i, f} and that satisfies the condition $c = 2$. This separator divides the code segment into two parts; hence, the interference graph is divided into the two subgraphs shown in Figure 4. 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 resulting subgraphs. In the example shown, {b, i, f} is included in both subgraphs. The subgraphs are colored independently, and their colorings are combined to obtain the coloring for the entire graph. As mentioned before, if there are no branches in the code the colorings can always be combined. However, in the presence of branches we cannot always combine the colorings through renaming.

To identify the separators, we scan the code in a trace from beginning to end, constructing and updating three sets, namely, *PRE*, *POST*, and *CLIQUE*. By examining the sets, we 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 either overlap at least one of the members of *CLIQUE*, are not colored, or interfere with a node not colored. The set *POST* contains the spans that have not yet begun, but overlap with at least one member of the set *CLIQUE*. Thus, in the example in Figure 4, 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 iff 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 nonempty. 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 choosing a separator, sets *PRE* or *POST* are nonempty 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. In addition to locating separators in the above fashion, our algorithm also keeps track of the size of the current partition. If the number of spans in the current partition exceeds a certain maximum (*MAXSIZE*), the clique at that point is chosen as a separator. The algorithm that constructs the sets and checks for separators is summarized in Figure 5.

5. REGISTER ALLOCATION COLORING ALGORITHMS USING CLIQUES

When partitioning a trace, interference graphs are constructed for each partition, one at a time. The graphs are colored, and the results are combined with graphs of adjacent partitions. A new trace is selected, and partitioning of

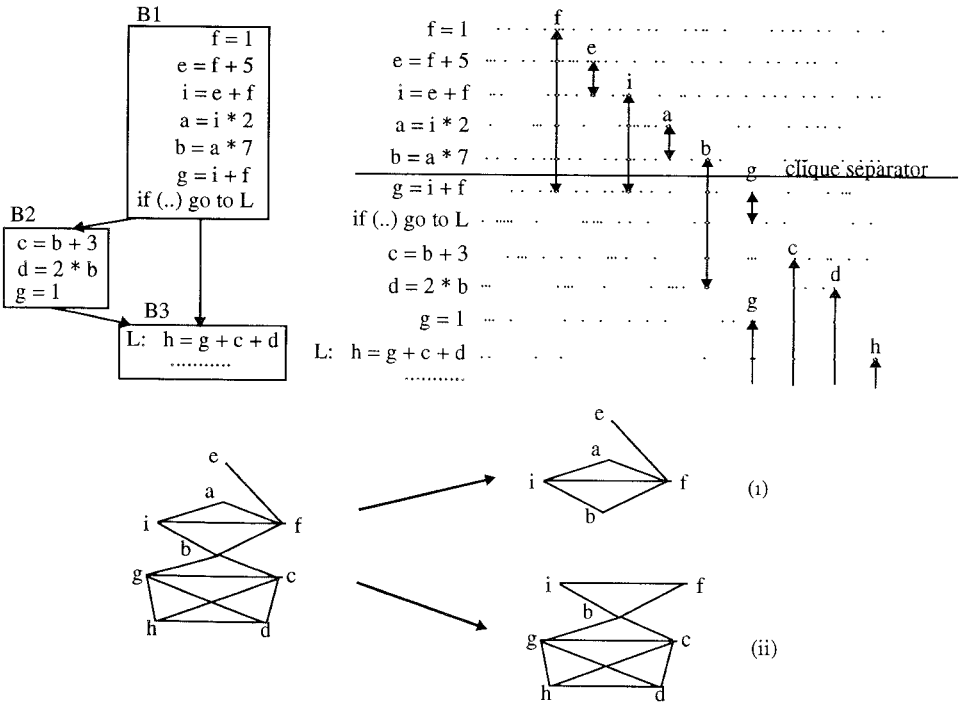

Fig. 4 Clique separators in program code.

that trace is performed. This process culminates in register allocation for the entire program code.

The spans that form a clique separator are present in the interference graphs of the code partitions preceding and succeeding the clique. Thus, they may be allocated 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 allocated registers that cannot be renamed. In this situation, code to transfer values from one register to another is introduced. 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.

Next we present adaptations of two specific coloring-based algorithms, Chaitin's [1982] algorithm and Chow and Hennessy's [1984] priority-based algorithm, to exploit the notion of clique separators.

5.1 Chaitin's Algorithm

A register allocator based on Chaitin's algorithm is given in Figure 6. An interference graph is constructed for a partition and then colored using Chaitin's coloring heuristic. The coloring heuristic removes each node in the

```
Partition Trace {
    PRE = POST = CLIQUE = ϕ;
    START = CURRENT = first instruction in the trace;
    SIZE = 0; PARTITIONLIST = ϕ;
    repeat
       for each span s_start that starts at the current instruction do
           SIZE = SIZE + 1;
           CLIQUE = CLIQUE ∪ {s_start}
           POST = POST - {s_start}
           for each span s_i, such that s_i has not yet started and s_start overlaps s_i do
               POST = POST ∪ {s_i}
           endfor
       endfor
       for each span s_end that ends at the current instruction do
           PRE = PRE ∪ {s_end}
           CLIQUE = CLIQUE - {s_end}
       endfor
       for each span s_i that no longer overlaps a member of CLIQUE
               ∧ is marked as belonging to a partition
               ∧ all of the spans with which it interferes are also marked
       do PRE = PRE - {s_i} endfor
       if (CURRENT = last instruction in the trace) then
           add partition containing instructions from START to CURRENT to the PARTITIONLIST
       else if ( SIZE = MAXSIZE ) then
           add partition containing instructions from START to CURRENT to the PARTITIONLIST
           SIZE = | CLIQUE | ; START = CURRENT = next instruction in the trace;
       else if Check ( CLIQUE ) then
           add partition containing instructions from START to CURRENT to the PARTITIONLIST
           SIZE = | CLIQUE | ; START = CURRENT = next instruction in the trace;
       endif
     until (CURRENT = last instruction in the trace)
   }

Check ( CLIQUE ) {
    CLIQUE_PRE = { s : s ε CLIQUE and it overlaps a span from PRE }
    CLIQUE_POST = { s : s ε CLIQUE and it overlaps a span from POST }
    CLIQUE_POST =
    if ( PRE ≠ ϕ ) ∧ ( POST ≠ ϕ )
        ∧ ( CLIQUE_PRE ∩ CLIQUE_POST = ϕ )
    then return(true) endif
    return(false)
   }
```

Fig. 5. Finding separators.

graph for which the number of edges incident to the node is less than the number of colors. If no such node exists, then spilling is needed. A span is chosen to spill, and its associated node is then deleted from the graph. The attempt to color continues, spilling again if necessary until all nodes have been removed. The node chosen to spill is selected based on the number of edges connected to the node and the nesting levels of the code partition. The

```
Allocate Registers {
    Split the program into execution traces
    repeat
        Select a trace that has not been processed
        repeat
            Find a partition in the current trace
            repeat
                Construct interference graph for the partition
                repeat
                    Delete all nodes with the number of neighbors less than the number of registers.
                    if graph not empty then
                        from the remaining nodes choose one to spill
                        Spill all its uses and definitions along this trace
                    endif
                until graph is empty
            until no new spills occur
            Color the Graph
            Combine colors and generate copy code if needed
        until no more partitions in the trace
    until until no more traces
}
```

Fig 6. Register allocation algorithm based on Chaitin's approach.

higher the number of edges, the more likely it is that after the removal of the node it will be possible to color other nodes in the graph. The nesting level is considered in an attempt to ensure that spill code is introduced in code partitions that are not nested inside loops and, therefore, are executed less frequently. After all nodes have been removed, if any spilling was necessary, then the graph is reconstructed using the revised code for the partition. Once the coloring of the graph is successful, register allocation performed in the current partition is propagated to other partitions that contain the same spans but have not been colored, eliminating the need to combine the colorings of individual partitions by renaming colors. Any spill code for the partition is also generated. This technique incorporates spills by storing the definition and loading the value before each of the uses along the trace being processed.

In Chaitin's algorithm, a name span was either entirely spilled or assigned a register for the entire duration of the live range. However, in our algorithm, if a definition is spilled along a trace, it is not considered spilled for any new trace. This introduces some copy code at the onset of the trace, moving the definition from memory to the register allocated on the new trace. However, this amount of copy code is likely to cost less than the cost of spilling a span in all traces.

5.2 Priority-Based Algorithm

We next describe a priority-based coloring algorithm that uses our partitioning method. In implementing a priority-based coloring algorithm, a method for computing node priorities is used. The priority of a node or span is measured in terms of the savings in execution time (*TOTALSAV*) that are

incurred by its being allocated a register instead of memory [Chow and Hennessy 1984]. A variable referenced inside a loop body is likely to be referenced more often, and hence, the total savings resulting from allocating a register to a span is normalized with respect to the loop-nesting depth. The priorities are maintained to guide the coloring of an interference graph. During coloring, when a portion of a span is spilled, the priority of the span is updated.

An overall algorithm for global register allocation using clique separators and based on priorities is summarized in Figure 7. In this algorithm, the program is partitioned into code segments, and an interference graph for a single partition is constructed and colored. The span priorities are maintained globally and updated as portions of spans are spilled. One by one, the subgraphs are constructed and colored. The constrained nodes in the graph, those that have fewer neighbors than the number of registers, are colored last, as they can be colored no matter what colors are allocated to their neighbors. During the coloring of a partition, only priorities of those nodes that belong to the current partition are examined. A constrained node in the current graph is colored before other nodes with lower priorities are considered. The node with highest priority is selected and colored if a register is available. If the portions of the name span represented by this node had already been assigned a color during the processing of other partitions, an attempt is made to assign the same color. If this is not possible, another color is assigned, and following the coloring of the partition, copy code is introduced at appropriate points in the program. If no register is available to color a currently selected node, the portion of the span belonging to the current partition is spilled, and the priority of the span is updated. Overallocation of registers is prevented by allocating registers to only those spans for which *TOTALSAV* is positive.

In the algorithm presented, the priority of a node, which is the value *TOTALSAV*, is not normalized by the length of the span. In the algorithm developed by Chow and Hennessy [1984], the priority is normalized by the length of the span, 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, based on the occurrence frequencies of variables, smoothes out the frequency differences in spans during global allocation. Thus, the adjustment of the priority by the span 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.

6. PERFORMANCE EVALUATION

To evaluate the performance of the clique separator approach to register allocation, we present space and time complexities for the coloring process and experimental results detailing the space savings and cost of the clique technique.

```
Allocate Registers {
    for each span do compute priority TOTALSAV endfor
    unconstrained ← { nodes whose degree is less than the number of registers }
    constrained ← { all nodes that do not belong to unconstrained set }
    Partition the program by identifying the separators and
      determine the order for processing the partitions.
    repeat
        Construct the interference graph for the partition to be processed next.
        repeat
            Choose span lr from constrained with highest priority TOTALSAV.
            if lr has more colored neighbors than number of registers then
                Spill the portion of span lr in the current partition.
                Update the priority TOTALSAV for lr.
            elseif color assigned to lr in earlier partitions is not available then
                Assign another color to lr and introduce Copy code.
            else
                Assign appropriate color to lr.
            endif
        until all constrained nodes in current partition have been processed
    until all partitions have been processed
    Assign colors to unconstrained nodes.
}
```

Fig. 7. Priority-based register allocation.

In the analysis below, n is the number of live ranges, and m is the number of program partitions created by the clique separators. We assume that nodes to be colored are chosen using a priority-based scheme.

Space complexity. 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 a priority-based coloring heuristic when applied to an interference graph with x nodes is $O(x^2)$, since in each iteration of the loop, one span 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 $O(n/m)$ spans. Since there are m partitions to process, the run-time complexity of the coloring algorithm is $O(n^2/m)$. This time does not include the time for partitioning.

In the above analysis, it is assumed that the subgraphs resulting from partitioning are constructed and colored one at a time. An approach for further speeding up the coloring process is to construct the graphs and color them in parallel. However, there will not be any savings in storage, as all graphs would have to be constructed simultaneously. The complexities of various priority-based register allocation approaches are summarized in Table I.

Table I. Complexities of Priority-Based Register Allocation Approaches

	Without Partitioning	With partitioning	
		Sequential	Parallel (m processors)
Space	$O(n^2)$	$O(n^2/m^2)$	$O(n^2)$
Time	$O(n^2)$	$O(n^2/m)$	$O(n^2/m^2)$

The clique separator technique using Chaitin's approach and described in Section 3 was implemented in C on a Sun 3/50 Workstation. In order to analyze the space performance of the clique separator approach, the exhaustive technique using Chaitin's coloring algorithm in which one interference graph for the entire program is constructed was also implemented. Both techniques use the same heuristics in selecting nodes to color and spill. Results of experiments to investigate the space performance of the two approaches for a sample of programs are given in Table II. 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.

The experiments were designed to determine the difference in sizes of the interference graphs generated by the two methods and the quality of code produced. Table II displays the results assuming an unlimited numbers of registers, thus eliminating the need for spill code. In the table, the first column is the name of the program, and the second column gives the number of nodes in the interference graph for the entire program that was generated by the exhaustive algorithm. The column labeled "Clique subgraphs" lists all of the sizes of interference graphs constructed using the clique technique, with the average-size subgraph given in the fourth column, labeled "Average size." The last column, "Savings," gives the percentage difference between the graph constructed with the exhaustive technique and the maximum-size graph constructed using cliques.

As can be seen from Table II, the size of the graphs constructed by the clique method were considerably smaller than that constructed for the entire program. On average, the size of the graphs colored in the clique separator method are more than five times smaller than the graph generated by the exhaustive approach. The largest graph found using cliques in each program was, on average, more than two times smaller than the graph for the entire program. The results indicate a significant memory savings, since once the graphs have been used, the storage can be reclaimed for use by another graph. These experiments also found that the same number of registers were allocated in both schemes. Thus, the same quality of code was produced when no spills were generated.

Another set of experiments was performed to compare the two register allocation approaches in the presence of spilling. Table III presents the results of these experiments. The second column in Table III gives the number of registers that are available to the register allocators, and the fourth column gives the number of spills for each technique. These results

Table II. Comparison of Techniques without Spilling

Program	Nodes in Entire program	Clique subgraphs	Average size	Savings (%)
Sieve	34	4, 5, 5, 4, 8, 6, 7, **9**, 5, 6	6	74
Bubble Sort	57	1, 17, 5, 10, 15, **23**, 8, 8	11	59
Changer	26	3, 7, **11**, 2, 7	6	58
Towers of hanoi	6	**6**	6	0
Hanoi proc	11	**11**	11	0
FFT	233	6, 28, 5, 2, 10, 14, 13, 14, 12, **92**, 20, 10, 4, 42, 20, 15, 14, 12	19	60
Matrix Multiply	74	2, 4, 5, 4, 3, 6, 15, **34**, 7, 17, 7, 17	10	54

indicate that both techniques performed about the same in terms of generating spill code. In all but one case, the same number of spills was generated. In that one case, the clique approach generated one more spill; however, since heuristics are being used in both cases, the exhaustive method could also generate more spills. Importantly, these results indicate that the quality of code produced by the two methods is similar.

Table III also gives the time that it took both allocators to execute on the longer programs. These numbers were generated to determine the increased execution time of the clique technique. To determine these numbers, the allocators were run five times, and the results were averaged. Column 5 gives the total time that it took each allocator to execute. Columns 6–9 give a more detailed analysis of the timings for the clique separator approach: The times given in column 6 are for partitioning the program into cliques, in column 7 for building the interference graphs, in column 8 for coloring the graphs, and in column 9 for spilling registers. From the timings, the clique separator took more time to execute, basically due to the partitioning. The percentage of increased execution time of the clique approach to the exhaustive approach ranged from 34 percent to 96 percent. Approximately 60 percent of the execution time for the clique technique was spent in partitioning. The clique technique took less time to build the graphs, less time to color, and less time to spill than did the exhaustive approach. From these results, it is clear that, although the time to allocate using cliques may be more than for the exhaustive approach, it is a practical approach and can reduce memory overhead.

7. CONCLUSION

We have presented a technique for allocating registers using coloring that avoids construction of the interference graph for the entire program code. Graphs for smaller portions of the code are constructed and then colored independently. The colors of adjacent graphs are then combined, leading to a coloring of the entire graph. By using this method, the memory demands for coloring are dramatically reduced. Experimental results indicate that the

Table III. Clique Performance in the Presence of Spilling

Program Name	Number of Registers	Method	Number of spills	Timing (average of 5 samples in seconds)				
				Total	Partition	BuildGraph	Color	Spill
Bubble Sort	7	Chaitin	2	2.35	—	0.98	0.26	0.00
		Clique	2	3.23	2.33	0.67	0.24	0.00
	8	Chaitin	1	2.48	—	1.02	0.25	0.00
		Clique	1	3.32	2.28	0.68	0.28	0.00
Matrix Multiply	9	Chaitin	4	3.33	—	1.34	0.40	0.02
		Clique	4	4.88	2.89	1.15	0.47	0.02
	10	Chaitin	3	3.39	—	1.31	0.46	0.00
		Clique	3	4.85	2.80	1.17	0.48	0.00
FFT	9	Chaitin	8	26.60	—	11.84	2.52	0.10
		Clique	9	51.72	32.62	9.98	2.44	0.08
	10	Chaitin	6	24.86	—	11.40	2.07	0.06
		Clique	6	44.82	26.91	7.55	1.95	0.04
	11	Chaitin	4	24.96	—	11.46	2.18	0.06
		Clique	4	45.22	27.09	7.66	1.97	0.02
	13	Chaitin	1	25.25	—	11.94	2.01	0.02
		Clique	1	44.10	25.82	7.68	1.91	0.02

largest subgraph constructed is about five times smaller than the entire interference graph. We also demonstrated that the time to allocate registers using cliques is higher than constructing and coloring the entire interference graph due to the need to partition the code. It is expected that the clique approach would lead to better time performance when the heuristic used to color the graph is expensive and based on the number of nodes. Using the clique approach, more expensive algorithms, such as the optimal, may be possible.

Although program traces were used in the technique presented in this paper to determine cliques, another approach is to consider the basic blocks in a flow graph one at a time [Gupta et al. 1989]. In this approach clique separators are determined across basic block boundaries, taking into account the branching and merging of control flow. At a divergence of control flow, the different paths are considered independently, searching for clique separators. When control flow merges, a separator for one of the paths is found (the one that is more likely to execute), and the other path uses this clique separator.

Although we basically have considered performing register allocation sequentially, another advantage of the clique-based approach is that a parallel version of register allocation can be performed. Once the traces are found, the code partitioning can be performed in parallel for each trace, and then the graphs can all be constructed and colored in parallel. Renaming would have to be done to adjust the colors of the registers. We are currently developing a parallel algorithm for allocating registers using the clique approach in order to investigate its performance. We are also considering the performance of using cliques when more expensive algorithms are used for coloring.

ACKNOWLEDGMENTS

We would like to thank Jiyang Liu for assisting in performing the experiments. We would also like to thank Lori Pollock and the reviewers for their constructive comments on an earlier version of this paper.

REFERENCES

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In Proceedings of the SIGPLAN 82 Symposium on Compiler Construction. *SIGPLAN Not* (ACM) *17*, 6 (June), 98–105.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Comput. Lang 6*, 1, 47–57.

CHI, C. H. AND DIETZ, H. G. 1988. Register allocation for GaAs computer systems. In *21st Annual Hawaii International Conference on System Sciences*, vol. I (Jan.). IEEE Computer Society, Washington, D.C., 266–274

CHOI, J.-D., CYTRON, R., AND FERRANTE, J. 1990. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York, 55–66.

CHOW, F. AND HENNESSY, J. 1984. Register allocation by priority-based coloring. In Proceedings of the SIGPLAN 84 Symposium on Compiler Construction. *SIGPLAN Not* (ACM) *19*, 6 (June), 222–232.

DONGARRA, J. J. AND JINDS, A. R. 1979. Unrolling loops in Fortran. *Softw. Pract. Exper. 9*, 3 (Mar.), 219–226.

FISHER, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30*, 7 (July), 478–490.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness* Freeman, San Francisco, Calif.

GAVRIL, F. 1977. Algorithms on clique separable graphs. *Discrete Math. 19*, 159–165.

GOLUMBIC, M. C. 1980. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York.

GUPTA, R., SOFFA, M. L. AND STEELE, T. 1989. Register allocation via clique separators. In *Proceedings of SIGPLAN 89 Conference on Programming Language Design and Implementation* (June). ACM, New York, 264–275.

LARUS, J. R. AND HILFINGER, P. N. 1986. Register allocation in the SPUR Lisp compiler. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*. ACM, New York, 255–263.

MACLAREN, M. D. 1984. Inline routines in VAXELN Pascal. In Proceedings of the SIGPLAN Symposium on Compiler Construction. *SIGPLAN Not.* (ACM) *19*, 6 (June), 226–275.

TARJAN, R. E. 1985. Decomposition by clique separators. *Discrete Math. 55*, 2, 221–231

Received September 1990; revised May 1993; accepted June 1993