Coloring Register Pairs

Preston Briggs
Keith Cooper
Linda Torczon

CRPC-TR91176
October, 1991

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Coloring Register Pairs

Preston Briggs, Keith D. Cooper, and Linda Torczon

Department of Computer Science
Rice University
Houston, Texas 77251–1892

Many architectures require that a program use pairs of adjacent registers to hold double-precision floating-point values. Register allocators based on Chaitin's graph-coloring technique have trouble producing a *fair* allocation under these circumstances – that is, an allocation that favors neither paired registers nor single registers. We show that one side effect of our *optimistic* coloring scheme is that it produces a fair allocation for registers and pairs of registers.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – compilers, optimization

Additional Keywords and Phrases: graph coloring, register allocation, shipbuilding problem

1 Introduction

Register allocation is an important problem that arises in the back end of a compiler. As input, the allocator takes code that uses an arbitrary number of registers – we call these *virtual registers* – and maps it into code that uses a limited set of registers – those registers available on the target machine. Thus, the allocator selects the set of values that reside in registers at each point in the routine. The goal of allocation is to choose the set of values in a way that minimizes execution time. Unfortunately, optimal allocation is NP-complete [13].

In recent years, the graph coloring paradigm has emerged as one of the techniques of choice for allocation. The first coloring allocator was built by Chaitin and his colleagues at IBM for the PL.8 compiler [3, 4]. Chow and Hennessy describe a somewhat different approach to allocation via graph coloring [5, 6]. A Chaitin-style allocator works by constructing an *interference graph* that models conflicts between the virtual registers. Next, the allocator tries to find a k-coloring for the graph, where k is the number of machine registers. If it succeeds, the colors can be mapped onto machine registers. If a k-coloring cannot be found, one or more values are spilled; that is, loads and stores are inserted around each reference to the value. Then, the allocator repeats this process on the modified code, building a new interference graph and trying to color it.

In general, Chaitin-style allocators produce good allocations; however, they have a tendency to over-spill in some situations. This problem has prompted at least two published improvements that reduce the amount of spilling [1, 2]. When the allocator must deal with register pairs, this over-spilling is exaggerated. The allocator spills pairs in preference to single registers – possibly spilling pairs when registers are actually available to hold them.

In 1986, Hopkins described a method to handle the register pair constraints that arise in the shift instructions on the ROMP microprocessor – the engine in RT/PC workstations [10]. In 1990, Nickerson published a method for allocating structures into an aggregate set of adjacent registers. He observed that an allocator based on our 1989 paper produced good allocations under his scheme [2, 12]. This paper explores the issue in some detail. It discusses how to represent pairs in the interference graph, shows why Chaitin's allocator over-spills pairs, and explains how our 1989 allocator produces a fair allocation.

2 Background

Throughout this paper, we assume that the allocator works on low-level intermediate code or assembly code. Before allocation, the code can reference an unlimited number of virtual registers. Rather than map virtual registers directly onto physical registers, it discovers the distinct *live ranges* in a procedure and allocates them to physical registers. A single virtual register can have several distinct values that are interesting in different parts of the program – each of these values will become a separate live range. Figure 1 illustrates a Chaitin-style allocator. It has six phases.

Renumber finds the live ranges and gives them unique names. It creates a new live range for each definition point. It then unions together the live ranges that reach each use point.

Build constructs the interference graph. The nodes in the graph represent live ranges. The edges correspond to "interferences" between live ranges. Two live ranges interfere if they are simultaneously live and cannot reside in a single register.

Coalesce attempts to combine live ranges. Two live ranges l_i and l_j are combined if the initial definition of l_j is a copy from l_i and l_i and l_j do not otherwise interfere. This has two beneficial effects: it shrinks the number of live ranges that the allocator must consider in later passes, and it eliminates the copy instruction that connects them.

Simplify constructs an ordering of the nodes. It initializes a stack of nodes to empty and then repeats the following steps until the graph is empty:

1. If there exists a node l_i with fewer than k neighbors, remove l_i and all of its edges from the graph. Place l_i on the stack for coloring.
2. Otherwise, use a *spill metric* to choose a node l_i to spill. Remove l_i and all of its edges from the graph. Mark l_i to be spilled.

If any node is marked for spilling, the allocator inserts spill code and repeats the allocation process. If no spilling is required, it proceeds to *select*.

Select assigns colors to the nodes of the graph in the order determined by *simplify*. It repeats the following steps until the stack is empty:

1. pop a live range from the stack,
2. insert its corresponding node into G, and
3. give it a color distinct from its neighbors.

Spill code is invoked if *simplify* decides to spill a node. Each spilled live range is converted into a collection of tiny live ranges by inserting a load or store at each use or definition.

This algorithm has been used in a number of implementations; in general, it produces good allocations.

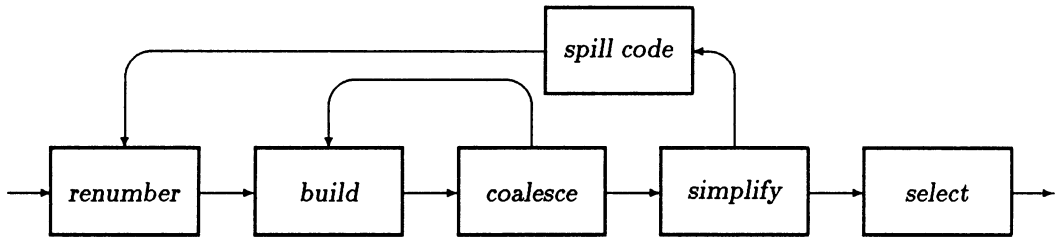
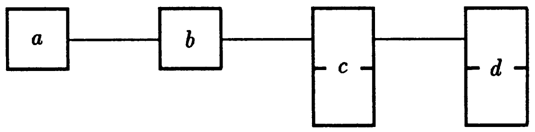


FIGURE 1: Chaitin's Allocator

3 Register Pairs

Register pairs are a fundamental part of many machine architectures. For example, many machines use pairs of registers to represent double-precision floating-point values. The two most common constraints imposed on register pairs are requiring that the registers be adjacent (named with consecutive integers), and requiring that an adjacent pair be aligned (typically requiring the first register to have an even number).

To help in the discussion, we will use the following simple example to illustrate our points. Our target machine has four registers. Now, imagine four live ranges, a, b, c, and d; where a and b are single-precision values and c and d are double-precision values. Assume that b interferes with a and c, and that c also interferes with d. The interference graph for this example looks like:



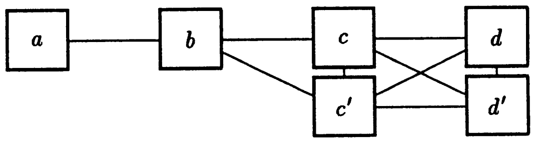
It is important to realize that we are dealing with a fundamentally harder problem – coloring a *weighted* graph. This is more difficult than coloring an unweighted graph. For example, the interference graphs that result from straight-line code have a particularly simple structure; they are *interval* graphs. A minimal coloring for an unweighted interval graph can be found in linear time [9, page 14]. In contrast, finding a minimal coloring for a weighted interval graph is identical to the Shipbuilding Problem which has been shown to be NP-complete, even when weights are constrained to be either 1 or 2 [9, page 204]. In any case, a global register allocator must be prepared to handle procedures that have interesting control flow. In fact, Chaitin *et al.* showed that any arbitrary interference graph can be generated by some procedure [4].

Fabri explored variations of this problem in the context of packing arrays in memory [7, 8]. While the work is interesting, it is not directly applicable to our problem. For example, her problem has no analog for the problems of spill choice and spill placement. Thus, for the purposes of register allocation, we will continue to work with Chaitin's allocator.

The goal, of course, is to provide a fair allocation – competition between values that need a single register and those that need a pair should be resolved on the basis of some model that accurately reflects the real costs of each choice. The remainder of this section discusses how we might apply Chaitin's allocator to unconstrained pairs of registers and to adjacent pairs of registers. In both cases, we show how to represent them in the interference graph and discuss the allocations that result.

3.1 Unconstrained Pairs

We initially consider the simplest case, assuming that the target machine places no adjacency or alignment restrictions on pairs. This is actually no problem at all. Noting that the above graph overly constrains the coloring, we can simply handle the two halves of each register pair separately. This yields the following graph:

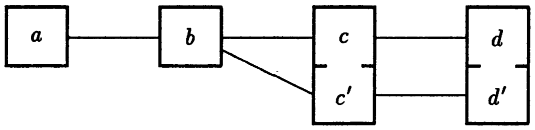


Note that we have avoided the problem of trying to color a weighted graph. Therefore, we may simply use Chaitin's algorithm, unmodified, to find the minimal four-coloring. On this example, Chaitin's method will always find a four-coloring.

3.2 Adjacent Pairs

Extensions to handle adjacent register pairs correctly are more difficult. For the moment, assume that the target machine requires that pairs be allocated to aligned adjacent registers. The problem arises during color selection; the allocator must coordinate the colors for two nodes that appear unrelated. If it assigns a color to one and cannot assign an adjacent color to the other node, it must either reconsider the colors that it has already assigned or report complete failure. Neither of these is a good alternative. For *select* to reconsider colors will require backtracking, which can take exponential time. Reporting failure seems unhelpful; it provides no clear direction for recovery.

The best alternative appears to be changing our representation for pairs; we should consider treating the pairs as indivisible units. This gives us the following graph:



The resulting graph (more accurately *multigraph*) resembles our original graph from Section 3, with additional edges to represent necessary interferences. The simplicity of this representation is appealing. Intuitively, the extra edge between b and c reflects the additional constraint placed on b. Similarly, the extra edge between c and d balances their extra width.

Multigraph Representation

So, why have we moved to a multigraph representation, besides the intuitive appeal of the pictures? For stronger justification, we must consider the role that edges play in the coloring process. First, edges represent interferences – they are critical to the correctness of the resulting allocation. Second, they trigger spilling in *simplify*.

Recall that *simplify* examines the graph and repeatedly removes nodes with less than k neighbors, where k is the number of available colors. A node that has fewer than k neighbors will always receive a color – a color is available for it independent of context. If a node has k or more neighbors, then *simplify* marks it for spilling. To measure the number of neighbors, the algorithm uses the node's *degree*, the number of edges incident on the node. Thus, for the allocator to work correctly, a node's degree should accurately reflect its colorability.

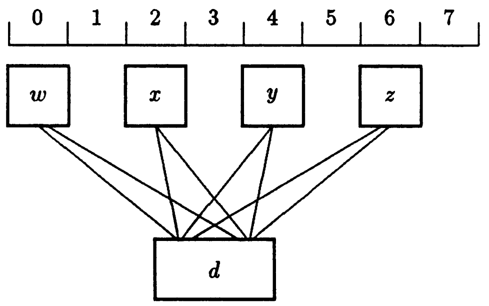
The graph shown above correctly models the colorability of each of node. Any interference that involves a value stored in a pair of registers adds two edges to the graph. Thus, the interference between b and c creates a pair of edges, as does the interference between c and d. (While we draw the nodes with larger boxes and two labels, in the graph, c and c' are but a single node.)

This rule makes sense. On our four-register machine, two single registers that interfere with a register pair raise the pair's degree to four. Placed correctly, the singles could block allocation of registers to the pair. Similarly, three register pairs that all mutually interfere create a situation where all three have degree of four. This reflects that fact that three register pairs cannot fit into four registers.

Sometimes, it is convenient to introduce an interference between a single register and one half of a pair. Often, one half of a register pair may be used as the source or destination of an operation. For example, the even half of a complex pair is copied to another register. In this case, the target of the copy should interfere only with the odd half of the complex pair. An interference between the target register and the even half of the pair would prevent *coalesce* from combining them and eliminating the copy.

A Problem

Unfortunately, Chaitin's allocator performs poorly on graphs of this form; it tends to over-spill register pairs. To understand this problem requires a somewhat more complex example. Consider a machine with eight single-precision floating-point registers that requires double-precision values to be allocated to adjacent pairs of registers. If we have a double-precision value, four single-precision values are all that is required to force a spill under Chaitin's allocator. The following picture shows why.



Assume that w, x, y, and z each have at least six other interferences, and that all of d's interferences are shown. If we apply Chaitin's algorithm to this graph, it will spill d (assuming d is picked by the spill metric; recall the discussion on *simplify* in Section 2). Why? There exists a coloring of w, x, y, and z, suggested above, that can preclude d's allocation. This possibility is accounted for by the "extra" edges in the interference graph – the second edge from d to each of the other nodes. Because it makes spill decisions before any color is assigned, Chaitin's allocator must spill pairs whenever there is a chance that they may not be assigned a color. Thus, in any region where there is strong competition for registers and a mixture of single registers and register pairs, it will prejudicially spill the register pairs.

4 Another Coloring Algorithm

Originally, we thought that this problem was endemic to all coloring allocators. Fortunately, a colleague at IBM asked us a question that caused us to reconsider the problem of adjacent pairs in the context of another allocator. In 1989, we published a simple modification to Chaitin's allocator that improved its behavior on a wide variety of programs [2]. Figure 2 shows the layout of that allocator. It differs from the allocator depicted in Figure 1 in the placement of the edge leading to *spill code*. The revised control flow reflects two fundamental changes.

1. *Simplify* constructs an ordering of the nodes by removing them from the graph and placing them on a stack. It removes nodes with degree less than k and pushes them on the stack. When no such node

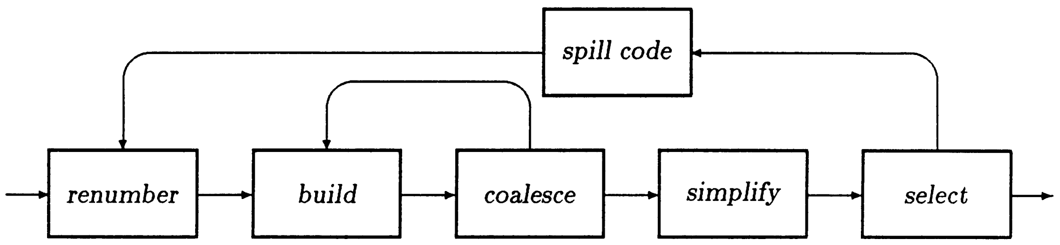


FIGURE 2: Optimistic Allocator

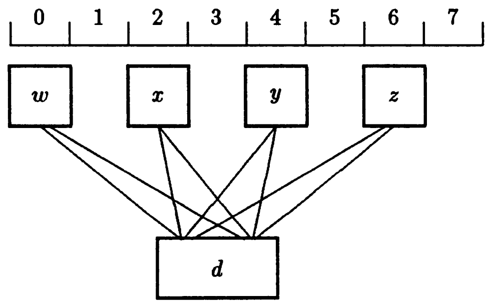
remains, it selects a node (the same node that Chaitin's allocator would select). Rather than spill that node, our allocator pushes it on the stack and continues.

2. *Select* tries to color the nodes on the stack. The change in *simplify* means that *select* can be invoked on a stack that does not color – it may encounter a node for which there is no remaining color. If this happens, it leaves that node uncolored and continues. If all nodes receive colors, the allocator has succeeded. If one or more nodes remain uncolored, it invokes *spill code* to spill those live ranges; it then begins the allocation process again, starting with *renumber*.

We call this an *optimistic* heuristic; in contrast, Chaitin's method is *pessimistic.*

The optimistic allocator behaves very differently than the pessimistic allocator with respect to spilling adjacent pairs. Because it defers spill decisions into *select*, it only spills a node when it discovers that it cannot color the node. With adjacent pairs, the behavior is the same; it only spills a pair when it discovers that no adjacent pair is available.

This results in a fair allocation. To see this more clearly, reconsider the graph that caused problems in Section 3.2.

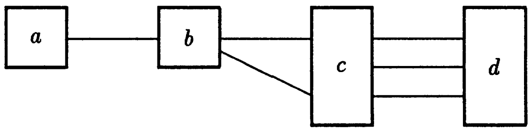


A pessimistic allocator would spill d because the single-precision values w, x, y, and z might be assigned to registers in a way that precludes successful coloring of d. An optimistic allocator would simply push d on the stack, hoping that the actual coloring of w, x, y, and z leaves a pair available for d. This can happen in many ways; for example, w and x might be assigned the same color, y might be spilled, or y and z might be assigned to consecutive registers. In fact, the *only* way that d could be blocked is by an even spacing of the sort suggested by the figure.

The optimistic allocator often succeeds on graphs where the pessimistic allocator fails. *Simplify* determines an order for assigning colors; it treats single nodes and pairs identically. (The difference between them is encoded in the number of edges.) *Select* makes the actual spill decisions; it spills a node only after discovering that it cannot find the needed color(s).

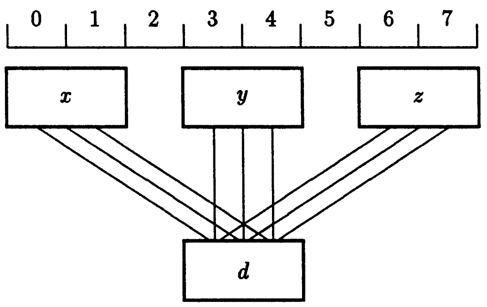
5 Unaligned Pairs

A few architectures allow the use of unaligned adjacent pairs of registers. The interference graph required for this situation is slightly more complex than for the aligned case. For our continuing example, the following interference graph captures all of the needed properties:

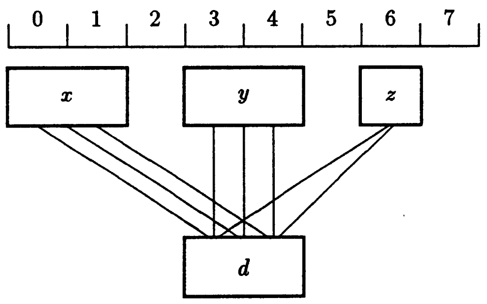


The extra edge (between *c* and *d*) is required to correctly trigger the selection of a spill candidate in *simplify*. The next two graphs help show why this third edge is necessary between unaligned pairs.

Note that three pairs (x, y, and z) can be colored so that there is no adjacent pair of colors for *d*. The fact that *d* has nine edges will trigger the spill heuristic in *simplify*, causing it to select a spill candidate. Of course, the candidate will not necessarily be spilled – this is the key difference between the optimistic and pessimistic approaches.



Even two pairs and a single may be placed so that *d* cannot be colored, as shown below. Again, *d*'s eight edges are sufficient to warn that *d* may have to be spilled during *select*.



Finally, we note that requiring four edges between each interfering pair would be too conservative. This would suggest that only two pairs (say x and y) would suffice to preclude coloring *d*.

These examples show that the extra flexibility offered by eliminating alignment restrictions complicates the allocation process by enlarging the graph. Furthermore, because of the additional constraints that it adds, it can actually lead to worse allocations.

6 Using Pairs for Memory Access

In some cases, it is desirable to use adjacent registers for loads and stores. For example, complex numbers are often represented in storage as a pair of adjacent single-precision floating-point numbers. On some machines, it is advantageous to load these values into an adjacent register pair using a double-precision floating-point load. Unfortunately, tying all subsequent uses of the component parts of the complex number to the adjacent registers restricts the allocator's freedom. Our compiler handles this issue by carefully shaping the code before the allocator sees it. It generates a double-precision load into an adjacent pair of virtual registers, and then immediately copies the component values into single registers. This allows the allocator to keep the values in adjacent registers at points where they are loaded and stored, while offering it the chance to keep them in non-adjacent registers during the rest of their lifetimes. It decides between these choices during coalescing, based on the structure of the interference graph. The allocator is the proper place to make this decision – it relies on information that cannot be made available earlier in compilation.

These ideas may become more significant in the future. Architects can make more memory bandwidth accessible through the use of wider load and store instructions. For example, on Intel's i860 XP, the quad-word, floating-point load `fld.q` loads four registers at a time, allowing a program to move twice as much data as the double-word version `fld.d` and four times as much as the single-word version `fld.l` [11]. Naturally, any appreciable use of this feature ties down a large number of registers and allocating them carefully becomes very important.

7 Conclusions

This paper examines the problem of dealing with register pairs in a graph coloring register allocator. We have shown how to represent several flavors of register pairs: unconstrained pairs, adjacent pairs, and unaligned adjacent pairs. Our scheme extends in a straightforward way to larger aggregate register groupings. We have shown that pessimistic allocators have trouble producing fair allocations for target machines that require the use of adjacent register pairs. We have shown that our optimistic allocator produces a fair allocation – spill decisions are based on estimated cost to spill and the actual availability of registers. Where the pessimistic allocator preferentially spills pairs, the optimistic allocator shows no preference. Previous work has shown that the optimistic scheme produces better allocations than earlier pessimistic schemes [2]; this paper shows that optimism also simplifies the treatment of register pairs.

Acknowledgements

Randy Scarborough prompted our re-examination of this topic; his timely questions always prove to be enlightening. Peter Markstein and Ken Kennedy have encouraged us throughout our work on register allocation with their comments and support. Financial support for this work was provided by IBM and the Defense Advanced Research Projects Agency.

References

[1] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* SIGPLAN Notices, 24(7), pages 258–263, July 1989.

[2] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* SIGPLAN Notices, 24(7), pages 275–284, July 1989.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction,* SIGPLAN Notices, 17(6), pages 98–105, June 1982.

[4] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, Jan. 1981.

[5] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,* SIGPLAN Notices, 19(6), pages 222–232, June 1984.

[6] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Prog. Lang. Syst.*, 12(4):501–536, Oct. 1990.

[7] J. Fabri. Automatic storage optimization. In *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction,* SIGPLAN Notices, 14(8), pages 83–91, Aug. 1979.

[8] J. Fabri. *Automatic Storage Optimization.* UMI Research Press, Ann Arbor, Michigan, 1982.

[9] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[10] M. E. Hopkins. Compiling for the RT PC ROMP. In *IBM RT Personal Computer Technology*, pages 76–82. IBM, 1986.

[11] Intel Corporation. *i860™ XP Microprocessor*, 1991.

[12] B. R. Nickerson. Graph coloring register allocation for processors with multi-register operands. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation,* SIGPLAN Notices, 25(6), pages 40–52, June 1990.

[13] R. Sethi. Complete register allocation problems. *SIAM J. Comput.*, 4(3):226–248, 1975.