Iterated Register Coalescing

Lal George
AT&T Bell Labs, rm. 2A426
600 Mountain Avenue
Murray Hill, NJ 07974
george@research.att.com

Andrew W. Appel
Dept. of Computer Science
Princeton University
Princeton, NJ 08544-2087
appel@princeton.edu

PRINCETON CS-TR-498-95
August 15, 1995

Abstract

An important function of any register allocator is to target registers so as to eliminate copy instructions. Graph-coloring register allocation is an elegant approach to this problem. If the source and destination of a move instruction do not interfere, then their nodes can be coalesced in the interference graph. Chaitin's coalescing heuristic could make a graph uncolorable (i.e., introduce spills); Briggs et al. demonstrated a conservative coalescing heuristic that preserves colorability. But Briggs's algorithm is *too* conservative, and leaves too many move instructions in our programs. We show how to interleave coloring reductions with Briggs's coalescing heuristic, leading to an algorithm that is safe but much more aggressive.

1 Introduction

Graph coloring is a powerful approach to register allocation and can have a significant impact on the execution of compiled code. A good register allocator does copy propagation, eliminating many move instructions by "coloring" the source temporary and target temporary of a move with the same register. Having copy propagation in the register allocator often simplifies code generation. The generation of target machine code can make liberal use of temporaries, and function call setup can naively move actuals into their formal parameter positions, leaving the register allocator to minimize the moves involved.

Optimizing compilers can generate a large number of move instructions. In static single assignment (SSA) form[9], each variable in the intermediate form may be assigned into only once. To satisfy this invariant, each program variable is split into several different temporaries that are live at different times. At a join point of program control flow, one temporary is copied to another as specified by a "ϕ-function." The SSA transformation allows efficient program optimization, but for good performance these artificial moves must later be removed by good register allocation.

Even non-SSA based compilers may generate a large number of move instructions. At a procedure call, a caller copies actual parameters to formals; then upon entry to the procedure, the callee moves formal parameters to fresh temporaries at procedure entry. The formal parameters themselves need not be fixed by a calling convention; if a function is local and all its call sites are identifiable, the formals may be temporaries to be colored (assigned to machine registers) by a register allocator[12]. Again, copy propagation is essential.

Briggs et al. [4] conjecture that SSA is a good heuristic for splitting live ranges—to minimize spills at the cost of a few extra moves—and our experience (with a similar intermediate

representation, continuation-passing style) bears them out. The copying done in the calling conventions described in the previous paragraph is very similar to the behavior of ϕ-functions in SSA.

These techniques generate large numbers of temporaries each with a small "live range." This is good, because each temporary interferes with (is live at the same time as) relatively few other (small-live-range) temporaries. A different approach to code generation would do copy propagation before register allocation; but the result will be variables that are live for a very long time, leading to many interferences with other long-lived variables. Then spilling (writing registers to memory) will be necessary.

Our new result can be stated concisely: Interleaving Chaitin-style simplification steps with Briggs-style conservative coalescing eliminates many more move instructions than Briggs's algorithm, while still guaranteeing not to introduce spills. Consider the interference graph of Figure 3. Briggs's *conservative coalescing* heuristic, as we will explain, cannot coalesce the move-related pair `j` and `b`, or the pair `d` and `c`, because each pair is adjacent to too many high-degree nodes. Our new algorithm first simplifies the graph, resulting in the graph of Figure 4(a). Now each move-related pair can be safely coalesced, because simplification has lowered the degree of their neighbors.

2 Graph coloring register allocation

Chaitin et al. [6, 7] abstracted the register allocation problem as a graph coloring problem. Nodes in the graph represent *live ranges* or temporaries used in the program. An edge connects any two temporaries that are simultaneously live at some point in the program, that is, whose live ranges *interfere*. The graph coloring problem is to assign colors to the nodes such that two nodes connected by an edge are not assigned the same color. The number of colors available is equal to the number of registers available on the machine. K-coloring a general graph is NP-complete [10], so a polynomial-time approximation algorithm is used.

There are five principal phases in a Chaitin-style graph coloring register allocator:

1. *Build:* Construct the interference graph. Dataflow analysis is used to compute the set of registers that are simultaneously live at a program point, and an edge is added to the graph for each pair of registers in the set. This is repeated for all program points.
2. *Coalesce:* Remove unnecessary move instructions. A move instruction can be deleted from the program when the source and destination of the move instruction do not have an edge in the interference graph. In other words, the source and destination can be coalesced into one node, which contains the combined edges of the nodes being replaced.

   When all possible moves have been coalesced, rebuilding the interference graph for the new program may yield further opportunities for coalescing. The build-coalesce phases are repeated until no moves can be coalesced.
3. *Simplify:* Color the graph using a simple heuristic[11]. Suppose the graph G contains a node m with fewer than K neighbors, where K is the number of registers on the machine. Let G' be the graph $G - \{m\}$ obtained by removing m. If G' can be colored, then so can G, for when adding m to the colored graph G', the neighbors of m have at most $K - 1$ colors among them; so a free color can always be found for m. This leads naturally to a stack-based algorithm for coloring: repeatedly remove (and push on a stack) nodes of degree less than K. Each such simplification will decrease the degrees of other nodes, leading to more opportunity for simplification.
4. *Spill:* But suppose at some point during simplification the graph G has nodes only of *significant degree*, that is, nodes of degree $\geq K$. Then the *simplify* heuristic fails, and a node is marked for spilling. That is, we choose some node in the graph (standing for a temporary variable in the program) and decide to represent it in memory, not registers,

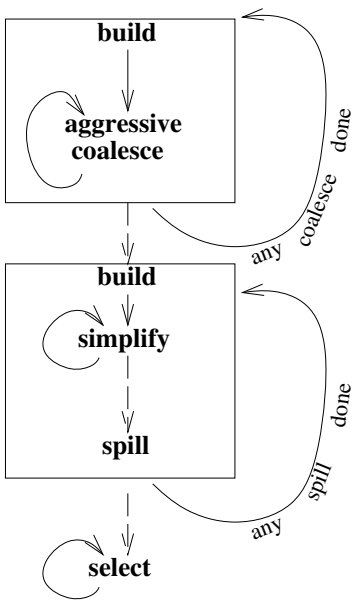


Figure 1: Flowchart of Chaitin graph coloring algorithm

during program execution. An optimistic approximation to the effect of spilling is that the spilled node does not interfere with any of the other nodes remaining in the graph. It can therefore be removed and the simplify process continued.

In fact, the spilled node must be fetched from memory just before each use; it will have several tiny live ranges. These will interfere with other temporaries in the graph. If, during a simplify pass, one or more nodes are marked for spilling, the program must be rewritten with explicit fetches and stores, and new live ranges must be computed using dataflow analysis. Then the *build* and *simplify* passes are repeated. This process iterates until *simplify* succeeds with no spills; in practice, one or two iterations almost always suffice.

5. *Select:* Assigns colors to nodes in the graph. Starting with the empty graph, the original graph is built up by repeatedly adding a node from the top of the stack. When a node is added to the graph, there must be a color for it, as the premise for it being removed in the simplify phase was that *it could always be assigned a color provided the remaining nodes in the graph could be successfully colored.*

Figure 1 shows the flowchart for the Chaitin graph-coloring register allocator.[6, 7] Dotted lines are control flow paths that are taken when none of the paths in solid lines apply.

Example

An example program is shown in Figure 2 and its interference graph in Figure 3. The nodes are labeled with the temporaries they represent, and there is an edge between two nodes if they are simultaneously live. For example, nodes `d`, `k`, and `j` are all connected since they are live simultaneously at the end of the block. Assuming that there are four registers available on the machine, then the simplify phase can start with the nodes `g`, `h`, `c`, and `f` in its working set, since they have less than four neighbors each. A color can always be found for them if the remaining graph can be successfully colored. If the algorithm starts by removing `h` and `g`, and all their edges, then node `k` becomes a candidate for removal and can be added to the worklist. Figure 4(a) shows the state of the graph after nodes `g`, `h`, and `k` have been removed. Continuing in this fashion a possible order in which nodes are removed is represented by the stack shown in Figure 4(b), where the stack grows upwards.

The nodes are now popped off the stack and the original graph reconstructed and colored simultaneously. Starting with `m`, a color is chosen arbitrarily since the graph at this point consists of a singleton node. The next node to be put into the graph is `c`. The only constraint is that it be given a color different from `m`, since there is an edge from `m` to `c`. When the original graph has been fully reconstructed, a possible assignment for the colors is shown in Figure 4(c).

3 Coalescing

It is easy to eliminate redundant move instructions with an interference graph. If there is no edge in the interference graph between the source and destination of a move instruction, then the move can be eliminated. The source and destination nodes are coalesced into a new node whose edges are the union of those of the nodes being replaced.

Chaitin[7] coalesced any pair of nodes not connected by an interference edge—avoiding coalescing with real machine registers "where possible." This aggressive form of copy propagation is very successful at eliminating move instructions. Unfortunately, the node being introduced is more constrained than those being removed, as it contains a union of edges. Thus, it is quite possible that a graph, colorable with K colors before coalescing, may no longer be K-colorable after reckless coalescing.

If some nodes are "pre-colored"—assigned to specific machine registers before register allocation (because they are used in calling conventions, for example), they cannot be spilled. Some coloring problems with pre-colored nodes have no solution: if a temporary interferes with K

```
liveIn: k j
      g := mem[j+12]
      h := k - 1
      f := g * h
      e := mem[j+8]
      m := mem[j+16]
      b := mem[f]
      c := e + 8
      d := c
      k := m + 4
      j := b
      goto d
liveOut: d k j
```

Figure 2: Example program

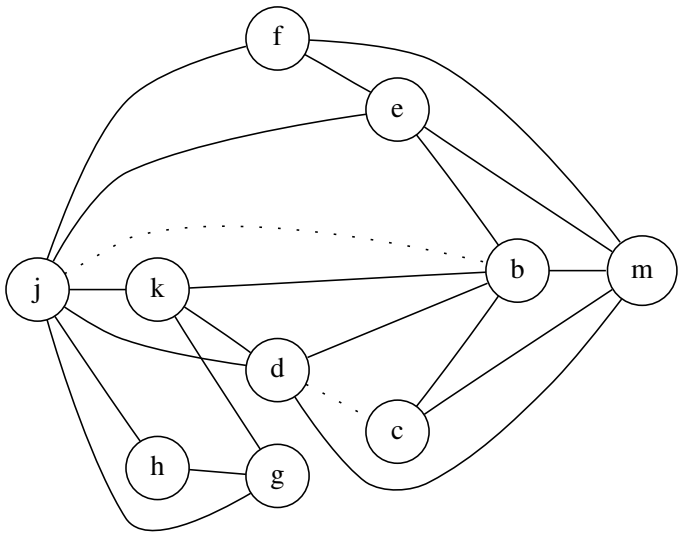
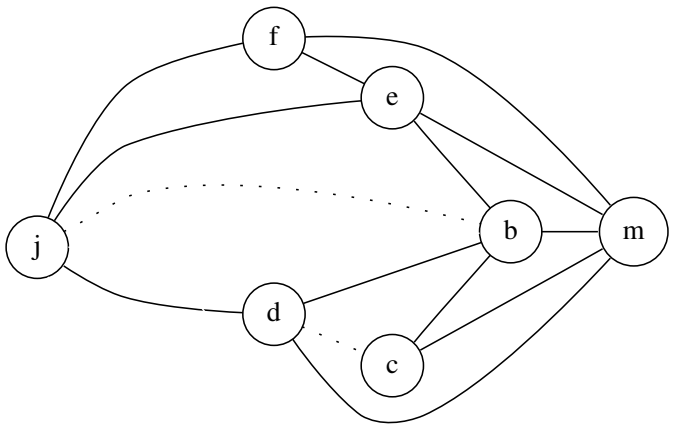


Figure 3: Interference graph
Dotted lines are not interference edges but indicate move instructions.



stack	assignment
m	1
c	3
b	2
f	2
e	4
j	3
d	4
k	1
h	2
g	4

(a) (b) (c)

Figure 4: (a) shows the intermediate graph after removal of nodes `h`, `g`, and `k`; (b) shows the stack after all nodes have been removed; and (c) is a possible assignment of colors.

pre-colored nodes (all of different colors), then the temporary must be spilled. But there is no register into which it can be fetched back for computation! We say such a graph is uncolorable, and we have found that reckless coalescing often leads to uncolorable graphs.

Briggs et al. [4] describe a *conservative* coalescing strategy that addresses this problem. If the node being coalesced has fewer than K neighbors of *significant* degree, then coalescing is guaranteed not to turn a K-colorable graph into a non-K-colorable graph. A node of significant degree is one with K or more neighbors. The proof of the guarantee is simple: after the simplify phase has removed all the insignificant-degree nodes from the graph, the coalesced node will be adjacent only to those neighbors that were of significant degree. Since these are less than K in number, *simplify* can remove the coalesced node from the graph. Thus if the original graph was colorable, the conservative coalescing strategy does not alter the colorability of the graph.

The strategy is conservative because a graph may still be colorable (using the heuristic), when a coalesced node has more than K neighbors of significant degree.

Conservative coalescing is successful at removing many move instructions without introducing spills (stores and fetches), but Briggs found that some moves still remain. For these he used a *biased coloring* heuristic during the *select* phase: When coloring a temporary X that is involved in a move instruction $X \leftarrow Y$ or $Y \leftarrow X$ where Y is already colored, the color of Y is selected if possible. Or, if Y is not yet colored, then a color is chosen that might later be eligible for the coloring of Y. If X and Y can be given the same color (assigned to the same register), then no move instruction will be necessary.

In Figure 3 nodes `c`, `d`, `b` and `j` are the operands of move instructions. Using the conservative coalescing strategy, these nodes cannot be coalesced. Coalescing `b` and `j` would produce a node with four significant-degree neighbors, namely `m`, `d`, `e`, and `k`. However, during the selection phase it is possible to bias the coloring so that these nodes get the same color. Therefore when coloring `j`, the color of `b` is given preference. If `b` has not been colored yet, then an attempt is made to prohibit the colors used by neighbors of `b`, to enhance the possibility of coalescing later.

The success of biased color selection is based on chance. In our example, `b` happened to be colored first with the register `r2`, and `f` was also assigned the same register, thus prohibiting

the choice of `r2` for node `j`. Therefore, the move between `b` and `j` cannot be eliminated. If `f` had been assigned another register, then the move could have been eliminated. This type of lookahead is expensive. For similar reasons the move between `c` and `d` cannot be eliminated. In the example of Figure 3 none of the moves were eliminated using either conservative coalescing or biased selection.

Although conservative coalescing cannot cause a node to spill, it can increase the number of stores and fetches required for the spilled node. Suppose the graph G contains nodes X and Y; X is rarely accessed but has a long live range, and Y is frequently accessed. And suppose, without coalescing, that X spills and Y does not. If coalescing is done before spilling, then the combined live range XY spills *and* is fetched frequently. To avoid this problem, Briggs does not coalesce until after one pass of *simplify* has been done to find the spills.

In our algorithm, if X might potentially be a candidate for spilling, then it will never be a candidate for coalescing. But we believe the same to be true of Brigg's algorithm, so we do not understand why Briggs should need to do this extra pass.

Briggs et al. [4] introduced *optimistic coloring*, which reduces the number of spills generated. In the *simplify* phase, when there are no low-degree nodes, instead of marking a node for spilling they just remove it from the graph and push it on the stack. This is a *potential spill*. Then the *select* phase may find that there is no color for the node; this is a spill. But in some cases *select* may find a color because the K (or more) neighbors will be colored with fewer than K distinct colors. Figure 5 shows the flow of control in Briggs's register allocator.

4 Difficult coloring problems

Graph-coloring register allocation is now the conventional approach for optimizing compilers. With that in mind, we implemented an optimizer for our compiler (Standard ML of New Jersey [2]) that generates many short-lived temporaries with enormous numbers of move instructions. Several optimization techniques contribute to register pressure. We do optimization and register allocation over several procedures at once. Locally defined procedures whose call sites are known can use specially selected parameter temporaries [12, 1, 8]. Free variables of nested functions can turn into extra arguments passed in registers [12, 1]. Type-based representation analysis [13, 15] spreads an n-tuple into n separate registers, especially when used as a procedure argument or return value. Callee-save register allocation [8] and callee-save closure analysis [3, 14] spread the calling context into several registers.

Our earlier phases have some choice about the number of simultaneously live variables they create. For example, representation analysis can avoid expanding large n-tuples, closure analysis can limit the number of procedure parameters representing free variables, and callee-save register allocation can use a limited number of registers. In all these cases, our optimization phases are guided by the number of registers available on the target machine. Thus, although they never assign registers explicitly, they tend to produce register allocation problems that are as hard as possible, but no harder: they don't spill much, yet there are often $K - 1$ live variables.

In implementing these optimization techniques, we assumed that the graph-coloring register allocator would be able to eliminate "all" the move instructions and assign registers without too much spilling. But instead we found that Chaitin's reckless coalescing produced too many spills, and Briggs's conservative coalescing left too many move instructions. It seems that our register-allocation and copy-propagation problems are more difficult than those produced by the FORTRAN compilers measured by Briggs.

Our measurements of realistic programs show that conservative coalescing eliminates only 24% of the move instructions; biased selection eliminates a further 39% (of the original moves), leaving 37% of the moves in the program. Our new algorithm eliminates all but 16% of the move instructions. This results in a speedup of 4.4% over programs compiled using the Briggs algorithm.

Briggs et al.

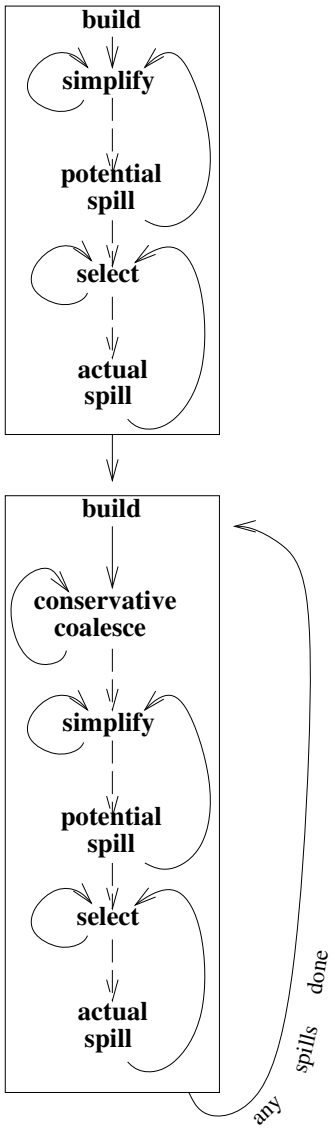


Figure 5: Briggs's algorithm

Iterated Coalescing

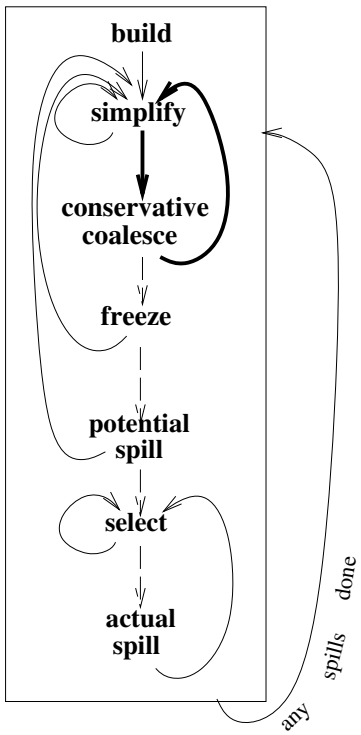


Figure 6: Iterated algorithm

5 Iterated register coalescing

Interleaving Chaitin-style simplification steps with Briggs-style conservative coalescing eliminates many more move instructions than Briggs's algorithm, while still guaranteeing not to introduce spills.

Our new approach calls the coalesce and simplify procedures in a loop, with *simplify* called first. The building blocks of the algorithm are essentially the same, but with a different flow of control shown in Figure 6. Our main contribution is the dark backward arrow. There are five principal phases in our register allocator:

1. *Build:* Construct the interference graph, and categorize each node as either *move-related* or not move-related. A move-related node is one that is the either the source or destination of a move instruction.
2. *Simplify:* One at a time, remove non-move-related nodes of low degree from the graph.
3. *Coalesce:* Perform Briggs-style conservative coalescing on the reduced graph obtained in the simplification phase. Since the degrees of many nodes has been reduced by *simplify*, the conservative strategy is likely to find many more moves to coalesce than it would have in the initial interference graph. After two nodes have been coalesced (and the move instruction deleted), if the resulting node is no longer move-related it will be available for the next round of simplification. *Simplify* and *Coalesce* are repeated until only significant-degree or move-related nodes remain.
4. *Freeze:* If neither *simplify* nor *coalesce* applies, we look for a move-related node of low degree. We *freeze* the moves in which this node is involved: that is, we give up hope of coalescing those moves. This causes the node (and perhaps other nodes related to the frozen moves) to be considered non-move-related. Now, *simplify* and *coalesce* are resumed.
5. *Select:* Same as before. Unlike Briggs, we do not use biased selection because it is not necessary with our improved coalescing heuristic.

Appendix A shows the algorithm in pseudocode.

Consider the initial interference graph shown in Figure 3. Nodes `b`, `c`, `d`, and `j` are the only move-related nodes in the graph. The initial worklist used in the simplify phase must contain only non-move related nodes, and consists of nodes `g`, h, and `f`. Node `c` is not included as it is move related. Once again, after removal of `g`, `h`, and `k` we obtain the graph in Figure 4(a).

We could continue the simplification phase further, however, if we invoke a round of coalescing at this point, we discover that `c` and `d` are indeed coalescable as the coalesced node has only two neighbors of significant degree — namely `m` and `b`. The resulting graph is shown in Figure 7, with the coalesced node labeled as `d&c`.

From Figure 7 we see that it is possible to coalesce `b` and `j` as well. Nodes `b` and `j` are adjacent to two neighbors of significant degree, namely `m` and `e`. The result of coalescing `b` and `j` is shown in Figure 8.

After coalescing these two moves, there are no more move-related nodes, and therefore no more coalescing possible. The simplify phase can be invoked one more time to remove all the remaining nodes. A possible assignment of colors is shown below:

stack	coloring
e	1
m	2
f	3
j&b	4
d&c	1
k	2
h	2
g	1

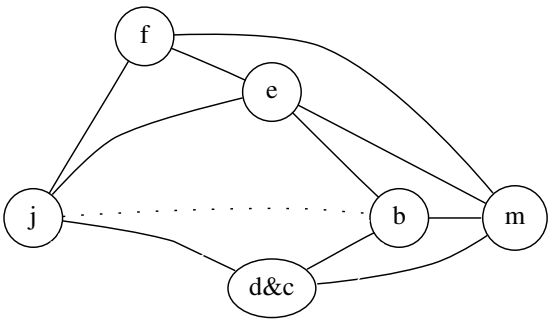


Figure 7: Interference graph after coalescing `d` and `c`

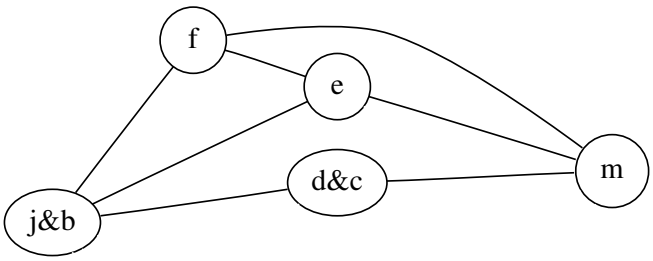


Figure 8: Interference graph after coalescing `b` and `j`

This coloring is a valid assignment for the original graph in Figure 3.

Theorem *Assume an interference graph G is colorable using the simplify heuristic. Conservative coalescing on an intermediate graph that is produced after some rounds of simplification of G produces a colorable graph.*

Definition *A simplified graph $S(G)$ is one in which some or all low-degree, non-move related nodes of G and their edges have been removed.*

Nodes that have been removed from a graph G cannot affect the colors of nodes that remain in $S(G)$. Indeed, they are colored after all nodes in $S(G)$ have been colored. Therefore, conservative coalescing applied to two nodes in $S(G)$ cannot affect the colorability of the original graph G. □

This technique is very successful: The first round of simplification removes such a large percentage of nodes that the conservative coalescing phase can usually be applied to all the move instructions in one pass.

Some moves are neither coalesced nor frozen. Instead, they are *constrained*. Consider the graph X, Y, Z, where (X, Z) is the only interference edge and there are two moves $X \leftarrow Y$ and $Y \leftarrow Z$. Either move is a candidate for coalescing. But after X and Y are coalesced, the remaining move $XY \leftarrow Z$ cannot be coalesced because of the interference edge (XY, Z). We say this move is *constrained*, and we remove it from further consideration: it no longer causes nodes to be treated as move-related.

Pessimistic or optimistic spilling

Our algorithm is compatible with either pessimistic or optimistic spilling. With Chaitin's pessimistic spilling, we guarantee not to introduce new spills. With optimistic spilling, we can only guarantee not to increase the number of *potential* spills; the number of actual spills might change.

If spilling is necessary, *build* and *simplify* must be repeated on the whole program. The simplest version of our algorithm discards any coalescings found if *build* must be repeated. Then it is easy to prove that coalescing does not increase the number of spills in any future round of *build*.

However, coalescing significantly reduces the number of temporaries and instructions in the graph, which would speed up the subsequent rounds of *build* and *simplify*. We believe it is safe to keep any coalescings done before the first spill node is removed from the graph. In the case of optimistic coloring, this means the first *potential* spill. We recommend this variant of the algorithm for increased speed, though we have shown the simpler algorithm in section A.3.

6 Graph coloring implementation

The main data structure used to implement graph coloring is the adjacency list representation of the interference graph. During the selection phase, the adjacency list is used to derive the list of neighbors that have already been colored, and during coalescing, two adjacency lists are unioned to form the coalesced node.

Chaitin and Briggs use a bit-matrix representation of the graph (that gives constant time membership tests) in addition to the adjacency lists. Since the bit matrix is symmetrical, they represent only one half of the matrix, so the number of bits required is $n(n+1)/2$. In practice, n can be large (for us it is often over 4000), so the bit matrix representation takes too much space. We take advantage of the fact that the matrix is sparse, and use a hash table of integer pairs. For a typical average degree of 16 and for $n = 4000$, the sparse table takes 256 Kbytes (2 words per entry, assuming no collisions) and the bit matrix would take 1 Mbyte.

Some of our temporaries are "pre-colored," that is, they represent machine registers. The front end generates these when interfacing to standard calling conventions across module boundaries, for example. Ordinary temporaries can be assigned the same colors as pre-colored reg-

isters, as long as they don't interfere, and in fact this is quite common. Thus, a standard calling-convention register can be re-used inside a procedure as a temporary.

The adjacency lists of machine registers are very large (see figure 10); because they're used in standard calling conventions they interfere with many temporaries. Furthermore, since machine registers are precolored, their adjacency lists are not necessary for the *select* phase. Therefore, to save space and time we do not explicitly represent the adjacency lists of the machine registers. The time savings is significant: when X is coalesced to Y, and X interferes with a machine register, then the long adjacency list for the machine register must be traversed to remove X and add Y.

In the absence of adjacency lists for machine registers, a simple heuristic is used to coalesce pseudo-registers with machine registers. A pseudo-register X can be coalesced to a machine register R, if for every T that is a neighbor of X, the coalescing does not increase the number of T's significant-degree neighbors from $< K$ to $\geq K$.

Any of the following conditions will suffice:

1. T already interferes with R. Then the set of T's neighbors gains no nodes.
2. T is a machine register. Since we already assume that all machine registers mutually interfere, this implies condition 1.
3. Degree(T) $< K$. Since T will lose the neighbor R and gain the neighbor T, then degree(T) will continue to be $< K$.

The third condition can be weakened to require T has fewer than $K-1$ neighbors of significant degree. This test would coalesce more liberally while still ensuring that the graph retains its colorability; but it would be more expensive to implement.

Associated with each move-related node is a count of the moves it is involved in. This count is easy to maintain and is used to test if a node is no longer move-related. Associated with all nodes is a count of the number of neighbors currently in the graph. This is used to determine whether a node is of significant degree during coalescing, and whether a node can be removed from the graph during simplification.

To make the algorithm efficient, it is important to be able to quickly perform each *simplify* step (removing a low-degree non-move-related node), each *coalesce* step, and each *freeze* step. To do this, we maintain four work lists:

- Low-degree non-move-related nodes *(simplifyWorklist)*;
- Coalesce candidates: move-related nodes that have not been proved uncoalesceable *(worklistMoves)*;
- Low-degree move-related nodes *(freezeWorklist)*.
- High-degree nodes *(spillWorklist)*.

Maintenance of these worklists avoids quadratic time blowup in finding coalesceable nodes.

7 Move-worklist management

When a node `X` changes from significant to low degree, the moves associated with its neighbors must be added to the move worklist. Moves that were blocked with too many significant neighbors (including `X`) might now be enabled for coalescing. Moves are added to the move worklist in only a few places:

- During simplify the degree of a node `X` might make the transition as a result of removing another node. Moves associated with neighbors of `X` are added to the *worklistMoves*.
- When coalescing `U` and `V`, there may be a node `X` that interferes with both `U` and `V`. The degree of `X` is decremented as it now interferes with the single coalesced node. Moves associated with neighbors of `X` are added. If `X` is move related, then moves associated with `X` itself are also added as both `U` and `V` may have been significant degree nodes.

Benchmark	Lines	Type	Description
knuth-bendix	580	INT	The Knuth-Bendix completion algorithm
vboyer	924	INT	The Boyer-Moore theorem prover using vectors
mlyacc	7422	INT	A parser generator, processing the SML grammar
nucleic	2309	FP	Nucleic acid 3D structure determination
simple	904	FP	A spherical fluid-dynamics program
format	2456	FP	SMLNJ formatting library
ray	891	FP	Ray tracing

Figure 9: Benchmark description

Benchmark	live ranges		average degree		instructions	
	machine	pseudo	machine	pseudo	moves	non-moves
knuth-bendix	15	5360	1296	13	4451	9396
vboyer	12	9222	4466	10	1883	20097
mlyacc:						
yacc.sml	16	6382	1766	12	5258	12123
utils.sml	15	3494	1050	14	2901	6279
yacc.grm.sml	19	4421	1346	11	2203	9606
nucleic	15	9825	4791	46	1621	27554
simple	19	10958	2536	15	8249	21483
format	16	3445	652	13	2785	6140
ray	15	1330	331	16	1045	2584

Figure 10: Benchmark characteristics

- When coalescing `U` to `V`, moves associated with `U` are added to the move worklist. This will catch other moves from `U` to `V`.

8 Benchmarks

For our measurements we used seven Standard ML programs, and SML/NJ compiler version 108.3 running on a DEC Alpha. A short description of each benchmark is given in Figure 9. Five of the benchmarks use floating point arithmetic, namely: `nucleic`, `simple`, `format`, and `ray`.

Some of the benchmarks consist of a single module, whereas others consist of multiple modules spread over multiple files. For benchmarks with multiple modules, we selected a module with a large number of live ranges. For the *mlyacc* benchmarks we selected the modules defined in the files `yacc.sml`, `utils.sml`, and `yacc.grm.sml`.

Each program was compiled to use six callee-save registers. This is an optimization level that generates high register pressure and very many move instructions. Previous versions of SML/NJ used only three callee-save registers, because their copy-propagation algorithms had not been able to handle six effectively.

Figure 10 shows the characteristics of each benchmark. Statistics of the interference graph are separated into those associated with machine registers and those with pseudo registers. *Live ranges* shows the number of nodes in the interference graph. For example, the `knuth-bendix`

Benchmark	Start	After 1 round	% Removed
knuth-bendix	5360	3432	36%
vboyer	9222	1231	87
yacc.sml	6382	3500	45
utils.sml	3494	2330	33
yacc.grm.sml	4421	1881	57
nucleic	9825	8077	18
simple	10958	5828	47
format	3445	2123	38
ray	1330	771	41

Figure 11: Number of nodes in the graph at start, and after one round of simplification.

Benchmark	**Nodes spilled**	**Instructions** spill	reload
knuth-bendix	0	0	0
vboyer	0	0	0
yacc.sml	0	0	0
utils.sml	17	17	35
yacc.grm.sml	24	24	33
nucleic	701	701	737
simple	12	12	24
format	0	0	0
ray	6	6	10

Figure 12: Spill statistics

program mentions 15 machine registers and 5360 pseudo registers. These numbers are inflated as the algorithm is applied to all the functions in the module at one time; in practise the functions would be applied to connected components of the call graph. The *average degree* column, indicating the average length of adjacency lists, shows that the length of adjacencies associated with machine registers is orders of magnitude larger than those associated with pseudo registers. The last two columns shows the total number of move and non-move instructions.

9 Results

During the first round of simplify, 46% of all nodes and their edges are removed. Figure 11 shows the number of nodes in the graph at the start of the algorithm and after the first round of simplification. This gives the first round of coalescing a significantly better chance than the original Briggs algorithm.

Figure 12 shows the spilling statistics. The number of spills—not surprisingly—are identical for both the iterated and Briggs's scheme. Most benchmarks do not spill at all. From among the programs that contain spill code, the number of *spill* instructions is almost equal to the number of *reload* instructions suggesting that the nodes that have been spilled may involve just one definition and use.

Figure 13 compares Briggs's and the iterated algorithms on the individual benchmarks. There are two omissions in our "Briggs" implementation over that shown in Figure 5: We did not execute one round of spilling before falling into the main body of the algorithm, and optimistic coloring was not implemented. Since spilling is rare in our compiler, it is unlikely that

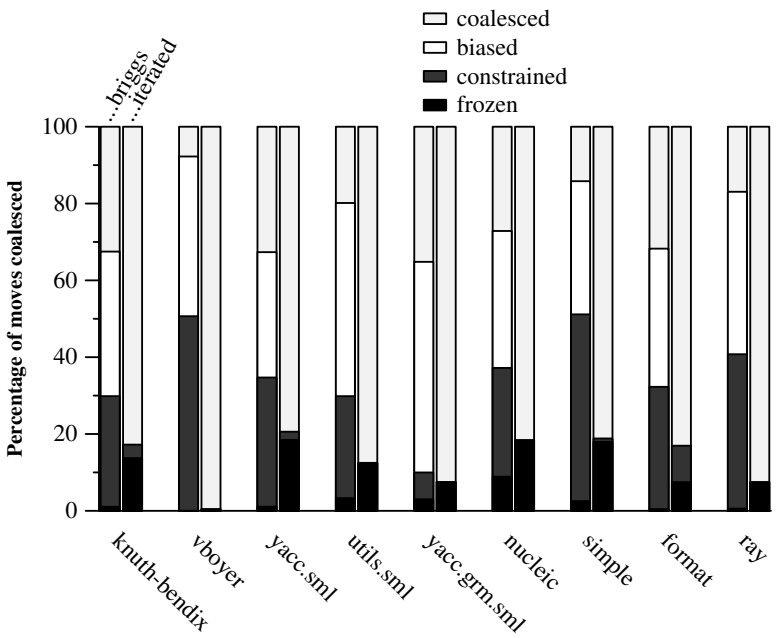


Figure 13: Comparison of moves coalesced by two algorithms
The black and dark-gray bars, labelled *frozen* and *constrained*, represent moves remaining in the program.

this omission will affect our results. For a fair comparison, we measured our iterated scheme without optimistic coloring. Referring to the bar charts for the Briggs allocator: *coalesced* are the moves removed using the conservative coalescing strategy; *constrained* are the moves that become constrained by having an interference edge added to them as a result of some other coalesce; *biased* are the moves coalesced using biased selection, and *frozen* are the moves that could not be coalesced using biased selection. On an average 24% of the nodes are removed in the coalesce phase and all the rest are at the mercy of biased selection. Considering all benchmarks together, 62% of all moves are removed.

For the iterated scheme *coalesced* and *constrained* have the same meaning as above, but *frozen* refers to the moves chosen by the *Freeze* heuristic. Biased selection was not implemented and so *biased* does not apply. More than 84% of all moves are removed with the new algorithm.

Figure 14 and 15 give more detailed numbers

The average improvement in code size is 5% (Figure 16). Since moves are the very fastest kind of instruction, we would expect that the improvement in speed would not be nearly this large. But taking the average timing from a series of 40 runs, we measured a surprising speedup average of 4.4% using the iterated scheme over Briggs allocation. Figure 17 shows the timings on the individual benchmarks. Each entry is the average of the sum of user, system, and garbage collection time. We believe that the significant speed improvement is partly due to the better I-cache performance of smaller programs.

There is a significant speed improvement when using six callee-save registers over three. The old register allocator in the SML/NJ compiler showed a degradation in performance when the number of callee-save registers was increased beyond three[3]. Appel and Shao attributed this to poor register targeting (copy propagation). The new compiler using iterated coalescing shows a distinct improvement when going from three to six callee-save registers, confirming Appel and Shao's guess. Use of a better register allocator now allows us to take full advantage of Shao's

Benchmark	**coalesced**	**constrained**	**biased**	**freeze**	% **coalesced**
knuth-bendix	1447	47	1675	1282	70%
vboyer	146	0	783	954	49
mlyacc:					
yacc.sml	1717	56	1716	1769	65
utils.sml	576	96	1459	770	70
yacc.grm.sml	775	66	1208	154	90
nucleic	440	144	578	459	63
simple	1170	209	2860	4010	49
format	884	12	1002	887	68
ray	177	6	442	420	59

Figure 14: Coalesce statistics for Briggs register allocator

Benchmark	**coalesced**	**constrained**	**freeze**	% **coalesced**
knuth-bendix	3684	611	156	83%
vboyer	1875	8	0	99
mlyacc:				
yacc.sml	4175	971	112	79
utils.sml	2539	362	0	88
yacc.grm.sml	2038	165	0	93
nucleic	1323	298	0	82
simple	6695	1482	72	81
format	2313	208	264	83
ray	967	78	0	93

Figure 15: Coalescing statistics for iterated register allocator

Benchmark	**Briggs**	**Iterated**	**Improvement**
knuth-bendix	42900	40652	5%
vboyer	84204	80420	4
yacc.sml	55792	52824	5
utils.sml	28580	26564	7
yacc.grm.sml	39304	39084	1
nucleic	112628	111408	1
simple	102808	92148	10
format	28156	26448	6
ray	12040	10648	11
Average			5%

Figure 16: Comparison of code size

Benchmark	**Briggs**	**Iterated**	**Improvement**
knuth-bendix	7.11	6.99	2%
vboyer	2.35	2.30	2
mlyacc	3.30	3.18	3
nucleic	2.91	2.59	11
simple	27.72	27.51	1
format	8.87	8.73	2
ray	49.04	44.35	10
Average			4.4%

Figure 17: Comparison of execution speed

Benchmark	Calleesave=3	Calleesave=6	Improvement
knuth-bendix	7.06 sec	6.99	1 %
vboyer	2.40	2.30	4
mlyacc	3.50	3.18	9
simple	28.21	27.51	2
format	8.76	8.73	0
ray	47.20	44.34	6

Figure 18: Execution speed comparing different calleesave registers

improved closure analysis algorithm [14]. Figure 18 shows the average execution time taken over 40 runs. All benchmarks show some improvement while *format* is the only benchmark that does not change with more callee-save registers.

10 Conclusions

Alternating the simplify and coalesce phases of a graph coloring register allocator eliminates many more moves than the older approach of coalescing before simplification. It ought to be easy to incorporate this algorithm into any existing implementation of graph-coloring-based register allocation, as it is easy to implement and uses the same building blocks.

A New algorithm in pseudocode

We give a complete implementation of the algorithm (Figure 6) in pseudocode. In this implementation, all coalescings are abandoned when an actual spill is detected.

A.1 Data structures

A.1.1 Node worklists, sets, and stacks

precolored — machine registers, preassigned a color.

initial — temporary registers, not preassigned a color and not yet processed by the algorithm.

simplifyWorklist — list of low-degree non-move-related nodes.

freezeWorklist — low-degree move-related nodes.

spillWorklist — high-degree nodes.

spilledNodes — nodes marked for spilling during this round; initially empty.

coalescedNodes — registers that have been coalesced; when the move $u{:=}v$ is coalesced, one of u or v is added to this set, and the other is put back on some worklist.

coloredNodes — nodes successfully colored.

selectStack — stack containing temporaries removed from the graph.

Invariant: These lists and sets are always *mutually disjoint* and every node is always in exactly one of the sets or lists. Since membership in these sets is often tested, the representation of each node should contain an enumeration value telling which set it's in.

Precondition: Initially (on entry to Main), and on exiting RewriteProgram, only the sets *precolored* and *initial* are non-empty.

A.1.2 Move sets

There are five sets of move instructions:

coalescedMoves — Moves that have been coalesced.

constrainedMoves — Moves whose source and target interfere.

frozenMoves — Moves that will no longer be considered for coalescing.

worklistMoves — Moves enabled for possible coalescing.

activeMoves — Moves not yet ready for coalescing.

Move Invariant: Every move is in exactly one of these sets (after Build through the end of Main).

A.1.3 Others

adjSet — The set of interference edges (u, v) in the graph. If $(u, v) \in$ adjSet then $(v, u) \in$ adjSet. We represent adjSet as a hash table of integer pairs.

adjList — adjacency list representation of the graph; for each non-precolored temporary u, adjList$[u]$ is the set of nodes that interfere with u.

degree — an array containing the current degree of each node.

Degree Invariant:

$$\text{For any } u \in \text{simplifyWorklist} \cup \text{freezeWorklist} \cup \text{spillWorklist}$$

it will always be the case that

$$\text{degree}(u) = |\text{adjList}(u) \quad \cap \quad (\text{precolored} \cup \text{simplifyWorklist} \cup \text{freezeWorklist} \cup \text{spillWorklist})|$$

moveList — a mapping from node to the list of moves it is associated with.

alias — When a move (u, v) has been coalesced, and v put in coalescedNodes, then alias$(v) = u$.

color — The color chosen by the algorithm for a node. For precolored nodes this is initialized to the given color.

simplifyWorklist Invariant:

$$(u \in \text{simplifyWorklist}) \Rightarrow \text{degree}(u) < K \wedge \text{moveList}[u] \cap (\text{activeMoves} \cup \text{worklistMoves}) = \{\}$$

freezeWorklist Invariant:

$$(u \in \text{freezeWorklist}) \Rightarrow \text{degree}(u) < K \wedge \text{moveList}[u] \cap (\text{activeMoves} \cup \text{worklistMoves}) \neq \{\}$$

spillWorklist Invariant:

$$(u \in \text{spillWorklist}) \Rightarrow \text{degree}(u) \geq K$$

A.2 Notes on the algorithm

Subsequent sections that outline the main phases of the algorithm should be read with reference to the implementation in Section A.3.

Main

The algorithm is invoked using the procedure `Main`, which loops (via tail recursion) until no spills are generated.

If `AssignColors` produces spills, then `RewriteProgram` allocates memory locations for the spilled temporaries, and inserts store and fetch instructions to access them. These stores and fetches are to newly created temporaries (albeit with tiny live ranges), so the Main loop must be performed on the altered graph.

Build

Procedure `Build` constructs the interference graph and bit matrix. We use the sparse set representation described by Briggs and Torczon[5] to implement the variable `live`. `Build` only adds an interference edge between a node that is defined at some point, and the nodes that are currently live at that point. It is not necessary to add interferences between nodes in the live set. These edges will be added when processing other blocks in the program.

Move instructions are given special consideration. It is important not to create artifical interferences between the source and destination of a move. Consider the program:

```
t := s                ; copy
...
x := ... s ...        ; use of s
...
y := ... t ...        ; use of t
```

After the copy instruction both `s` and `t` are live, and an interference edge would be added between `s` and `t` since `t` is being defined at a point where `s` is live. The solution is to temporarily remove `s` from the live set and continue. The pseudo code described by Briggs and Torczon[5] contains a bug, where `t` is removed from the live set instead of `s`.

The `Build` procedure also initializes the `worklistMoves` to contain all the moves in the program.

DecrementDegree

The process of removing a node from the graph involves decrementing the degree of its *current* neighbors. If the `degree` is already less than $K-1$ then the node must be move-related and is not added to the `simplifyWorklist`. When the degree of a node transitions from K to $K-1$, moves associated with its neighbors may be enabled.

Coalesce

Only moves in the `worklistMoves` are considered in the coalesce phase. When a move is coalesced, it may no longer be move related and can be added to the simplify worklist by the procedure `AddWorkList`. `OK` implements the heuristic used for coalescing a pre-colored register. `Conservative` implements the Briggs conservative coalescing heuristic.

Freeze

Procedure `Freeze` pulls out a node from the `freezeWorklist` and freezes all moves associated with this node. In principle, a heuristic could be used to select the freeze node. In our experience, freezes are not common, and a selection heuristic is unlikely to make a significant difference.

RewriteProgram

We show a variant of the algorithm in which all coalesces are discarded if the program must be rewritten to incorporate spill fetches and stores. But as Section 5 explains, we recommend keeping all the coalesces found before the first call to `SelectSpill`, and rewrite the program to eliminate the coalesced move instructions and temporaries.

A.3 Program code

```
procedure Main()                                                    ... Main
    LivenessAnalysis()
    Build()
    MkWorklist()
    repeat
        if simplifyWorklist ≠ {} then
            Simplify()
        else if worklistMoves ≠ {} then
            Coalesce()
        else if freezeWorklist ≠ {}  then
            Freeze()
        else if spillWorklist ≠ {} then
            SelectSpill()
    until simplifyWorklist = {}  ∧ worklistMoves = {}
          ∧ freezeWorklist = {}  ∧ spillWorklist = {}
    AssignColors()
    if spilledNodes ≠ {} then
        RewriteProgram(spilledNodes)
        Main()

procedure Build ()                                                  ... Build
    forall b ∈ blocks in program
        let live = liveOut(b)
        forall I ∈ instructions(b) in reverse order
            if isMoveInstruction(I) then
                live := live\use(I)
                forall n ∈ def(I) ∪ use(I)
                    moveList[n] := moveList[n] ∪ {I}
                worklistMoves := worklistMoves ∪ {I}
            forall d ∈ def(I)
                forall l ∈ live
                    AddEdge(l, d)
            live := use(I) ∪ (live\def(I))
```

```
procedure AddEdge(u, v)                                          ... AddEdge
    if (u, v) ∉ adjSet then
        adjSet := adjSet ∪ {(u, v), (v, u)}
        if u ∉ precolored then
            adjList[u] := adjList[u] ∪ {v}
            degree[u] := degree[u] + 1
        if v ∉ precolored then
            adjList[v] := adjList[v] ∪ {u}
            degree[v] := degree[v] + 1

function Adjacent(n)                                             ... Adjacent
    adjList[n] \ (selectStack ∪ coalescedNodes)

function NodeMoves (n)                                           ... NodeMoves
    moveList[n] ∩ (activeMoves ∪ worklistMoves)

function MoveRelated(n)                                          ... MoveRelated
    NodeMoves(n) ≠ {}

procedure MkWorklist()                                           ... MkWorklist
    forall n ∈ initial
        initial := initial \ {n}
        if degree[n] ≥ K then
            spillWorklist := spillWorklist ∪ {n}
        else if MoveRelated(n) then
            freezeWorklist := freezeWorklist ∪ {n}
        else
            simplifyWorklist := simplifyWorklist ∪ {n}
```

```
procedure Simplify()                                        ... Simplify
    let n ∈ simplifyWorklist
    simplifyWorklist := simplifyWorklist \ {n}
    push(n, selectStack)
    forall m ∈ Adjacent(n)
        DecrementDegree(m)

procedure DecrementDegree(m)                                ... DecrementDegree
    let d = degree[m]
    degree[m] := d-1
    if d = K then
        EnableMoves({m} ∪ Adjacent(m))
        spillWorklist := spillWorklist \ {m}
        if MoveRelated(m) then
            freezeWorklist := freezeWorklist ∪ {m}
        else
            simplifyWorklist := simplifyWorklist ∪ {m}

procedure EnableMoves(nodes)                                ... EnableMoves
    forall n ∈ nodes
        forall m ∈ NodeMoves(n)
            if m ∈ activeMoves then
                activeMoves := activeMoves \ {m}
                worklistMoves := worklistMoves ∪ {m}
```

```
procedure Coalesce()                                                    ... Coalesce
    let m_(=copy(x,y)) ∈ worklistMoves
    x := GetAlias(x)
    y := GetAlias(y)
    if y ∈ precolored then
        let (u, v) = (y, x)
    else
        let (u, v) = (x, y)
    worklistMoves := worklistMoves \ {m}
    if (u = v) then
        coalescedMoves := coalescedMoves ∪ {m}
        AddWorkList(u)
    else if v ∈ precolored ∨ (u, v) ∈ adjSet then
        constrainedMoves := constrainedMoves ∪ {m}
        addWorkList(u)
        addWorkList(v)
    else if u ∈ precolored ∧ (∀t ∈ Adjacent(v), OK(t, u))
          ∨ u ∉ precolored ∧ Conservative(Adjacent(u) ∪ Adjacent(v)) then
        coalescedMoves := coalescedMoves ∪ {m}
        Combine(u,v)
    else
        activeMoves := activeMoves ∪ {m}

procedure AddWorkList(u)                                                ... AddWorkList
    if (u ∉ precolored ∧ not(MoveRelated(u)) ∧ degree[u] < K) then
        freezeWorklist := freezeWorklist \ {u}
        simplifyWorklist := simplifyWorklist ∪ {u}

function OK(t,r)                                                        ... OK
    degree[t] < K ∨
    t ∈ precolored ∨
    (t,r) ∈ adjSet

function Conservative(nodes)                                            ... Conservative
    let k = 0
    forall n ∈ nodes
        if degree[n] ≥ K then
            k := k + 1
    return (k < K)
```

```
function GetAlias (n)                                          ... GetAlias
    if n ∈ coalescedNodes then
        GetAlias(alias[n])
    else n

procedure Combine(u,v)                                         ... Combine
    if v ∈ freezeWorklist then
        freezeWorklist := freezeWorklist \ {v}
    else
        spillWorklist := spillWorklist \ {v}
    alias[v] := u
    forall t ∈ Adjacent(v)
        DecrementDegree(t)
        AddEdge(t,u)

procedure Freeze()                                             ... Freeze
    let u ∈ freezeWorklist
    freezeWorklist := freezeWorklist \ {u}
    simplifyWorklist := simplifyWorklist ∪ {u}
    forall m(= copy(u,v) or copy(v,u)) ∈ NodeMoves(u)
        if m ∈ activeMoves then
            activeMoves := activeMoves \ {m}
        else
            worklistMoves := worklistMoves \ {m}
        frozenMoves := frozenMoves ∪ {m}
        if NodeMoves(v) = {} ∧ degree[v] < K then
            freezeWorklist := freezeWorklist \ {v}
            simplifyWorklist := simplifyWorklist ∪ {v}

procedure SelectSpill()                                        ... SelectSpill
    let m ∈ spillWorklist selected using favorite heuristic
        Note: avoid choosing nodes that are the tiny live ranges
        resulting from the fetches of previously spilled registers
    spillWorklist := spillWorklist \ {m}
    simplifyWorklist := simplifyWorklist ∪ {m}
```

```
procedure AssignColors()                                    ... AssignColors
    while SelectStack not empty
        let n = pop(SelectStack)
        okColors := {0, ..., K-1}
        forall w ∈ Adjacent(n) ∩ (coloredNodes ∪ precolored)
            okColors := okColors \ {color[w]}
        if okColors = {} then
            spilledNodes := spilledNodes ∪ {n}
        else
            coloredNodes := coloredNodes ∪ {n}
            let c ∈ okColors
            color[n] := c
    forall n ∈ coalescedNodes
        color[n] := color[GetAlias(n)]

procedure RewriteProgram()                                  ... RewriteProgram
    Allocate memory locations for each v ∈ spilledNodes,
    Create a new temporary v_i for each definition and each use,
    In the program (instructions), insert a store after each
    definition of a v_i, a fetch before each use of a v_i.
    Put all the v_i into a set newTemps.
    spilledNodes := {}
    initial := coloredNodes ∪ coalescedNodes ∪ newTemps
    coloredNodes := {}
    coalescedNodes := {}
```

References

[1] Appel, A. W. *Compiling with Continuations*. Cambridge Univ. Press, 1992. ISBN 0-521-41695-7.

[2] Appel, A. W., and MacQueen, D. B. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming* (New York, August 1991), M. Wirsing, Ed., Springer-Verlag, pp. 1–13.

[3] Appel, A. W., and Shao, Z. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation 5*, 3 (1992), 191–221.

[4] Briggs, P., Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. *ACM transactions on programming languages and systems 16*, 3 (May 1994), 428–455.

[5] Briggs, P., and Torczon, L. An efficient representation for sparse sets. *ACM Letters on programming languages and systesms 2*, 1-4 (March-December 1993), 59–69.

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

[7] 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* (January 1981), 47–57.

[8] Chow, F. C. Minimizing register usage penalty at procedure calls. In *Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation* (New York, June 1988), ACM Press, pp. 85–94.

[9] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on programming languages and systems 13*, 4 (October 1991), 451–490.

[10] Garey, M. R., and Johnson, D. S. *Computers and Intractability, A guide to the theory of NP-completeness*. Freeman, 1979. ISBN 0-7167-1044-7.

[11] Kempe, A. B. On the geographical problem of the four colors. *American Journal of Mathematics 2* (1879), 193–200.

[12] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction) 21*, 7 (July 1986), 219–33.

[13] Leroy, X. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages* (New York, January 1992), ACM Press, pp. 177–188.

[14] Shao, Z., and Appel, A. W. Space-efficient closure representations. In *Proc. 1994 ACM Conf. on Lisp and Functional Programming* (1994), ACM Press, pp. 150–161.

[15] Shao, Z., and Appel, A. W. A type-based compiler for Standard ML. In *Proc 1995 ACM Conf. on Programming Language Design and Implementation* (1995), ACM Press, pp. 116–129.