Computer Languages Vol. 6, pp. 47 to 57. 1981
Printed in Great Britain.

0096-0551/81/010047-11$02.00/0

REGISTER ALLOCATION VIA COLORING

GREGORY J. CHAITIN, MARC A. AUSLANDER, ASHOK K. CHANDRA, JOHN COCKE,
MARTIN E. HOPKINS and PETER W. MARKSTEIN
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

(*Received* 9 *October* 1980)

Abstract—Register allocation may be viewed as a graph coloring problem. Each node in the graph stands for a computed quantity that resides in a machine register, and two nodes are connected by an edge if the quantities interfere with each other, that is, if they are simultaneously live at some point in the object program. This approach, though mentioned in the literature, was never implemented before. Preliminary results of an experimental implementation in a PL/I optimizing compiler suggest that global register allocation approaching that of hand-coded assembly language may be attainable.

Register allocation Optimizing compilers Graph coloring

1. OVERVIEW OF REGISTER ALLOCATION

In this paper we describe the Register Allocation Phase of an experimental PL/I compiler for the IBM System/370. (For an overview of the entire compiler see Cocke and Markstein [1], for background information on optimization, see Refs [1] and [2].) It is the responsibility of this phase to map the unlimited number of symbolic registers assumed in the intermediate language into the 17 real machine registers, namely the 16 general-purpose registers (*R*0–*R*15), and the condition-code (*CC*).

The essence of our approach is that it is uniform and systematic. Compiler back-ends must deal with the idiosyncrasies of the machine instructions; for example, register pairs, the fact that register *R*0 is an invalid base register, and that the contents of some machine registers are destroyed as a side-effect of particular instructions. In our approach all these idiosyncrasies are entered in a uniform manner in our data structure, the interference graph. Afterwards this data structure is manipulated in a very systematic way.

Also, our approach has a rather different personality than traditional ones because we do *global* register allocation across entire procedures. Furthermore, except for the register which always contains the address of the DSA ("dynamic storage area", i.e. current stack frame) and is the anchor for all addressability, all other registers are considered to be part of a uniform pool and all computations compete on an equal basis for these registers. Most compilers reserve subsets of the registers for specific purposes; we do the exact opposite.

In our compiler a deliberate effort is made to make things as hard as possible for register allocation, i.e. to keep as many computations as possible in registers rather than in storage. For example, automatic scalars are usually kept in registers rather than in the DSA, and subroutine linkage also attempts to pass as much information as possible through registers. It is the responsibility of code generation and optimization to take advantage of the unlimited number of registers allowed in the intermediate language in order to minimize the number of loads and stores in the program, since these are much more expensive than register to register instructions. Then hopefully register allocation will map all these registers into the 17 that are actually available in the hardware. If not, it is register allocation's responsibility to put back into the object program the minimum amount of spill code, i.e. of stores and reloads of registers, that is needed.

As long as no spill code need be introduced, we feel that our approach to register allocation does a better job than can be done by hand-coders. For example, if there is a slight change in a program, when it is recompiled the Register Allocation Phase may produce a completely different allocation to accommodate the change. A hand-coder would be irresponsible to proceed in such a fashion. We also feel that our compiler

succeeds in keeping things in registers rather than in storage better than other compilers, and that this is one of the salient features of the personality of the object code we produce. Moreover the mathematical elegance of the graph coloring approach described below, its systematic and uniform way of dealing with hardware idiosyncrasies, and the fact that its algorithms are computationally highly efficient, are convincing arguments in its favor.

2. REGISTER ALLOCATION AS A GRAPH COLORING PROBLEM

Our approach to register allocation is via graph coloring. This has been suggested by Cocke [2], Yershov [3], Schwartz [4], and others, but has never been worked-out in detail nor implemented before. Recall that a coloring of a graph is an assignment of a color to each of its nodes in such a manner that if two nodes are adjacent, i.e. connected by an edge of the graph, then they have different colors. A coloring of a graph is said to be an n-coloring if it does not use more than n different colors. And the chromatic number of a graph is defined to be the minimal number of colors in any of its colorings, i.e. the least n for which there is an n-coloring of it.

It is well-known [5] that given a graph G and a natural number $n > 2$, the problem of determining whether G is n-colorable, i.e. whether or not there is an n-coloring of G, is NP-complete. This suggests that in some cases an altogether impractical amount of computation is needed to decide this, i.e. that in some cases the amount of computation must be an exponential function of the size of G.

In fact experimental evidence indicates that the NP-completeness of graph coloring is not a significant obstacle to a register allocation scheme based on graph coloring. However it should be pointed out that given an arbitrary graph it is possible to construct a program whose register allocation is formulated in terms of coloring this graph (see Appendix 2). Thus some programs must give rise to serious coloring problems.

Our approach to register allocation is to build a *register interference graph* for each procedure in the source program, and to obtain 17-colorings of these interference graphs. Roughly speaking, two computations which reside in machine registers are said to interfere with each other if they are live simultaneously at any point in the program.

For each procedure P in the source program an interference graph is constructed whose nodes stand for the 17 machine registers and for all computations in the procedure P which reside in machine registers, and whose edges stand for register interferences. If the chromatic number of this graph is 17, then a register allocation has been achieved, and the register assigned to a computation is that one of the 17 machine registers which has the same color that it does. Thus computations which interfere cannot be assigned to the same machine register. On the other hand, if the chromatic number is greater than 17, then spill code must be introduced to store and reload registers in order to obtain a program whose chromatic number is 17.

3. THE CONCEPT OF INTERFERENCE

If a program has two loops of the form *DO J = 1 TO 100*, J could be kept in a different register in each of the loops. In order to make this possible, each symbolic register is split into the connected components of its def-use (definition-use) chains, and it is these components, called names, which are the nodes of our interference graph. This is especially important because we always do global register allocation for entire procedures. Much additional freedom in coloring is obtained by uncoupling distant regions of the procedure by using names instead of symbolic registers as the nodes of the interference graph. However, as we explain below, some of these names are later coalesced, at which point the mapping from symbolic registers to names becomes many-many rather than one-many.

Our notion of liveness is not quite the same as that used in optimization. We consider a name X to be live at a point L in a program P if there is a control flow path from the entry point of P to a definition of X and then through L to a use of X at point U, which

has the property that there is no redefinition of X on the path between L and the use of X at U. I.e. a computation is live if it has been computed and will be used before being recomputed.

Above it was stated that two names interfere if they are ever live simultaneously. Thus if at a point in the program there are k live names N_i, it is necessary to add $k(k-1)/2$ edges to the interference graph. However, we do not actually do this. If k names N_i are live at the definition point of another name N', we add the k interferences (N',N_i) to the graph. In other words, the notion of interference that we actually use is that two names interfere if one of them is live at a definition point of the other. This interference concept is better than the previous one for two reasons: it is less work to build the interference graph (k edges added vs $k(k+1)/2$), and there are programs for which the resulting interference graph has a smaller chromatic number. Here is an example of such a program:

```
P: PROC(MODE);

 DCL
    MODE          BIT(1),
    (A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,
     B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,
     SUM)         FIXED BIN(15) AUTO,
    (U(10),V(10))  FIXED BIN(15) STATIC EXT;

 IF MODE
    THEN DO;
        A1=U(1); A2=U(2); A3=U(3); A4=U(4); A5=U(5);
        A6=U(6); A7=U(7); A8=U(8); A9=U(9); A10=U(10);
    END;
    ELSE DO;
        B1=V(1); B2=V(2); B3=V(3); B4=V(4); B5=V(5);
        B6=V(6); B7=V(7); B8=V(8); B9=V(9); B10=V(10);
    END;

 LABEL:;

 IF MODE
    THEN SUM = A1+A2+A3+A4+A5+A6+A7+A8+A9+A10;
    ELSE SUM = B1+B2+B3+B4+B5+B6+B7+B8+B9+B10;

 RETURN (SUM);

END P;
```

At the point in the program P marked *LABEL* the ten A_i and the ten B_i are simultaneously live, and so is *MODE*. Thus with the first method of building the interference graph there is a 21-clique and the chromatic number of the graph is 21. [Recall that an n-clique is an n-node graph with all possible $n(n-1)/2$ edges.] With the second method, however, none of the ten A variables interferes with any of the ten B variables, and the chromatic number of the interference graph is only 11. (A technical point: we have ignored the fact that all our interference graphs contain the 17-clique of machine registers as a subgraph. Thus the chromatic number is actually 17 instead of 11.)

4. MANIPULATING THE INTERFERENCES

There are 3 stages in processing the interference graph of a procedure. The first stage is building the graph in the manner described above. This is done by the routine C__ITF. The second stage is coalescing nodes in this graph in order to force them to get the same color and be assigned to the same machine register. This is done by the routine C__LR. The third and final stage is attempting to construct a 17-coloring of the resulting graph. This is done by a fast routine called C__CLR, or by a slower routine C__NP which uses backtracking and is guaranteed to find a 17-coloring if there is one. Of course, backtracking is dangerous; in some unusual circumstances C__NP uses exponential amounts of time.

We now make a few general remarks about the preprocessing of the interference graph which is done for the purpose of assuring that separate nodes in the graph must get the same color. This is done by coalescing nodes, i.e. taking two nodes which do not interfere and combining them in a single node which interferes with any node which either of them interfered with before. Note that coalescing nodes in the graph before coloring it is also a way of doing some pre-coloring, for any node which is coalesced with one of the 17 machine registers has in fact been assigned to that register. Of course, such pre-colorings are a strong constraint on the final coloring, and should be avoided if possible, preferably replaced by coalesces not involving real machine registers. It should be pointed out that preprocessing the graph in this manner gives much better results than warping the coloring algorithms to try to give certain nodes the same color.

Here is an example of a typical situation in which one might wish to coalesce nodes. If there is a *LR T, S* (load register T from S) in the object program, it is desirable to give the names S and T the same color so that it isn't actually necessary to copy the contents of register S into register T and thus the Final Assembly Phase needn't emit any code for this intermediate language instruction. (This optimization is traditionally referred to as subsumption.) C__LR achieves this by checking the source S and target T of each *LR* instruction in the object program to see whether or not they interfere. If they don't, then C__LR alters the graph by combining or coalescing the nodes for S and T. Thus any coloring of the graph will necessarily give them the same color.

However, in order to make this work well, the definition of interference presented above must be altered yet again! The refinement is that the target of an *LR* doesn't necessarily have to be allocated to a different register than its source. Thus a *LR T,S* at a point at which S and the k names N_i are live only yields the k interferences of the form (T,N_i), but not the interference (T,S). (See Appendix 1 for a consistent philosophy of the "ultimate" notion of interference and approximations to it.)

Subsumption is a very useful optimization, because intermediate language typically contains many *LR*'s. Some of these are produced for assignments of one scalar to another. But even more are generated for subroutine linkages and are introduced by value numbering and by reduction in strength. Besides eliminating *LR*'s by coalescing sources and targets, C__LR also attempts to coalesce computations with the condition code, and to coalesce the first operand and the result of instructions like subtract which are actually two-address (to avoid the need for the Final Assembly Phase to emit code to copy the operand). C__LR also attempts to coalesce the operands of certain instructions with real registers in order to assign them to register pairs.

How is the interference graph actually colored? This is done by using the following idea, which is surprisingly powerful. If one wishes to obtain a 17-coloring of a graph G, and if a node N has less than 17 neighbors, then no matter how they are colored there will have to be a color left over for N. Thus node N can be thrown out of the graph G. The problem of obtaining a 17-coloring of G has therefore been recursively reduced to that of obtaining a 17-coloring of a graph G' with one node (and usually several edges) less than G. Proceeding in this manner, it is often the case that the entire graph is thrown away, i.e. the problem of 17-coloring the original graph is reduced to that of 17-coloring the empty graph. In fact, C__CLR gives up if the original graph cannot be reduced to the empty graph, and so spill code has to be introduced.

On the other hand, C__NP won't give up until it proves that the graph is not 17-colorable; it uses an urgency criterion to select nodes for which to guess colors, and backtracks if guesses fail. The urgency of a node is defined to be (the current number of uncolored neighbors that it has) divided by (the number of possible colors that are currently left for it). C__CLR runs in time linear in the size of the graph, while C__NP in the worst case is exponential, although this doesn't seem to happen often. The usual situation is that C__NP quickly confirms that graphs for which C__CLR gave up indeed have no 17-coloring. In fact, up to now in our experiments running actual PL/I source programs through the experimental compiler, in the handful of cases in which C__NP found a 17-coloring and C__CLR didn't, C__NP has achieved this by guessing without having to backtrack. In view of this situation, we have disabled the dangerous backtracking feature of C__NP. Furthermore, C__NP is only invoked when C__CLR fails and the user of the compiler has requested a very high level of optimization.

5. REPRESENTATION OF THE INTERFERENCE GRAPH

One of the most important problems in doing register allocation via graph coloring is to find a representation for the interference graph, i.e. a data structure, for which the 3 different kinds of operations which are performed on it—namely building the graph, coalescing nodes, and coloring it—can be done with a reasonable investment of CPU time and storage. In order to do these three different kinds of manipulations efficiently, it is necessary to be able to access the interference graph both at random and sequentially. In other words, it is necessary to be able to quickly determine whether or not two given names interfere, and to also be able to quickly run through the list of all names that interfere with a given name.

While building the graph one accesses it at random in order to determine whether an edge is already in the graph or must be added to it. While coloring the graph one accesses it sequentially, in order, for example, to count the number of neighbors that a node has (so that if this number is less than 17 the node can be deleted). And while coalescing nodes one accesses the graph both in a random and in a sequential fashion. For each *LR T,S* in the object code one must first check whether or not T and S interfere, which is a random access. If T and S don't interfere, one must then make all interferences of the form (S,X) into ones of the form (T,X). To do this requires sequential access to all names that interfere with S, and random access to see which interferences (T,X) are new and necessitate adding an edge to the graph.

Our solution to the problem of satisfying both of these requirements—fast random and sequential access—is to simultaneously represent the interference graph in two different data structures, one of which is efficient for random access, and the other for sequential access.

For random access operations we use an area *ITFS* in which the interference graph is represented in the form of a bit matrix. We take advantage of the fact that the adjacency matrix of the interference graph is symmetrical to halve the storage needed. The precise addressing rule is as follows. Consider two nodes numbered i and j, where without loss of generality we assume that i is less than or equal to j. Then these are adjacent nodes in the interference graph if the $i + j^2/2$ th bit of the area *ITFS* is a 1, and if this bit is a 0 they are not adjacent. (Here the result of the division is truncated to an integer.)

Since the adjacency matrix is usually quite sparse, and the number of bytes in the *ITFS* area grows roughly as a quadratic function $f(n) = n^2/16$ of the number n of nodes in the interference graph, for large programs it would be better if hashing were used instead of direct addressing into a bit matrix (somewhat more CPU time would be traded for much less main memory). Since the coefficient 1/16 of n^2 is small, if the program is not too large our bit matrix approach is ideal since it uses a small amount of storage and provides immediate access to the desired information.

For sequential access operations we keep in an area *LSTS* lists of all of the nodes which are adjacent to a given one, in the form of linked 32-byte segments. Each segment begins with a 4-byte forward pointer which is either 0 or is the offset in *LSTS* of the first byte after the next segment of the list. This forward pointer is followed in the segment by fourteen 2-byte fields for the adjacent nodes. For any given node *J*, the *J*th element of the vector *NXT* is either 0, or gives the offset in *LSTS* of the first empty adjacent-node field in the latest segment of the list of nodes which are adjacent to *J*, or, if the latest segment is full, it gives the offset of the first byte after the latest segment. All segments in a list are full (give all 14 adjacent nodes), except possibly the latest one.

6. DELETING INTERFERENCES AND PROPAGATING COALESCES

Consider a *LR T,S* at a point in the object program where besides *S* the names $L_1, L_2, \ldots$ are also live. Furthermore, suppose *S* was subsumed with L_i. We carefully avoided making *T* and *S* interfere, but it turns out that we erroneously made *T* and L_i interfere. This may have blocked our subsuming *T* and L_i, which in turn may have blocked other subsumptions. Our solution to this problem is as follows: After C_LR does all possible desirable coalesces, the entire interference graph is rebuilt from scratch, and typically there will be fewer interferences than before. We then run C_LR again to see if any of the coalesces which were impossible before have now become possible. This entire process is repeated either a fixed number of times (usually twice will do), or until no further coalesces are obtained. It turns out that in practice this is as fast and uses much less storage than the expensive data structure (described below) which directly supports deleting interferences and propagating coalesces.

Here is a more arcane example of a situation which requires interferences to be removed: If the source and target of a *LR* instruction are coalesced, then the *LR* no longer makes its source and target interfere with the condition code, nor does it make its target interfere with all names live at that point.

As it is of some theoretical interest, we now describe the alternate representation of the interference graph mentioned above. The graph has a count associated with each edge. This is called the interference count, and it is the number of program points at which the two computations interfere. As interferences are deleted, these counts are decremented, and if they reach zero then the two computations no longer interfere with each other.

Let us be more precise. In the framework necessary to directly propagate coalesces, the interference graph is best thought of as consisting of three sparse symmetric matrices. The first one gives the interference count of any two given names. The second one gives a pointer to the list of interferences that must be deleted if these two names are coalesced, and the third sparse matrix is boolean and indicates whether it is desired to coalesce the pair of names if their interference count hits zero. In practice these three sparse matrices can be combined into a single one. Hash tables are needed to provide random access to elements of the matrix, as well as pointers in both directions to chain rows and columns together for sequential access and to permit fast deletion.

The problem with this scheme for directly deleting interferences and propagating coalesces is the large amount of memory needed to represent the interference graph.

7. REPRESENTATION OF THE PROGRAM DURING COLORING

Here are some details about the way we represent the program in terms of names. In order to avoid rewriting the intermediate language text, it is actually left in terms of

symbolic registers. But it is supplemented by a vector NM__MAP giving the name of the result produced by each intermediate language instruction, and also by a "ragged" array giving for each basic block in the intermediate language text a list of ordered pairs (symbolic register live at entry to the basic block, corresponding name). And the name of a computation is represented as the index into the intermediate language text of an arbitrarily chosen canonical definition point for it. It is then possible to interpret one's way down a basic block maintaining at each moment a map from the symbolic registers into the corresponding names. C__ITF does this, keeping track of which names are live at each point, in order to build the interference graph. We also take advantage of this scheme to avoid rewriting the intermediate language text to reflect coalesces—only the ragged array and the NM__MAP vector are changed.

8. HANDLING OF MACHINE IDIOSYNCRASIES

It was mentioned above that one of the important advantages of the coloring approach to register allocation is that special case considerations can be taken care of by additional interferences in the graph. For example, the fact that the base register in a load instruction cannot be assigned to the register *R*0, is handled by making all names that are used as base registers interfere with *R*0. The fact that a call to a PL/I subprogram or a library routine has the side-effect of destroying the contents of certain machine registers is handled by making all names live across the call interfere with all registers whose contents are destroyed. Thus if j computations are live across the call and k registers are destroyed by it, a total of jk interferences are added to the graph to reflect this fact.

Although subtract is a destructive 2-address instruction, in the intermediate language subtract is 3-address and non-destructive. This is done to make possible a systematic uniform optimization process. Consider the intermediate language instruction *SR* *N*1,*N*2,*N*3 (*N*1 := *N*2 − *N*3). If *N*1 and *N*2 are assigned to the same register, then code emission in the Final Assembly Phase will emit a single instruction, subtract, for this intermediate language instruction. If not, it will emit *LR* *N*1,*N*2 followed by *SR* *N*1,*N*3. However, if *N*1 and *N*3 are assigned to the same register, then the Final Assembly Phase is in trouble, because copying *N*2 into *N*1 destroys *N*3. In order to avoid this code-emission problem, we make *N*1 and *N*3 interfere when building the interference graph.

A large set of special-purpose interferences has to do with intermediate language instructions involving the condition code (*CC*). The intermediate language ignores the fact that there is actually only one *CC*. The way we get around this is exemplified by contrasting the compare intermediate language instruction with the actual compare instruction. The intermediate language compare is three-address: two registers are compared, and bits 2 and 3 of the result register express the result of the compare. However compare always sets the bits of the *CC*, not those of an arbitrary register. Code emission in the Final Assembly Phase emits machine code for the compare intermediate language instruction in the following manner. If the result of the compare intermediate language instruction is assigned to the *CC*, then it merely generates a compare. If the result of the compare intermediate language instruction is assigned to one of the 16 general-purpose registers, then code emission generates a compare followed by a *BALR* which copies the contents of the *CC* into the indicated general-purpose register.

(A very special issue is how to deal with the fact that some instructions set the *CC* to reflect the sign of their result. For instance, subtract does this. In the Final Assembly Phase no code is emitted for a compare with zero of the result of a subtraction if it comes later in the same basic block as the subtract and none of the intervening instructions destroys the *CC*.)

9. TECHNIQUES FOR INSERTING SPILL CODE

Our techniques for inserting spill code are quite heuristic and *ad hoc*. The following notion is the basis for our heuristic. At any point in the program, the *pressure on the registers* is defined to be equal to the number of live names (it might be interesting to change this to the number of live colors) plus the number of machine registers which are

unavailable at that point because their contents are destroyed as a side-effect of the current instruction. Under the level two optimization compiler option, we insert spill code to immediately lower the maximum pressure on the registers in the program to 14. Under the level three optimization compiler option, successive trys are made. Spill code is inserted to bring the maximum pressure down to 20, then down to 19, etc., until a colorable program is obtained.

After inserting spill code it is necessary to recompute the def-use chains and the right number of names; there are generally more names than before. We also rerun dead code elimination, which has the side-effect of setting the operand-last-use flag bits in the intermediate language text—these flags are needed by C_ITF to keep track of which names are live at each point in the program. Note that since intermediate language text containing spill code is reanalyzed by optimization routines, and these routines only understand intermediate language written in terms of symbolic registers, the intermediate language text containing spill code must be correct in terms of symbolic registers as well as names.

How is spill code inserted to lower the register pressure? We attempt to respect the loop structure of the program and to put spill code in regions of the program which are not executed frequently. This is done in the following manner. First the decomposition of the program into flow-graphs is used bottom-up to compute the maximum register pressure in each basic block and each interval of all orders. As we do this we also obtain a bit vector of mentioned names for each basic block and interval. A *pass-through* is defined to be a computation which is live at entry to an interval but which is not mentioned (i.e. neither used nor redefined) within it. Clearly pass-throughs of high-order intervals are ideal computations to spill, i.e. to keep in storage rather than in a register throughout the interval for which they are a pass-through. We use the decomposition of the program into flow-graphs top-down in order to fix all those intervals in which the maximum pressure is too high by spilling pass-throughs.

We have explained how spill decisions are made for pass-throughs, but we have not explained how the spill code is actually inserted. This is done by using two rules. First of all, if a name is spilled anywhere, then we insert a store instruction at each of its definition points. And pass-throughs are reloaded according to the following rule: load at entry to each basic block B every name live at entry to B that is not spilled within B, but that is spilled in some basic block which is an immediate predecessor of B. These rules for inserting spill code are easy to carry out, but the other side of the coin is that they sometimes insert unnecessary code. However this unnecessary spill code is eliminated by a pass of dead code elimination which immediately follows.

Further remarks: Another idea used here is that some computations have the property that they can be redone in a single instruction whose operands are always available. We call such computations *never-killed*. An example of a never-killed computation is a load address off of the register which gives addressability to the DSA. Such computations are recalculated instead of being spilled and reloaded. Furthermore, if spilling pass-through computations doesn't lower the register pressure enough, as a last resort we traverse each basic block inserting spill code whenever the pressure gets too high.

Another approach to using recomputation as an alternative to spilling and reloading, is what we call the *rematerialization* of uncoalesced *LR* instructions. Here the idea is to replace a *LR* which can't be coalesced away by a recomputation that directly leaves the result of the computation in the desired register. (Of course, this should only be done if repeating the computation at this point still gives the same result.) Rematerialization usually decreases the pressure on the registers. Furthermore, assuming that all intermediate language instructions seen at this stage of the compilation are single-cost, replacing an uncoalesced *LR* by a recomputation cannot increase object program path lengths, and it sometimes actually shortens them. Thus there is a sense in which rematerialization is an optimization as opposed to a spill technique.

Rematerialization is most helpful when there are LR's into real registers. Typically this occurs when parameters are passed in standard registers. The standard parameter regis-

ters are destroyed over calls so the computation to be passed cannot be kept in the standard register over the call. The adverse consequence of this is most severe in loops where many loop constant parameters may be kept in registers and are loaded into standard parameter registers before each procedure invocation. Rematerialization tends to reduce the requirement for registers to hold loop constant parameters.

An entirely different approach to spilling might be based on the following observation. It is possible to have C__CLR make the spill decisions as it colors the interference graph. Each time C__CLR is blocked because it cannot delete any more nodes (all of them have more than 16 neighbors), it simply deletes a node by deciding to always keep that computation in storage rather than in a register. By increasing the granularity in the names, one could perhaps develop this into a more global and systematic approach to spilling than the one sketched above.

10. CONCLUSIONS

We have shown that in spite of the fact that graph coloring is NP-complete, it can be developed into a practical approach to register allocation for actual programs. It is also a pleasant surprise that coalescing nodes of the graph turns out to be an important optimization technique, and that machine idiosyncrasies can be handled in a uniform manner. We believe that our approach is able to pack computations into registers globally across large programs more cleverly than a hand-coder can or should. However, when not all computations can be kept in registers across the entire program, then the spill code that we insert sometimes leaves much to be desired.

Acknowledgements—The authors wish to state that the experimental compiler described herein could not have been completed without the efforts of the remaining members of their team: Richard Goldberg, Peter H. Oden, Philip J. Owens, and Henry S. Warren Jr. Although they were not directly involved with the compiler's register allocation scheme, this enterprise was very much a team effort to which all involved made essential contributions. We also wish to thank Erich J. Neuhold for reading an earlier version of this paper and suggesting improvements in the exposition.

REFERENCES

1. J. Cocke and P. W. Markstein, Measurement of program improvement algorithms. In *Information Processing 80* (Edited by S. H. Lavington), pp. 221–228. North-Holland, Amsterdam (1980).
2. F. E. Allen and J. Cocke, A program data flow analysis procedure. *Commun. ACM* **19**, 137–147 (1976).
3. A. P. Yershov, *The Alpha Automatic Programming System*. Academic Press, London (1971).
4. J. T. Schwartz, On Programming: An Interim Report on the SETL Project. Courant Institute of Math. Sciences, New York University (1973).
5. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison Wesley, Reading, MA (1974).

APPENDIX 1. THE "ULTIMATE" NOTION OF INTERFERENCE

The intuitive definition of the concept of interference is that two symbolic registers (i.e. results of computations) interfere if they cannot reside in the same machine register. Similarly, a symbolic register and a machine register interfere if the symbolic register cannot be assigned to that real register. Thus two registers interfere if there exists a point in the program, and a specific possible execution of the program for which:

1. Both registers are defined (i.e. they have been assigned by previous computations in the current execution);
2. Both registers will be used (note that we are considering a specific execution. Thus we mean use, **not** potential use);
3. The values of the registers are different.

It is clear that if these conditions are met, then assigning both symbolic registers to the same real register would be incorrect for that execution. It should also be clear that if any of the three conditions is not met, then such an assignment is correct at that point in the program, for that execution.

Of course, the criteria stated above are in general undecidable properties of the program. Thus a compiler must use more restrictive conditions of interference, potentially increasing the number of registers or amount of spill code required.

One particularly simple and sufficient criterion is that two symbolic registers interfere if they are ever simultaneously live (in the data flow sense). Consideration or experiment will show that this criterion is both expensive to compute and overly conservative. The difficulty is that application of this standard involves adding interferences for all pairs of live values at every point in the program. One could attempt to reduce this cost by observing how the liveness set changes during a linear reading of the program, so that only potentially new

interferences are added. Only growth of the liveness set need be taken into account, that is to say, the fact that (a) symbolic registers become alive on assignment, and (b) the set grows by union at a control flow join. The cost of computing the simultaneously alive criterion could be reduced by applying these observations.

However, one can safely take into account (a) all by itself, and ignore (b), the effect of control flow joins. This approach, which may be called point of definition interference, is not only inexpensive to compute, but omits certain apparent interferences for which both symbolic registers can never be defined simultaneously in any particular execution of the program. Thus we approximate interference by reading the program, using precomputed data flow information so that the set of live values is known at every computation. At each computation, the newly defined symbolic register is made to interfere with all currently live symbolic registers which cannot be seen to have the same value as the newly defined register.

APPENDIX 2. PROOF THAT ALL GRAPHS CAN ARISE IN REGISTER ALLOCATION

Consider the following program. It has declarations of the variables $NODE_i$, and there are just as many of these variables as there are nodes in the desired graph. For each edge $(NODE_i, NODE_j)$ in the desired graph, the corresponding variables are summed in order to make them interfere.

```
P: PROC(EDGE,MODE) RETURNS(FIXED BIN);
        DCL (MODE,EDGE,X) FIXED BIN;
        DCL LABEL(number-of-edges) LABEL;
        ...
        DCL NODEi FIXED BIN STATIC EXT;
        ...
        GO TO LABEL(EDGE);
        ...
        /*************************************/
        /* THE CALL PREVENTS OPTIMIZATION    */
        /* FROM MOVING THE LOADS OF NODEi,j. */
        /* THE ASSIGNMENT STATEMENT          */
        /* MAKES NODEi AND NODEj INTERFERE.  */
        /* JOINi,j CODE FRAGMENTS MAKE       */
        /* NAMES COME OUT CORRECTLY.         */
        /*************************************/
    LABEL(edge-number):
        CALL EXTERNAL_ROUTINEedge-number;
        X = NODEi + NODEj;
        IF MODE THEN GO TO JOINi;
                ELSE GO TO JOINj;
        ...
    JOINi:
        RETURN (X*NODEi);
        ...
END P;
```

About the Author—GREGORY J. CHAITIN was born in Chicago, Illinois, in 1947. In 1965, while he was an undergraduate at the City College of the City University of New York, he proposed a definition of randomness or patternlessness in terms of the size of computer programs, which he later developed into an information theoretic approach to Gödel's incompleteness theorem. He has been with IBM since 1967, first in Buenos Aires, Argentina, and since 1976 at the Thomas J. Watson Research Center in Yorktown Heights, New York. His present interests include theoretical physics and observational astronomy.

About the Author—MARC A. AUSLANDER received his A.B. degree in Mathematics from Princeton University in 1963. He has spent the majority of his career working on operating systems and programming languages at the IBM Thomas J. Watson Research Center.

About the Author—ASHOK K. CHANDRA was born in Allahabad, India, in 1948. He completed his B.Tech. in Electrical Engineering from the Indian Institute of Technology, Kanpur, in 1969. He subsequently received his M.S. and Ph.D. in Computer Science, the former at the University of California, Berkeley, in 1970, and the latter at Stanford University in 1973. Since then he has been on the research staff at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. His areas of interest include algorithms and complexity, program schemata, and data bases. He is the author of several publications in these fields.

About the Author—JOHN COCKE was born in Charlotte, North Carolina, in 1925. He received a B.S. in mechanical engineering from Duke University in 1946 and a Ph.D. in mathematics from Duke University in 1956. In 1956 he joined IBM in its Research Division. A main research interest has been in systems architecture, particularly hardware design and program optimization. He is an IBM Fellow and has been a visiting professor at M.I.T. and at the Courant Institute of Mathematical Sciences, N.Y.U. He is a member of the National Academy of Engineering.

About the Author—MARTIN E. HOPKINS was born in New York in 1933. He received his B.A. from Amherst College in 1957. After working 10 years for a computer software company in the field of compilers and software systems, he joined the IBM Research Division. His principal interests are in compilers, programming languages, and machine architecture.

About the Author—PETER W. MARKSTEIN was born in Vienna, Austria, in 1937. He received his S.B. from M.I.T. in 1958, and his Ph.D. in Computer Science from N.Y.U. in 1975. He has been with IBM as a Research Staff Member since 1959, where he has spent the majority of his time working on operating systems and compilers.