Graph Coloring Register Allocation for Processors with Multi-Register Operands

Brian R. Nickerson
int$_e$l Corporation
5200 NE Elam Young Parkway
Hillsboro, OR. 97124

Though graph coloring algorithms have been shown to work well when applied to register allocation problems, the technique has not been generalized for processor architectures in which some instructions refer to individual operands that are comprised of multiple registers. This paper presents a suitable generalization.

1. Introduction

Graph coloring, as applied to register allocation, is a process whereby a compiler attains a good mapping between the arbitrarily large demand for registers that an input program may have, and the few registers implemented in the target processor. Much of the demand is handled by mapping several of the input program's **virtual registers** with non-overlapping lifetimes onto a single physical register. This is done over each of the available physical registers. Excess virtual registers are handled by spilling them to secondary storage (i.e. usually memory; sometimes secondary registers).

Hsu gives a good summary of graph coloring in his PhD dissertation (Hsu 87, pp 61-62):

> ...graph coloring ... was successfully developed and implemented by Chaitin. The global register allocation is formulated 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 two quantities **interfere** with each other, that is, if they are simultaneously live at some point in the program. The problem is to assign different colors (registers) to connected nodes. It is hard to obtain an optimal coloring, but the implementation showed [that] a fast heuristic method for assigning colors to these graphs generally resulted in a very good assignment. When the compiler cannot color the register conflict graph, it must add code to spill some nodes. Spill decisions are made on the basis of the register conflict graph and cost estimates of the value of keeping the variable in a register rather than in memory."

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1990 ACM 0-89791-364-7/90/0006/0040 $1.50

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

Chaitin et al developed register allocation by coloring in 1980 (CACCHM 81). The following year, Chaitin refined the criterion for selecting which virtual registers to spill to secondary storage (Chai 82). Whereas in the first paper, selection of a spill candidate was ad hoc, the refinements were to base the selection on an estimate of which candidate would cost the least to spill and would have the greatest impact toward reducing the **register pressure** so that other virtual registers might not need to be spilled.

In the SIGPLAN '89 Conference, Pinter et al and Briggs et al provided valuable refinements to graph coloring. Pinter (BGGKMNP 89) offered improved heuristics for the selection of spill candidates. He also detailed "a specially tuned greedy heuristic for determining the order of deleting (and hence coloring) the **unconstrained**[1] [nodes]" of the register interference graph. Programs tend to be colorable with fewer physical registers when Pinter's heuristic is used in place of the earlier method, where unconstrained nodes were deleted in arbitrary order. Also, Pinter suggested an optimization he referred to as **cleaning**, in which an attempt is made to reduce the amount of spill code produced for a spilled entity. To do so, the compiler inserts only one load or store of the spilled entity per basic block, instead of a load or store at each reference to the entity.

The contribution of Briggs et al (BCKT 89) was to modify graph coloring in such a way that less spilling is required. They observed that deleting a constrained node from the register interference graph does not have to condemn that node to being spilled. While unconstrained nodes are guaranteed to be colorable into the physical registers, some or all of the constrained nodes also might fit into the physical registers. Therefore, pruning a constrained node from the interference graph should only set a flag on the node that indicates it may need to be spilled. It is left up to the coloring routine to either ratify or reset that flag.

Briggs also suggested that if unconstrained nodes are pruned from the interference graph in ascending order of

[1] A node is unconstrained when the number of coloring constraints imposed upon the node (termed the degree of the node) is less than the

degree, an improvement could be applied to the pruning function which would make it faster. While this is true, it is contrary to Pinter's greedy pruning heuristic. In the question and answer session following Briggs' presentation it was agreed that better implementations would prefer Pinter's greed over Briggs' speed.

None of these sources has dealt with the difficulties of applying the graph coloring technique to hardware architectures in which some instructions accept individual operands that are comprised of multiple contiguous registers. Chow and Hennessy (ChHe 84), who implemented a similar register allocation technique called priority-based coloring, reported, "The problem of allocating overlapping registers of different sizes [another way of saying multiple contiguous registers] have [sic] not been considered.... It would be interesting to see to what extent priority-based coloring [and, I submit, graph coloring] can be adapted to such situations."

This paper explains a set of extensions sufficient to solve the problem. It is assumed that the reader already has basic familiarity with register allocation by graph coloring. If this is not the case, the reader is encouraged to refer to the sources provided in the final section.

2. Background

2.1. Motivation

Several **int$_e$l** architectures have instructions in which one or more operands are each comprised of a contiguous cluster of registers. To address these operands, the identity of the cluster's low-order register (referred to as the **0-mate**) is placed in the appropriate operand field of the instruction; the addresses of the other **cluster-mates** are inferred in the obvious way.

In register allocation, the difficulty in dealing with register clusters stems from the fact that the clusters must be hosted by contiguous registers. It is a bad coloring that would, for example, assign the 0-mate of a two word cluster to register *r4*, and the 1-mate of the same cluster to register *r9*.

It would be easier for compiler writers if the multi-register operands were handled by widening the instructions so that they explicitly address all the constituent registers. Then the existing register coloring algorithms could be employed, placing the cluster constituents in whatever registers were available in the region where the cluster is live. Both the added width of the instruction and the greater complexity of dereferencing the extra register addresses makes this suggestion unpalatable from a hardware design perspective.

number of colors (physical registers).

Another alternative is to eliminate the instructions that make use of multi-register operands, i.e. the RISC approach. The trade-off for doing this is that the instructions that require multi-register operands become sequences of other less demanding instructions. These instruction sequences might be slower in their own right, and the loss of code density can slow the operation further.

Therefore, in light of the realities of hardware design, compilers must be endowed with effective register allocators. If register coloring is done, one way to handle multi-register clusters is to "pre-color" them to contiguous physical registers, then use the standard coloring algorithms on the remaining, single-register clusters (which, after all, tend to comprise the majority of the program's register demand). A drawback of this approach is that it raises the priority of multi-register clusters to a level artificially higher than the priority of single-register clusters. Another drawback is that it places an unnecessarily strong constraint on the locations to which the multi-register clusters must be assigned (i.e. to the specific registers that the pre-colorer assigns). As a further consequence, it places an unnecessarily strong constraint on the locations to which the single-register clusters must *not* be assigned.

Another approach might be to "post-color" the multi-register clusters, basicly giving them what's left after coloring the single-register clusters. This method suffers similar drawbacks as the pre-coloring approach.

This paper details an approach in which the multi-register clusters are colored simultaneously and harmoniously with coloring the single-register clusters. It turns out that this is not difficult to do. Only seven weeks of the author's time were required to design and implement this approach.

2.2. Hardware Characteristics

As indicated above, the processor architectures of interest to the author require multi-register clusters to be assigned to contiguous registers. More specifically, clusters must also be aligned on "natural" boundaries. That is, when assigning a two-word cluster, the 0-mate must be placed in an even-numbered register while the 1-mate must be placed in the odd-numbered register immediately subsequent to the 0-mate.

Moreover, in the **int$_e$l** 80960 family of microprocessors (Intel 89), where three-word and four-word operand clusters are defined for some instructions, the 0-mate of these clusters must be aligned to a register number divisible by four. The processor architecture imposes alignment constraints upon multi-word clusters, because doing so simplifies the "scoreboard" logic (see Intel 89).

2.3. Software Characteristics

One might be content to model register cluster usage subject to the assumption that the individual registers in a cluster all have exactly the same lifetimes. This would be entirely sufficient to model elementary types such as double precision floating point. It would also be feasible, though timid, to model aggregate data types, i.e. structs.

Such a model would have the advantage of simplicity, since the compiler simply tracks whether one cluster interferes with another, instead of tracking what portions of the one interfere with what portions of the other. But this model has the disadvantage of being overly pessimistic with the assignment of the precious physical register resource.

Figure 2-1A shows a sample code sequence that would benefit from the more aggressive model. In compiling this code, it is advantageous to fetch both elements of *y* with a single two-word load instruction, store the two words to *x*, and then maintain the integrity of the register containing *y.a* until it is stored to *r*. While keeping this register around, the register that contained *y.b* can be used for something else, such as the transfer of *q* to *p*. So, with this aggressive model, the source code shown in Figure 2-1A is compiled to the object code shown in Figure 2-1B, which uses only two registers. With the simpler model, three registers would be used instead.

```
A -- Source Code:

struct {
   int a;
   int b;
} x, y;
int p, q, r;
   ...
x = y;
p = q;
r = y.a;
   ...


B -- Object Code

ldl _y,G0     # load Y into G0 and G1
stl G0,_x     # store G0 and G1 to X
ld  _q,G1     # load Q into G1
st  G1,_p     # store G1 to P
st  G0,_r     # store reg that still
              # has Y.A and X.A to R
```

Figure 2-1 An example favoring aggressive model

3. The Algorithm

Register allocation is performed after all optimization phases and after the optional instruction scheduler phase. Register allocation attempts to fit (by graph coloring) the arbitrarily large number of virtual registers referred to in the intermediate program representation, to the few physical registers in the actual hardware.

Sometimes the register allocator cannot successfully fit the virtual registers into the limited number of physical registers. When this happens, the register allocator inserts spill code into the intermediate program representation, and causes control to reiterate some of the earlier phases. In particular, the register allocator itself must be reiterated. But before doing so, it is probable that loop invariant code removal, for example, should be reiterated. Also, if code scheduling is being done, it should be reiterated.

Chaitin (Chai 82) indicates the reason for reiterating the register allocator, as follows:

> 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 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 [intermediate language form], rebuild the interference graph, and then reattempt to obtain a [coloring].... Convergence is usually quite rapid.

Though reiterating the register allocator is required, reiteration of the earlier optimization phases is not. In fact, if the earlier phases are always reiterated, it may be possible to contrive a situation where the register allocator never succeeds in coloring the registers. On the other hand, if we never reiterate the earlier phases, the spill code is likely to noticeably damage the quality of the object code, beyond what is necessary. For example, we may be able to move the load of a loop-invariant spilled register out of a loop, but that is only possible if we reiterate the phase that does loop-invariant code removal.

Pinter's work (BGGKMNP 89) supports our decision to reiterate earlier optimizations. He applied a technique that he called "cleaning", in which he limited spill code insertion to a single load or store per basic block. Moreover, he wrote "It is evident that limiting the action of the cleaning routine to the scope of single basic blocks is an artificial constraint, and that it could be extended to larger regions of the program, for example, the elimination of loads and stores from loops and/or moving them out of a loop."

We have chosen to reiterate those optimization phases which will effect both the per-block cleaning that Pinter performed and the loop optimizations he suggested. We've also chosen to reiterate instruction scheduling. Empirically, Pinter found that cleaning was best applied to the spill code resulting from the first two iterations of the register allocator and thereafter suppressed. At the time of this writing, we have not yet determined whether that empirical control is appropriate for the other optimizations we intend to reiterate.

Aside from coloring the virtual registers into the physical registers, the register allocator performs an optimization called register **subsumption** (also called **coalescing**). The earlier phases of the compiler are encouraged to be quite loose and liberal about their use of virtual registers, including moving received parameters and called-function results out of physical registers and into virtual registers immediately, and likewise moving arguments and current-function results into physical registers just before needed. The register allocator attempts to eliminate the copious register move operations by "subsuming" their source and destination registers.

Register subsumption has the negative effect of imposing greater constraints on the coloring of the virtual registers to the physical registers. Empirically, these negative effects are small compared to the gain obtained by the elimination of unnecessary moves. Chaitin's (Chai 82, CACCHM 81) work suggests that most subsumption opportunities will be exploited in two or three iterations of the register allocator.

The following steps are performed by the register allocator:

(1) Build interference graph in matrix form.
(2) Perform register subsumption.
(3) Convert the interference graph from matrix form to vector form.
(4) Prune the interference graph.
(5) Color the interference graph, introducing spill code as necessary.
(6) Return a status indicating whether reiteration is necessary.

3.1. The Matrix Form of the Interference Graph

Chaitin (CACCHM 81) suggested that the interference graph should be stored as both a bit matrix for random access operations and as segmented lists for sequential access. He pointed out that building the graph requires random access, the pruning and coloring operations require sequential access, and the subsumption optimization requires both kinds of access.

Rather than keep both forms simultaneously, our implementation starts with the matrix form for building the interference graph and attempting the subsumption optimizations, then translates the matrix form to adjacency vectors for the pruning and coloring phase. This approach simplifies the graph-building step, because adding edges to the graph does not require updating a segmented or linked lists representation. Moreover, at the time the matrix is translated to lists, the number of edges emanating from each node of the graph is fixed, so the lists can be simple vectors.

Casual conversation with other compiler implementors has indicated that others have done much the same thing. These same conversations indicate an awareness that (as Chaitin points out) the bit matrix "grows roughly as a quadratic function $f(n) = n^2/16$ [bytes] of the number n of nodes in the interference graph, [and therefore] for large programs it would be better if hashing were used instead of direct addressing into a bit matrix." (ibid) As many others have done, we've chosen to implement the bit matrix rather than the hashing method.

Chaitin also points out that "[one] can take advantage of the fact that the adjacency matrix of the interference graph is symmetrical to halve the storage needed." (ibid) In particular, the major diagonal of the bit matrix divides the matrix into two reflexively equivalent triangles, so only one of them must be implemented.

3.2. Building the Interference Graph

The ability to allocate clusters of registers is an important issue for the **int$_e$l** 80960 microprocessor, where the general registers sometimes are allocated in aligned pairs, triples, or quadruples, and for the **int$_e$l** 80860 microprocessor where the floating point registers sometimes are allocated in aligned pairs or quadruples.

Two examples are presented to visualize the problems with respect to allocating register clusters. In the first example, suppose that the physical register set consists of eight contiguous registers numbered 0 through 7. Clusters consisting of three cluster-mates (i.e. triples) must be aligned on four-word boundaries, (i.e. must start at register 0 or register 4). Clusters consisting of two cluster-mates (i.e. pairs) must start at any even register. Remaining registers of a cluster must be consecutively numbered from the first (the 0-mate).

Now suppose we are trying to fit two triples and a pair into the eight physical registers. Figure 3-1 shows the overlap between these clusters.

Without the modifications to be explained subsequently, the interference graph for these clusters would look the same as it would for a program containing eight single,

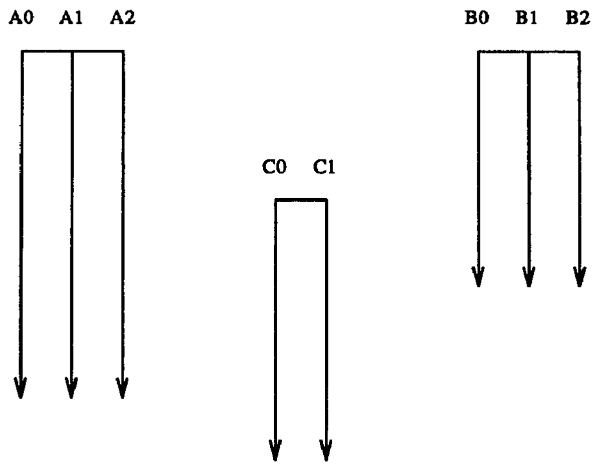

Figure 3-1 Example 1: Liveness Overlap

overlapping registers. In particular, it would look like eight nodes with maximal edges, as shown in Figure 3-2. But while eight single overlapping virtual registers are colorable among the eight physical registers, two triples and a pair are not. Suppose, for example, that the triples are colored to their only possible places, namely *r0-r2* for one, and *r4-r6* for the other. Only *r3* and *r7* are left for the two-word cluster, which is not an acceptable coloring.

For the second example, consider trying to fit three two-word clusters into four registers. Figure 3-3 shows the overlap between these clusters. (Cluster X in this diagram is split; it comes alive toward the bottom of a loop and stays alive into the next iteration. Also, for clusters X and Z, the lifetimes of the two cluster-mates differ.)

Again, without the modifications to be explained, the interference graph for these clusters would look the same as if the virtual registers were not clustered. But in the case where they are not clustered, the graph colors in four registers. In the case of the example, four registers can't color

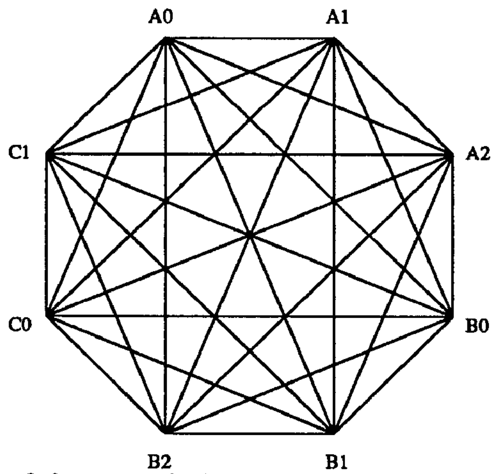

Figure 3-2 Example 1: Maximal Conflict Edges

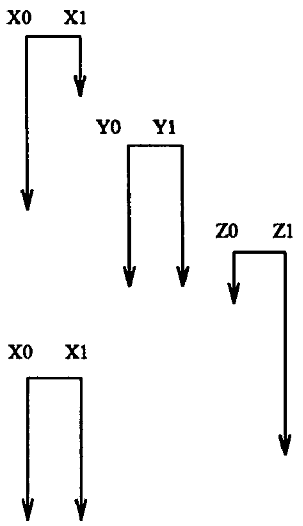

Figure 3-3 Example 2: Liveness Overlap

the graph. For instance, if Y0 (the 0-mate of cluster Y) and Y1 (the 1-mate) color to physical registers *r0* and *r1* respectively, and if Z0 and Z1 color to *r2* and *r3* respectively, then X0 can color to *r2* only while X1 can color to *r1* only. But that's unacceptable because *r2* and *r1* aren't an aligned pair.

While one would be tempted to conclude that coloring register clusters requires more detailed information about the nature of interferences, the solution algorithm actually benefits from having *less* information. That is, where register interferences are implicit, these interferences are best left unexpressed in the interference graph. With small modifications to the graph coloring algorithm, the compiler makes easy and fast use of the reduced volume of information.

While all edges in the graph indicate interference relations, not all interference relations require an edge. The fact that the cluster-mates of a single cluster cannot reside in the same register is implicit. For example, the 0-mate of a two-word cluster must go into an even register, while the 1-mate must go into an odd register. Likewise, the interference between the 0-mate of one two-word cluster and the 1-mate of another two-word cluster is implicit. Similar implications exist for other kinds of clusters.

By omitting the implied edges, the resolution of the graph coloring problem begins to appear. In the first example above, the two three-word clusters and one two-word cluster would have an interference graph as shown in Figure 3-4.

In the unmodified algorithm, all eight virtual registers would be considered unconstrained, because none of them interfered with eight or more other virtual registers. In the modified algorithm, we don't ask "does this node have fewer than 8 neighbors?" to decide whether a node is

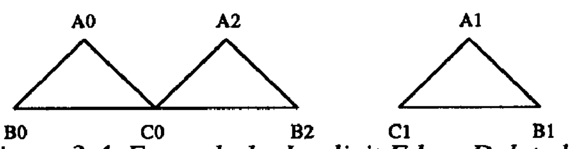

Figure 3-4 Example 1: Implicit Edges Deleted

unconstrained. Rather, we ask "Does this cluster have fewer neighbors than the number of places the cluster can be fitted into the register set?" So, the two neighbor edges of A0 are sufficient to mean that cluster A is constrained. Cluster B is constrained in the same way. And, though four suitable registers are available for C1 and that node only has two neighbors, nevertheless cluster C is constrained because there are only four suitable registers for C0 and that node has four neighbors.

In the second example above, elimination of the implicit interferences would cause the three pairs to have an interference graph as shown in Figure 3-5.

Though X0, X1, Z0, and Y1 appear to be unconstrained because they have fewer than two neighbors, this is not the case. Y1 is constrained because the cluster Y is constrained; the interferences imposed on Y0 alone are enough to constrain the cluster as a whole. Likewise, Z0 is constrained because cluster Z is constrained. This leaves X0 and X1. While on the surface they seem unconstrained, they are in fact constrained. As I said above, the question is whether the cluster as a whole has fewer neighbors than places in which the cluster could be fitted. Since X0 interferes with Y0, and X1 interferes with Z1, the cluster X interferes with Y and Z, the union of the clusters with which the individual cluster-mates interfere.

So far, the complexity of the interference graph has been greatly reduced by omission of implicit interferences. As a result, the solution strategy has begun to appear. A further simplification can be realized by recognizing the synonymous nature of some interference relations, and choosing to represent these synonyms in a single **normal form**. For example, if the lifetimes of the 1-mates of two two-word clusters overlap, then there is an interference relation between them. Now even if the 0-mates of these clusters do not have overlapping lifetimes, they cannot be assigned to the same register *r* because that would imply that the 1-mates would be assigned to *r+1*.

Two benefits can be realized by using the normal form. The first benefit is that it eliminates redundancy, further reducing the volume of data in the interference graph. In particular, the first example's conflict graph shows that A0 interferes with B0, that A1 interferes with B1, and that A2 interferes with B2. These statements are redundant; any one of those interference relations implies the other two. This redundancy is eliminated by only depicting one of the relations.

There is a cyclic nature to the synonymous interferences between clusters of differing sizes. For instance, between a two-word cluster P, and a four-word cluster Q, an interference relation between P0 and Q0 is synonymously expressed as an interference relation between P1 and Q1. Likewise an interference between P0 and Q2 could alternately be expressed as an interference between P1 and Q3. In the author's implementation, one or both of the virtual registers connected by a normalized interference edge is the 0-mate of its cluster.

Figure 3-6 shows the interference graph for the first example with the redundant interferences deleted.

The second benefit realized by using the normal form, is shown in the second example. It benefits in a more important way: after normalization, the interference graph clearly depicts the fact that all three register-pairs are constrained. This is shown in Figure 3-7.

Figure 3-8 gives the algorithm in pseudo-code for building the interference graph.

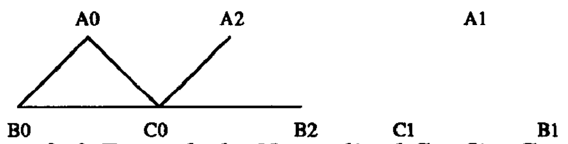

Figure 3-6 Example 1: Normalized Conflict Graph

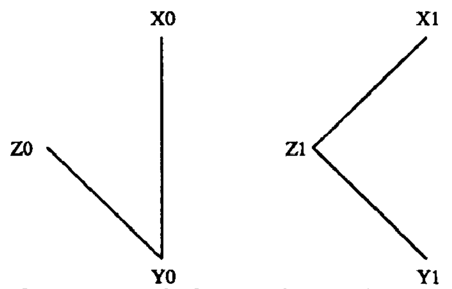

Figure 3-5 Example 2: Implicit Edges Deleted

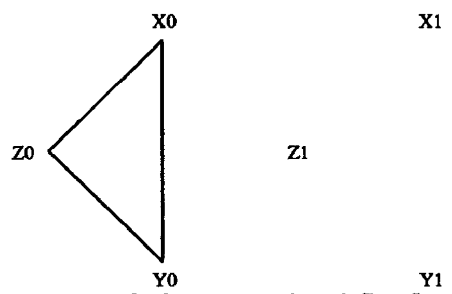

Figure 3-7 Example 2: Normalized Conflict Graph

```
For each pair of interfering registers ri and rj, with ri < rj {

   If any of these conditions exist, suppress the interference relation as implicit:

      1)  ri and rj belong to the same cluster;  or
      2)  ri and rj are not alignable;  or
      3)  ri and rj are both physical registers;  or
      4)  ri [or rj] is a physical register in a cluster too small to host the cluster containing
          rj [or ri]

   Otherwise normalize the proposed interference relation, by expressing it in terms of an inter-
   ference with the 0-mate of one cluster or the other (or both):

      While neither ri nor rj is the 0-mate of its cluster {
         ri := the index of the next lower cluster mate of ri's cluster;
         rj := the index of the next lower cluster mate of rj's cluster;
      }

   If ri and rj are already marked as interfering, suppress the interference relation as redundant;
   otherwise record the interference relation between ri and rj.
}
```

Figure 3-8 Algorithm to Collect Germane Conflicts

3.3. Understanding the Interference Graph

The normalized form of the interference graph is nearly what's desired, but not quite. It must be coupled with a shift in perspective on the part of the reader. Readers familiar with the graph coloring technique, as applied to architectures without register clustering, may operate under concepts that work for those architectures, but they do not work here. In particular, it becomes important to distinguish between the number of interference relations (edges) in the interference graph, and the number of coloring constraints they imply.

Casual inspection of Figure 3-6 shows us that the cluster C has four interference edges, i.e. two of its interferences are with cluster A and two are cluster B. Generally, one might expect that this would imply that four coloring constraints are imposed upon C by its neighbors, i.e. two coloring constraints imposed by A and two coloring constraints imposed by B. For instance, if A gets colored to registers *{r4, r5, r6}*, then C is precluded from being colored at *{r4, r5}* and at *{r6, r7}*.

But Figure 3-6 also shows that the cluster A [and similarly B] has three interference edges, so one might expect that these edges imply there are three coloring constraints imposed upon A, i.e. two coloring constraints by C and one by B. Actually this is not the case. No matter where C is assigned, it cannot impose two coloring constraints upon A. That is, of the possible places to assign A, assigning C first can only reduce by one the number of places A can go.

Therefore, Figure 3-6 accurately depicts the interference relations between A, B, and C. It even accurately depicts the coloring constraints imposed upon C, the smaller cluster. But it's not a picture of the coloring constraints imposed upon the larger clusters, A and B. The picture suffices though, if one remembers that the constraints imposed upon a larger cluster are not the sum of the constraints imposed upon its cluster-mates, but rather are the union thereof. The interferences between A2 and C0 and between A0 and C0 are two constraints upon C, but only one constraint upon A.

Accordingly, Chaitin's assertion about the bit matrix being symmetrical is not really true of multi-register clusters, when seeking to count coloring constraints imposed upon a cluster. However, the asymmetry is not so great as to require the full bit matrix to be expressed. It still will be possible to extract accurate information from only one triangle of the matrix.

Thus it has been shown that implied interference edges are not needed, and their deletion makes the graph pruning operation more apparent. In addition, fewer edges means visiting the triangular matrix less, and will reduce memory requirement when the matrix is converted to interference vector form. Traversal time through the graph is reduced as well.

3.4. Portraying Register Clusters and Subsumed Registers

Each virtual register is distinguished from each other virtual register by a unique virtual register index. Because the size of the interference graph bit matrix is proportional to the square of the greatest virtual register index, best performance is obtained if these indices are as small as possible. The register allocator in **int$_e$l**'s compiler receives a single data value associated with each virtual register index. This datum is an indication of how large the cluster is to which the virtual register belongs. In addition, the register allocator is assured that the indices for a cluster are contiguous and are assigned in accordance with natural alignment rules. For example, the 0-mate of a three- or four-word cluster will always have an index divisible by four.

From these data, the register allocator can form an association between the indices of all the virtual registers belonging to a cluster. The association takes the form of a list, in which the virtual register with the highest offset (the cluster-boss) must point to the second highest, etc., down to the 0-mate. In addition, associated with each virtual register is a pointer directly back to the cluster-boss, and an indication of the virtual register's offset within the cluster. An example of this is shown in Figure 3-9-A.

As register clusters are successfully subsumed by other register clusters, the list of the subsumed cluster is appended to the list of the subsuming cluster. The index of the cluster boss recorded for each register in the subsumed cluster is updated to the index of the cluster boss of the subsuming cluster. Also, the cluster offsets of the registers in the subsumed cluster are modified by the amount they are shifted to align with the subsuming cluster. For example, if a two-word cluster is subsumed into the third and fourth words of a four-word cluster, the cluster offsets of the subsumed registers are changed from zero and one to two and three. (See Figure 3-9.) Thus it is possible to determine the subsumer of any register by simply going to the register's cluster boss, and then skipping down the list until the first register with a matching cluster offset is found.

Subsumption of clusters is subject to a few straight-forward rules. In register allocators that don't deal with clusters, the two registers to consider subsuming must not both be physical registers, and they must not interfere. Analogously, when dealing with clusters, both clusters cannot be comprised of physical registers, and all the corresponding cluster-mates of the two clusters must not interfere. The latter condition can be checked simply by testing for the normalized interference relation.

In addition, the cluster-mates of the potential subsumee must be alignable with the subsumer cluster-mates. In Figure 3-9, a two-word cluster was successfully subsumed into the third and fourth words of a four-word cluster. They can also be subsumed into the first and second words of a four-word cluster. But they cannot be subsumed into the second and third words, because when the four-word cluster is successfully allocated to a four-word boundary, as it surely must be, access to the subsumed two-word cluster causes an alignment violation.

When dealing with architectures that support clusters as large as four words, one case is rather tricky to support: subsuming the first word of a two-word cluster with the last word of a three-word cluster, resulting in a four-word cluster. This case is shown in Figure 3-10. To support this case, a fourth virtual register index has to be reserved with each three-word cluster just in case this subsumption opportunity arises. If there is no interference, the two-word cluster is subsumed into the three-word cluster by extending the latter to a four-word cluster. There would be more similar cases for an architecture in which clusters larger than four words are supported.

Now, with respect to register allocators that don't deal with clusters, consider: When dealing with the subsumption of two virtual registers, the decision as to which one of the two is the subsumer and which is the subsumee is arbitrary. When dealing with the subsumption of a virtual register and a physical register, the virtual register should be the subsumee; i.e. if the subsumption succeeds, the use of the virtual register will be replaced by the use of the physical register throughout the program, and any moves of the physical register from or to the virtual register will be deleted.

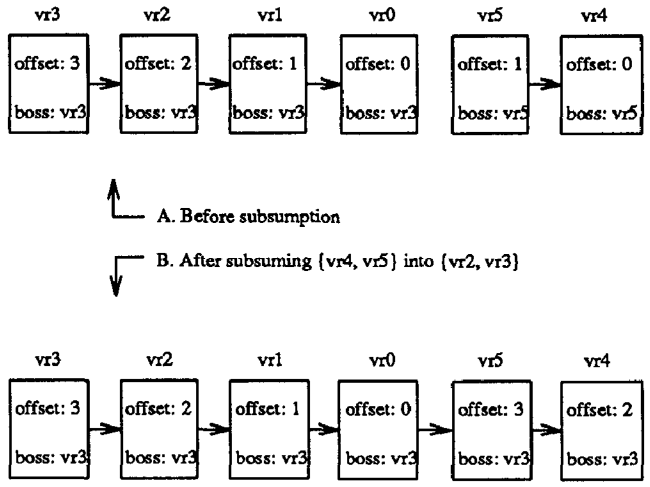

Figure 3-9 Subsuming a Pair and a Quadruple

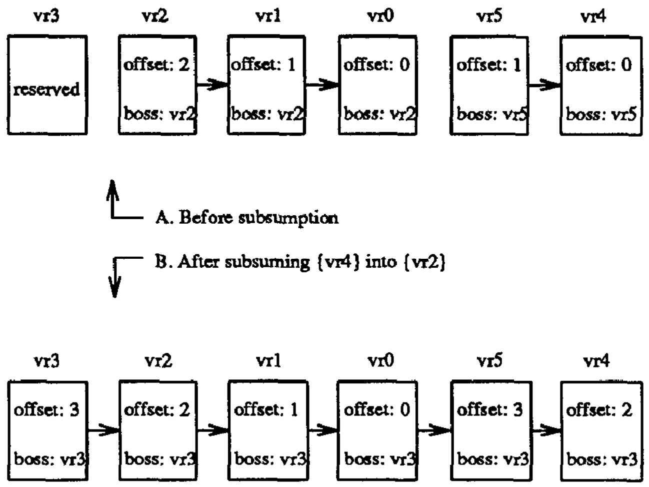

Figure 3-10 Subsuming a Pair and a Triple

Analogously, when dealing with clusters, clusters of virtual registers may be subsumed by clusters of physical registers, but not the other way around. One important condition is that a physical register cluster must not be enlarged by a subsumption. To clarify, the **int$_e$l** 80960 microprocessor's 32 registers are addressable as eight distinct four-word clusters. However, register *G15* is not colorable because, by convention, it always contains the frame pointer. Therefore, we can never subsume a four-word virtual register cluster with *{G12, G13, G14}*, because we can't extend the latter with *G15*.

When dealing with two clusters that are both comprised of virtual registers, if one of the clusters is larger, it will be the subsumer; when the clusters are the same size, the decision is arbitrary.

Armed with these conditions, the subsumption routine considers whether subsumption candidates can indeed be subsumed. Subsumption candidates are identified by scanning the intermediate program form for moves from one cluster to another; the source and destination should be subsumed if possible. If they can be subsumed, then the clusters are joined as described above, the corresponding move operations are deleted from the intermediate program form, and all the interferences recorded for the cluster-mates of the subsumee are transferred to the corresponding cluster-mates of the subsumer. The scanning of the program is done in depth-first order with respect to loops, because it is more desirable to delete move instructions from within loops than from without.

Figure 3-11 gives the algorithm in pseudo-code for subsumption.

3.5. Convert Interference Graph from Matrix Form to Vector Form

Up to now, the interference graph has been represented as a bit matrix. This form provides good support for the random accesses needed to install interference edges and to consider and apply subsumptions. After performing these operations, the remaining work of the register allocator is to actually color the interference graph. The operations that do this need to be able to iterate over the interference edges applicable to each cluster. This iteration does not benefit from the random access capability of the bit matrix form for the graph. Indeed, because the graph tends to be sparse, controlling the iterator by scanning the bit matrix is rather slow.

To wit, the interference graph is converted to a list form, in which each node (except those that have been subsumed by other nodes) has a list of all the interference edges that apply to it. Since no interference edges can be added to or deleted from a node from this time forward, it is not necessary to implement a linked list or even a segmented list, as implemented by Chaitin et al (CACCHM 81). A simple vector suffices.

3.6. Pruning the Interference Graph

Traditionally, pruning removes nodes from the interference graph iteratively until the graph is empty. Coloring will assign registers to the removed nodes in reverse order from their removal. In each pruning iteration, it is preferable to remove a node that is "unconstrained", i.e. has *degree* $< k$, where *degree* is the number of neighbors still in the graph, and where k is the number of physical registers available. Further, empirical methods have been used to determine that better results are attained by removing the most constrained of the unconstrained nodes first (i.e. the node with largest *degree* $< k$). See Pinter (BGGKMNP 89) for a discussion of these priorities.

To deal with multi-register clusters, the traditional implementation is altered as follows. First of all, pruning does not remove individual nodes, but clusters instead. To this end, the pruning initializer computes the initial *degree* values for each cluster as a whole, and the pruning iterator selects whole clusters based on this information. In the traditional implementation, the initial *degree* value for each node is the number of coloring constraints imposed upon the node by its neighbors, and this is equal to the length of its list of interference edges. But as has been pointed out in section 3.3, the number of coloring constraints imposed upon a cluster is not necessarily equal to the sum of the number of interference edges emanating from each node of the cluster. Recall that a smaller cluster imposes only one coloring constraint upon a larger cluster, whether there is one interference edge between the clusters, or more.

Pruning in the presence of multi-word clusters also differs from traditional pruning in that for large clusters, the degree k at which they become unconstrained is a smaller value than for small clusters. For example, the **int$_e$l** 80960 microprocessor has 32 general purpose registers, addressable as 32 singletons, 16 two-word clusters, or 8 three- or four-word clusters. However, four of the registers (*G15, R0, R1,* and *R2*) are not colorable because they have reserved meanings. The four reserved registers are not contiguous and four-word aligned, so they interfere with two different four-word clusters. Consequently, the value of k is 28 for single-word clusters, 13 for two-word clusters, 7 for three-word clusters and 6 for four-word clusters.

It is inconvenient to have four different "magic numbers" at which different size clusters become unconstrained. To alleviate this nuisance, a *weighted-degree* value is used, instead of actual degree. *weighted-degree* is calculated by multiplying actual degree by a weighting factor, and then adding a bias value. The weighting factors used are one, two, four, and four respectively for 1-, 2-, 3-, and 4-word clusters. The added bias values are such that clusters of all

```
Performing a depth-first scan of the intermediate program form, for each source register rs and
destination register rd found in a move instruction, do
{

   Because rs and/or rd may have already been subsumed by some other register:

      rs := subsumer_of (rs);
      rd := subsumer_of (rd);

   Call the subsumption attempt successful if rs and rd are now equal.

   Otherwise normalize the registers to check for the normalized interference relation.  This is
   done the same way as in the recording of interferences:

      While neither ri nor rj is the 0-mate of its cluster {
         ri := the index of the next lower cluster mate of ri's cluster;
         rj := the index of the next lower cluster mate of rj's cluster;
      }

   Reject the subsumption attempt as a failure under the following conditions:

      1)  if rs and rd interfere;  or
      2)  if rs and rd are both physical registers;  or
      3)  if rs or rd is a physical register, and the subsumption would enlarge its cluster;  or
      4)  if rs and rd cannot be aligned to the same register.

   Otherwise coalesce the clusters as depicted in the examples in Figure 3-9 and Figure 3-10;  and
   call the subsumption attempt successful.
}

Modify the program intermediate form to account for all successful subsumptions, by doing
{

   First, for all virtual register indices appearing in the program intermediate form, replace them
   by the corresponding active cluster mate of the same cluster.

   Then, delete all move instructions that move a register to itself.
}
```

Figure 3-11 Algorithm to Perform Subsumptions

four sizes have the same "magic number" at which they become unconstrained. The bias values are also large enough to assure that the *weighted-degree* of a cluster with few or no interference relations is a value no less than zero.

Figure 3-12 gives the algorithm in pseudo-code for computing the initial *weighted-degree* values and preparing for pruning.

Subsequently, clusters are pruned from the interference graph in the normal way. As with graph coloring for traditional register files, if the graph ever has only constrained nodes, one of them is selected for pruning. Pinter's best-of-three selection heuristic (ibid), coupled with Briggs' late spill decision technique (BCKT 89), yields an excellent implementation. For our implementation, the late spill decision is particularly valuable because, in the presence of large clusters, there tend to be more potential spills that end up being colorable after all. Consider that the worst possible placement of eight single-word clusters among 32 physical registers could preclude a four-word cluster from being colorable. But worst case placement of the singletons is not likely, so the larger cluster will probably be colorable after all.

```
For each cluster c of virtual registers {

   Count the number of constraints imposed upon c by its neighbors.

   $weighted\_degree = constraints * weight\_factor_{cluster\_size(c)} + weight\_bias_{cluster\_size(c)}$;

   If weighted_degree >= weighted_degree_below_which_clusters_become_unconstrained, mark c as constrained;
   otherwise mark c as unconstrained;
}
```

Figure 3-12 Algorithm to Initialize for Pruning

Figure 3-13 gives the algorithm for pruning in pseudocode.

Calculation of the heuristic estimates for spill *cost*, *area*, and *degree* (see BGGKMNP 89), is somewhat more complicated for our implementation. In particular, *cost* must estimate the number of loads and stores required for each cluster-mate, reduced by the savings available when adjacent mates of a cluster can be loaded simultaneously. *area* for a cluster is simply the sum of the *area* calculated for each cluster-mate, and *weighted-degree* is used instead of actual *degree*.

3.7. Coloring the Interference Graph

Coloring the clusters in reverse order from how they were pruned is rather straight-forward. The technique introduced by Chaitin (Chai 82), refined by others, and especially coupled with the late-spill-decision improvements by Briggs (BCKT 89) yields a high quality implementation. The three different spill heuristics applied in the pruning phase may yield three different orders for coloring. While Pinter suggested only coloring the one with the least overall cost, Briggs late-spill-decision (really a coloring-phase ratification of pruning-phase spill decisions) implies that all three colorings must be attempted. After each coloring, the overall costs of ratified spill decisions are compared, and the cheapest coloring is accepted.

Of course, when the heuristics do not digress in how they would prune the graph, a single coloring is adequate. This is the usual case, since most programs color with no spills at all.

One alteration peculiar to multi-word clustering is that, of course, the registers assigned to such clusters must be appropriately aligned. As with traditional coloring, if a cluster was unconstrained when it was pruned from the graph, it is certain an appropriate color can be found for it.

It is also important to be careful about how clusters are spilled. It is tempting to be lazy and simply spill entire clusters. However, it is important to avoid restoring portions of a cluster that are not live. Consider trying to color the example in Figure 3-14 into exactly two physical registers. (Ignore the fact that some simple reordering of the instructions might well avoid the problem altogether.) Figure 3-14A shows the source code for the example, and Figure 3-14B shows the intermediate object code as input to the register allocator. Figure 3-14C suggests a possible output code sequence, based on the supposition that the register allocator would choose to spill the cluster *{vr0, vr1}*.

The output code shown has a bug in it. In particular, at the points where *{vr0, vr1}* are saved and restored, really only *vr0* is live. The restoration of the whole cluster means that the wrong value will be stored to the variable *t*.

Strictly speaking, the register allocator output shown in Figure 3-14C wouldn't really occur. Instead, the spill code suggested would be inserted, and the register allocator would be reiterated. In the second iteration, it would become apparent that *vr4* must be spilled as well. Spill code would be inserted, and the register allocator would be iterated a third time. This would result in a successful output. Unfortunately, it would include the spilling of both a two-word cluster and a one-word cluster, when all that is really necessary is for the first word of the two-word cluster to be spilled.

4. Future Work

The techniques used here are directly applicable to graph-coloring environments in which it is desirable to support multi-register operands that are aligned and use contiguous registers. By relaxing the alignment restriction, the contiguity restriction, or both, these techniques could be applied to model other interesting resource management problems.

```
For each heuristic h {

   While the graph G is not empty {

      If there is an unconstrained cluster
         Choose cluster c with largest weighted_degree (among the unconstrained clusters);
         Reduce appropriately the weighted_degree of all unpruned neighbors of c
            (this may cause some to change from being constrained to unconstrained);
         Delete c from G and place on stack;
      Otherwise
         Choose cluster c with minimum h(c);
         Reduce appropriately the weighted_degree of all unpruned neighbors of c
            (this may cause some to change from being constrained to unconstrained);
         Delete c from G and place on stack, annotated that it may need to be spilled;
   }
}
```

Figure 3-13 Algorithm to Prune Clusters

```
A -- Source Code:

struct {
   int a;
   int b;
} x, y;
int p, q, r, s, t;
   ...
x = y;
r = p + q;
s = x.a;
t = r;
   ...


B -- Object Code, before register
     assignment

ldl  _y,{vr0,vr1}  # load 2 words of y
stl  {vr0, vr1},_x # store 2 words of x
ld   _p,vr2        #
ld   _q,vr3        #
addi vr2,vr3,vr4   #  /  r = p + q;
st   vr4,_r        # /
st   vr0,_s        # s = x.a;
st   vr4,_t        # t = r;

C -- after first register
     assignment attempt

ldl  _y,G0
stl  G0,_x
stl  G0,spill_area # too much saved
ld   _p,G1
ld   _q,G0
addi G1,G0,G1
st   G1,_r
ldl  spill_area,G0 # too much restored
st   G0,_s
st   G1,_t         # wrong value stored
```

Figure 3-14 Must Spill Only the Live Cluster-Mates

These other models might be more general than necessary for the problem of allocating processor registers, because alignment and contiguity are required on most processors that support multi-register operands. But they might be useful, for example, in distributing a base-load of real-time tasks over a network of processors. Consider a network of geographically scattered multi-processor supercomputers, where one user might decree "I don't care what four processors my important real-time application is divided among, but the processors must be in one geographical location so that they can communicate quickly with each other." Another user might say "I don't care what six processors my real-time application is divided among, but to be fault tolerant with respect to meteor strikes, I want the processors far apart."

Managing the processing elements, communication paths, and storage resources of systolic architectures might also benefit from a generalization of the techniques presented.

To generalize the model, one must revisit the issue of how to calculate the number of coloring constraints imposed upon a cluster from the set of interference relations incident upon the cluster. Likewise, complications would be expected to arise in the calculation of weighted degree from the number of coloring constraints. Finally, one would expect a reduced ability to regard interference relations as implicit and so delete them.

As a clarifying example, consider assigning only two-word clusters to a bank of thirty-two physical registers. If alignment and contiguity are required, clusters with degree less than sixteen could surely be colored. But if the alignment restriction is relaxed, with the most pessimistic coloring, one can only be sure of successful coloring of clusters with degree less than eleven. Moreover an interference relation between the 0-mate of one cluster and the 1-mate of another is no longer implicit, and so must be expressed in the interference graph.

5. Conclusion

intel's 80960 architecture demands that the compiler's register allocator deal well with clustered access of up to four words. Rather than resorting to non-graph-coloring techniques, or attaching pre-coloring or post-coloring phases to deal with clusters larger than a single word, we have pursued and found an integrated cluster coloring process. It was easier than we anticipated.

6. Acknowledgements

I wish to thank Joseph Goodman for his careful study and comments about this work; Preston Briggs of Rice University for several conversations that helped me recognize the weak spots in an early draft; and Dave Spencer who gave considered editorial suggestions.

7. References

(BCKT 89)	Briggs, P., Cooper, K.D., Kennedy, K., and Torczon, L., "Coloring heuristics for register allocation", *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, ACM Press (June 1989), 275-284.
(BGGKMNP 89)	Bernstein, D., Goldin, D.Q., Golumbic, M.C., Krawczyk, H., Mansour, Y., Nahshon, I., and Pinter, R.Y., "Spill code minimization techniques for optimizing compilers", *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, ACM Press (June 1989), 258-263.
(CACCHM 81)	Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., and Markstein, P.W., "Register allocation via coloring", *Computer Languages 6* (1981), 47-57.
(Chai 82)	Chaitin, G.J., "Register allocation and spilling via graph coloring", *Proceedings of the ACM Symposium on Compiler Construction* (June 1982), 98-105.
(Hsu 87)	Hsu, Wei-Chung, *Register Allocation and Code Scheduling for Load/Store Architectures*, University Microfilms International, 1988.
(ChHe 84)	Chow, F., and Hennessy, J., "Register allocation by priority-based coloring", *Proceedings of the ACM Symposium on Compiler Construction* (June 1984), 222-232
(LaHi 86)	Larus, J.R., and Hilfinger, P.N., "Register allocation in the SPUR Lisp Compiler" *Proceedings of the SIGPLAN'86 Conference on Programming Language Design and Implementation*, ACM Press (June 1986), 255-263.
(Intel 89)	*80960CA User's Manual*, Intel Corporation (1989)