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 storage 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 unspoilt 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. Proceeding 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

1. "The 801 minicomputer," G. Radin, Proceedings of the ACM Symposium on Architectural Support for Programming Languages and Operating Systems, March 1-3, 1982, Palo Alto, California.

2. "The history of language processor technology in IBM," F.E. Allen, IBM Journal of Research and Development 25 (1981), pp. 535-548.

3. "Measurement of code improvement algorithms," J. Cocke, P.W. Markstein, Information Processing 80, S.H. Lavington (ed.), North-Holland, Amsterdam, 1980, pp. 221-228.

4. "Register allocation via coloring," G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P.W. Markstein, Computer Languages 6 (1981), pp. 47-57.

5. "Optimization of range checking," V. Markstein, J. Cocke, P. Markstein, this Proceedings.

6. "Higher Level Programming: Introduction to the Use of the Set-Theoretic Programming Language SETL," R.B.K. Dewar, E. Schonberg, J.T. Schwartz, Courant Institute, New York University, 1981.

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) ? 0 + 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) ? 0 + 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' then
            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 +:= [['bb',
            [[reg,dead] ∈ def | reg ∉ spilled],use]];
        else
            before := after := newdef := newuse := [ ] ;
            (for [ reg, dead ] ∈ use)
                if reg ∈ spilled then
                    newuse +:= [[ (newreg:=newat), true ]] ;
                    before +:= [['reload',[[newreg,false]],[]]] ;
                else
                    newuse +:= [[ reg, dead ]] ;
                end if;
            end for;
            (for [ reg, dead ] ∈ def)
                if reg ∈ spilled then
                    newdef +:= [[ (newreg:=newat), false ]] ;
                    after +:= [['spill',[],[[newreg,true]]]] ;
                else
                    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;
```