REGISTER ALLOCATION VIA COLORING

Gregory J. Chaitin
Marc A. Auslander
Ashok K. Chandra
John Cocke
Martin E. Hopkins
Peter W. Markstein

IBM T. J. Watson Research Center

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.

Key Words and Phrases: register allocation, optimizing compiler, graph coloring
CR Categories: 4.12, 5.32

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 J. Cocke and P. Markstein [1]. For background information on optimization, see [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 seventeen real machine registers, namely the sixteen general-purpose registers (R0-R15), 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 R0 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 J. Cocke, A.P. Ershov [3], J.T. 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 (see [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
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 \((k \text{ edges added versus } k(k+1)/2)\), 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_j 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 three 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.

Note: the node a is obtained as the 17-color of the node it is coalesced with.

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 inte-
fered 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 coalescences 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 \( LK T, S \) (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 

---

Here is an example of a typical situation in which one might wish to coalesce nodes. If there is a \( LK T, S \) (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 three 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 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 COALESCE

Consider a _LR T.S_ at a point in the object program where besides _S_ the names _L_1, _L_2,... 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 two times 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 is 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 maintain-
ing at each moment a map from the symbolic registers into the corresponding names. C_TTF
does this, keeping track of which names are live at each point, in order to build the interfer-
ence 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 R0, is handled by making all names that are used as base
registers interfere with R0. 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 interfer-
ences are added to the graph to reflect this fact.

Although subtract is a destructive two-address instruction, in the intermediate language
subtract is three-address and non-destructive. This is done to make possible a systematic
uniform optimization process. Consider the intermediate language instruction SR N1,N2,N3
(N1 := N2 − N3). If N1 and N2 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 N1,N2 followed by SR N1,N3. However, if N1 and N3 are
assigned to the same register, then the Final Assembly Phase is in trouble, because copying N2
into N1 destroys N3. In order to avoid this code-emission problem, we make N1 and N3
interfere when building the interference graph.

A large set of special-purpose interferences has to do with intermediate language instruc-
tions 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 interme-
diate 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 sixteen 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 \textit{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 interme-
diate 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 \textit{pass-through} is defined to be a compu-
tation 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 \textit{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 \textit{rematerialization} of uncoalesced \textit{LR} instructions. Here the idea is to replace
a \textit{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 registers 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.

APPENDIX I. 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;

ACKNOWLEDGMENTS

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,
REGISTER ALLOCATION & SPILLING VIA GRAPH COLORING

G. J. Chaitin
IBM Research
P.O.Box 218, Yorktown Heights, NY 10598

ABSTRACT

In a previous paper we reported the successful use of graph coloring techniques for doing global register allocation in an experimental PL/I optimizing compiler. When the compiler cannot color the register conflict graph with a number of colors equal to the number of available machine registers, it must add code to spill and reload registers to and from storage. Previously the compiler produced spill code whose quality sometimes left much to be desired, and the ad hoc techniques used took considerable amounts of compile time. We have now discovered how to extend the graph coloring approach so that it naturally solves the spilling problem. Spill decisions are now made on the basis of the register conflict graph and cost estimates of the value of keeping the result of a computation in a register rather than in storage. This new approach produces better object code and takes much less compile time.

1. INTRODUCTION.

This paper is a progress report on part of the work on an experimental 32-bit minicomputer that has been pursued at the IBM Watson Research Center for the past several years (1). One of the main goals of this project is to attain very high performance by using a very simple and regular CPU on a single chip. In the current design the CPU contains thirty-two 32-bit general-purpose registers. The instruction set consists mostly of 3-address register to register operations, each of which executes in a single machine cycle. References to storage are through separate load and store instructions. In order to achieve the high performance goals of this project, it is essential to take advantage of the high-speed registers and keep data there rather than in storage as much as possible, in order to avoid having the very fast CPU waiting for the storage subsystem.

Another of our principal project goals is that this machine be programmed only in a high-level language, which is a PL/I variant. This version of PL/I is essentially a subset of PL/I which has been chosen because programs which remain in the subset are good subjects for an optimizing compiler. It was our hope, which we believe that we have largely achieved, that by systematically utilizing the best available optimizing compiler technology, object code produced by the compiler would be competitive with hand-coded assembly language, and there would no longer be much incentive to do programming at the machine language level. The simplicity and regularity of the instruction set of our experimental minicomputer not only enables its CPU to be simpler, smaller, and faster, but it also simplifies the design of the compiler for our PL/I subset.

Allen (2) discusses in general terms our approach to compiler design, contrasting it with other compiler efforts. More specific information on the optimization techniques we use is contained in the paper by Cocke and Markstein (3). Our previous paper on register allocation (4) details our approach at that time. Here we shall paint the picture in broader brushstrokes, emphasizing the improvements which have been made since (4).

2. OVERVIEW OF REGISTER ALLOCATION.

The register allocation phase of the compiler stands between the optimization phase and the final code assembly and emission phase. When the intermediate or internal language (IL) enters register allocation, it is written assuming a hypothetical target machine having an unlimited number of high-speed
general-purpose CPU registers. It is the responsibility of the optimization phase to eliminate references to storage by keeping data in these registers as much as possible. It is the responsibility of the register allocation phase to map the unlimited number of symbolic registers assumed during optimization into the 32 registers which are actually present in the CPU. In order to do this, it may be necessary to add code to the program to spill computations from registers to storage and later reload them. We shall refer to this as spill code.

Register allocation consists of the following main parts: usedef chaining plus getting the right number of names, building the interference graph, coalescing nodes, attempting to find a 32-coloring of the graph, and if one cannot be found, modifying the program and its graph until a 32-coloring is obtained. We now briefly describe each of these steps.

The first step in processing the program is to use well-known optimizing compiler techniques to do a global data-flow analysis. We must know which symbolic registers are live at each point in the IL program. This is done by indicating at the beginning of each basic block which computations are live going into it, and by marking each operand of each instruction in the IL to indicate if it goes dead.

Next the register interference graph is built. It contains one node for each symbolic register in the IL. Two nodes are adjacent, that is to say, two symbolic registers conflict or interfere, if they are ever live simultaneously, more precisely, if one of them is live at a definition point of the other. Thus a 32-coloring of the interference graph corresponds to a permissible register allocation, and if the chromatic number of the graph is greater than 32, spill code is necessary.

After the interference graph is built, unnecessary register copy operations are eliminated by coalescing or combining the nodes which are the source and targets of copy operations. Of course, this can only be done if these nodes do not interfere with each other. Once two nodes have been coalesced, they must get the same color and be allocated to the same register, and the copy operation becomes unnecessary. This copy-eliminating optimization is known as subsumption or variable propagation in the optimizing compiler literature.

Next we use the following seemingly trivial observation in order to construct a 32-coloring. Assume we wish to find a 32-coloring of a graph G having a node N of degree less than 32. Then G is 32-colorable if and only if the reduced graph G' from which N and all its edges have been omitted is 32-colorable. So our algorithm reduces the interference graph by throwing away all nodes of degree less than 32. This often cascades until the entire graph is thrown away, that is, until the problem of 32-coloring the original graph is reduced to that of 32-coloring the empty graph. Nodes are then added back on in the inverse order that they were removed, and as each node is restored, a color is picked for it. Experiments have shown that this algorithm produces excellent results. It is easy to see that it can be implemented in such a way that its running time is linear in the size of the graph; a full NP-complete algorithm for obtaining 32-colorings is of course out of the question. Note that the coloring algorithm fails only if at some point the reduced graph G' only has nodes of degree 32 or greater.

What can we do if the algorithm is blocked in this way? If the above procedure fails to produce a 32-coloring, we must add spill code and modify the interference graph until a 32-coloring is obtained. In fact this is essentially done by the same algorithm used to obtain 32-colorings. It is not far from the truth to say that the algorithm for obtaining 32-colorings will either do so or will modify the program and its graph until it can. If the algorithm is blocked because all nodes are of high degree, it will pick a node to delete from the graph in order to unblock things. Deleting this node will hopefully produce a cascade of nodes of degree less than 32 and enable the coloring algorithm to finish or at least to advance a considerable distance towards the empty graph. Deleting the node from the graph corresponds to making the decision that the computation which it represents will be spilled, that is, kept in storage rather than in a register. This means that each spill decision implies adding code to the IL to store a spilled computation at each of its definition points and to reload it at each of its use points.

3. THE INTERFERENCE GRAPH.

The register interference graph is a large and massive data structure, and it is important to represent it in a manner that uses as little storage as possible consistent with the ability to process it at high speed. We use a dual representation: a bit matrix and adjacency vectors.

The bit matrix for an N-node interference graph consists of a symmetric matrix of N bits by N bits. The bit at row I and column J is a 1 if and only if nodes I and J are adjacent. This bit matrix is excellent for random access to the interference graph, but it is quite sparse, and it is too time consuming to use it for sequential access to the graph. Thus it is supplemented by keeping for each node in the graph a vector giving the set of nodes which are adjacent to it. The length of this vector is equal to the degree of the node.
The algorithm for building the interference graph is therefore a two pass algorithm. In the first pass over the IL the bit matrix is used to calculate the degree of each node. Then the N adjacency vectors are stored allocated, and a second pass is made over the program IL in order to fill in the adjacency vectors. We believe that this two-pass approach is much better than the one-pass segmented scheme described in (4); non-segmented adjacency vectors can be processed more simply and quickly.

4. SUBSUMPTION.

Our approach to coalescing nodes of the graph in order to eliminate unnecessary register copy operations is also different from that in (4). As we coalesce nodes, we keep the bit matrix current, and chain together the interference vectors of nodes which have been coalesced. We do not attempt to eliminate entries in the adjacency vectors which have become duplicates due to node coalesces. The resulting interference graph is therefore not suitable for use by the coloring algorithm, which deduces the degree of a node from the length of its adjacency vector and is disturbed by duplicate entries.

In order to obtain a new interference graph reflecting the coalesces, the program IL is rewritten in terms of coalesced symbolic registers, and the two-pass interference graph building algorithm is re-run on the new and somewhat shorter IL. It may then be possible to eliminate register copy operations that could not previously be eliminated by performing further node coalesces (see (4)). So we continue building the graph and coalescing, until no more desirable coalesces are found to be possible and the graph is left unspoiled by coalescing. In practice two or three iterations will do.

5. SPILLING.

In the overview we briefly described how spill decisions are made from the interference graph. That description omitted two very important points: which node is chosen to spill when the coloring algorithm is blocked, and the fact that the interference graph must be rebuilt after spill code is inserted. Let us deal with the second point first.

In order to make spill decisions from the graph, it is important to keep the graph and program in step as spill decisions are made. However this can only be done in an approximate manner. Spilling a computation is not the same as eliminating its node from the graph, for it is still necessary to reload it at each use and to store it away at each definition point. So that what actually usually ought to happen is that one node corresponding to a globally live computation would have to be replaced by several new nodes corresponding to computations which are only live momentarily. However it is too expensive to proceed in this more exact manner.

Thus after all spill decisions are made, it is necessary to insert spill code in the program IL, rebuild the interference graph, and then reattempt to obtain a 32-coloring. This will usually succeed, but it is sometimes necessary to loop through this process yet again, adding a little more spill code, until a 32-coloring is finally obtained. In practice we have not found the fact that the process of inserting spill code and rebuilding the interference graph must be iterated until a coloring is obtained to be a problem. Convergence is usually quite rapid, and the compile time is dominated by that required to build the graph the first time - all successive graphs are substantially smaller.

The other point we must address is how to choose a node to spill when the coloring algorithm is blocked. Obviously one wishes to insert as little spill code as possible. More precisely, we attempt to increase the execution time of the object program as little as possible. In order to estimate execution times, we assume that all instructions execute in one machine cycle and that each instruction in a loop is executed ten more times than it would be if it were outside the loop.

While making spill decisions, we supplement the interference graph with a table which gives for each node in the graph an estimate of what it would cost to spill it. Then when the coloring algorithm is blocked, it decides to spill that node, among those remaining, for which the cost of spilling it divided by its current degree is as small as possible.

These cost estimates are made as follows. The cost of spilling a node is defined to be the increase in execution time if it is spilled, which is approximately equal to the number of definition points plus the number of uses of that computation, where each definition and use is weighted by its estimated execution frequency. The cost estimates also take into account the fact that some computations can be redone instead of spilling and reloading them, and that if the source or target of a register copy operation is spilled then the copy operation is no longer necessary. In fact spilling a computation that can be recomputed and which is used as the source of a register copy operation can have negative cost!

Finally a somewhat subtle point must be mentioned, which gives some local intelligence to our global algorithm. Suppose that there are several uses of a spilled computation within a single basic block. Pro-
ceeding naively as outlined above, one would reload it at each use. However if no computations go dead between the first use and the last use, then one might as well only insert a load before the first use, and keep the computation in that register until the last use. Similarly, if a computation is local to a basic block, and if nothing goes dead between its definition and its last use, then spilling the computation cannot help to make the program colorable. We therefore set the cost of spilling this node to infinity. This also keeps our algorithm from spilling computations that have already been spilled.

6. CONCLUSIONS.

By now thousands of programs have been run through the compiler, and it is regularly bootstrapped through itself. Based on this experience with it we can conclude that these register allocation techniques seem to take better advantage of the speed potential of using registers in preference to storage than previous approaches (see (3)). The cost in terms of compile time also seems reasonable: register allocation including spilling now takes an amount of compile time comparable with the more traditional optimization algorithms described in (3). However it must be admitted that a fair amount of virtual storage is needed to hold the program IL and interference graph in core during register allocation.

REFERENCES


APPENDIX

The following program written in SETL (see (6)) outlines in executable form the main ideas and algorithms presented in this paper.

program register_allocation;

var il; $ il is an ordered sequence of instructions

$ Each instruction is a triple (opcode,def,use),
$ where opcode is a character string,
$ & 'bb', 'copy', 'spill', and 'reload'
$ have special meanings.
$ Def is a tuple of outputs, each a pair (reg,dead), where
$ reg is a symbolic register and dead is a true/false value
$ indicating whether or not reg goes dead.
$ Use is a tuple of inputs, each a pair (reg,dead), where
$ reg is a symbolic register and dead is a true/false value
$ indicating whether or not reg goes dead.

$ Basic block header pseudo-ops:
$ 'bb' has def for each symbolic register live at entry to the basic block.
$ 'bb' has as use the estimated execution frequency of the basic block
$ (floating point number).
var graph;  $ register interference graph = set of edges,
            $ each edge being specified by the set of its endpoints
var colors;  $ set of available colors (machine registers)
var cost;  $ gives estimated cost of spilling each symbolic register
var spilled;  $ set of spilled symbolic registers

read( il );
read( colors );
if color_Il() = Ω then
    estimate_spill_costs;
    decide_spills;
    insert_spill_code;
    color_Il;
end if;
print( il );

proc color_Il;  $ build graph, coalesce, & color
build_graph;
coalesce_nodes;
coloring := color_graph( graph, registers_in_il() ) ;
if coloring = Ω then return Ω ; end if;
rewrite_il( coloring );
return( coloring );
end proc color_Il;

proc build_graph;  $ build the register interference graph
graph := { } ;
(for [ opcode, def, use ] ∈ il)
if opcode = 'bb' then
    liveness := { } ;
    (for [ reg, dead ] ∈ def | not dead)
        liveness(reg) := liveness(reg) + 1 ;
    end for;
else
    (for [ reg, dead ] ∈ use | dead)
        liveness(reg) := 1 ;
        if liveness(reg) = 0 then liveness(reg) := Ω ; end if;
    end for;
    (for [ reg, dead ] ∈ def)
        graph +:= { reg, x }
        : x ∈ domain liveness | x ≠ reg ;
        if not dead then
            liveness(reg) := liveness(reg) + 1 ;
        end if;
    end for;
end if;
end for;
end proc build_graph;
proc coalesce_nodes;  $ coalesce away copy operations
  (while [ opcode, def, use ] ∈ il
    | opcode = 'copy'
    and ( source := use(1)(1) ) ≠ ( target := def(1)(1) )
    and [ source, target ] ∉ graph)
    f := [ [ source, target ] ]
    graph := [ [ f(x) ? x : x ∈ edge ] : edge ∈ graph ];
  rewrite il( f );
end while;
end proc coalesce_nodes;

proc color_graph(g,n);  $ color the graph with edges g & nodes n
  if n = [ ] then return [ ]; end if;
  if not ∃ node ∈ n | # neighbors(node,g) < # colors
    then return Ω; end if;
  coloring := color_graph( [ edge ∈ g | node ∉ edge ],
    n - [ node ] );
  if coloring = Ω then return Ω; end if;
  coloring(node) :=
    arb( colors - [ coloring(x) : x ∈ neighbors(node,g) ] );
  return coloring;
end proc color_graph;

proc estimate_spill_costs;  $ estimate cost of spilling each register
  cost := [ ];
  (for [ opcode, def, use ] ∈ il)
    if opcode = 'bb'
      frequency := use(1)(1) ;
    else
      (for [ reg, ] ∈ def + use)
        cost(reg) := cost(reg) ? 0 + frequency ;
      end for;
    end if;
  end for;
end proc estimate_spill_costs;

proc decide_spills;  $ make spill decisions
  g := graph;
  n := registers_in_il();
  spilled := [ ];
  (while n ≠ [ ] )
    if not ∃ node ∈ n | # neighbors(node,g) < # colors then
      node :=
        arb { x ∈ n | cost(x) = min/ { cost(y) : y ∈ n } } ;
      spilled +=: { node } ;
    end if;
  g := [ edge ∈ g | node ∉ edge ];
  n -=: { node } ;
end while;
end proc decide_spills;
proc insert_spill_code; $ insert spill & reload instructions in il
newil := [];
(for [ opcode, def, use ] ∈ il)
  if opcode = 'bb' then
    newil := newil ++ ['bb',
      [ [ reg, dead ] ∈ def | reg ∉ spilled, use ]];
  else
    before := after := newdef := newuse := [];
    (for [ reg, dead ] ∈ use)
      if reg ∈ spilled then
        newuse := newuse ++ [ [ newreg := newat, true ] ];
        before := before ++ ['reload',[ [ newreg, false ] ]];
      else
        newuse := newuse ++ [ [ reg, dead ] ];
      end if;
    end for;
    (for [ reg, dead ] ∈ def)
      if reg ∈ spilled then
        newdef := newdef ++ [ [ newreg := newat, false ] ];
        after := after ++ ['spill',[ [ newreg, true ] ]];
      else
        newdef := newdef ++ [ [ reg, dead ] ];
      end if;
    end for;
    newil := before ++ [opcode,newdef,newuse] + after;
  end if;
end for;
il := newil;
end proc insert_spill_code;

proc rewrite_il(f); $ apply function f to each register in il
il := [opcode,
  [ f(reg) ? reg, dead ] : [ reg, dead ] ∈ def,
  [ f(reg) ? reg, dead ] : [ reg, dead ] ∈ use ];
] : [ opcode, def, use ] ∈ il ];
end proc rewrite_il;

proc registers_in_il; $ returns set of symbolic registers in il
return
{ reg : [reg,−] ∈ [] + / { def+use : [−,def,use] ∈ il }
  ∥ (type reg) ≠ 'REAL' };
end proc registers_in_il;

proc neighbors(x,g); $ x is node, g is set of edges
return { y ∈ {} + / g | {x,y} ∈ g };
end proc neighbors;

end program register_allocation;
APPENDIX. The IL.

The following PL/I program illustrates the IL used during register allocation. Its chief advantage is that it allows algorithms to quickly loop through all the registers in an instruction, and that it can be quickly rewritten to reflect register renamings. Also note that multiple results are permitted, as well as an unlimited number of input operands.

register_rename: proc(eq);

dcl

  eq(*) fixed bin, /* map from old to new register names */
  x offset(il), /* x points to current instruction */
  i fixed bin, /* i points to current operand */
  il area(*) ctl ext, /* intermediate language for proc */
  proc_begin offset(il) ext, /* offset in il of beginning of proc */

1 instruction based(x), /* current instruction in il for proc */
  2 next_instruction offset(il), /* forward chain */
  2 last_instruction offset(il), /* backward chain */
  2 source_statement fixed bin, /* for listings & tracebacks */
  2 opcode fixed bin, /* & pseudo-ops like label definition */
  2 defs fixed bin, /* index of last output operand */
  2 uses fixed bin, /* index of last input operand */
  2 kills fixed bin, /* index of last operand destroyed */
  2 operand(i refer(kills)), /* def's, then use's, then kill's */
    3 register fixed bin(31), /* or integer value or dictionary ref */
    3 operand__type fixed bin, /* see list of types below */
    3 dead bit, /* operand's value is no longer alive */

/* operand types: */

  o__null fixed bin ext, /* no operand in this position */
  o__symreg fixed bin ext, /* symbolic register (computation) */
  o__dictref fixed bin ext, /* dictionary reference (storage) */
  o__integer fixed bin ext; /* immediate value (displacement etc) */

do x = proc__begin repeat next_instruction /* look at each instruction */
  until(next_instruction = proc__begin);
  do i = 1 to kills; /* look at each operand */
    if operand__type(i) = o__symreg then /* if it is a register, */
      register(i) = eq(register(i)); /* then rename it */
  end;
end;

end register_rename;