Optimistic Register Coalescing [*]

Jinpyo Park Soo-Mook Moon
School of Electrical Engineering
Seoul National University, Korea
{jp,smoon}@altair.snu.ac.kr

Abstract

Graph-coloring register allocators eliminate copies by coalescing the source and target node of a copy if they do not interfere in the interference graph. Coalescing is, however, known to be harmful to the colorability of the graph because it tends to yield a graph with nodes of higher degrees. Unlike aggressive coalescing which coalesces any pair of non-interfering copy-related nodes, conservative coalescing or iterated coalescing perform safe coalescing that preserves the colorability. Unfortunately, these heuristics give up coalescing too early, losing many opportunities of coalescing that would turn out to be safe. Moreover, they ignore the fact that coalescing may even improve the colorability of the graph by reducing the degree of neighbor nodes that are interfering with both the source and target nodes being coalesced. This paper proposes a new heuristic called optimistic coalescing which optimistically performs aggressive coalescing, thus fully exploiting the positive impact of coalescing, yet when a coalesced node is to be spilled, it is split back into separate nodes; there is a better chance of coloring one of them which reduces the overall spill cost.

Index Terms: *Register allocation, copy coalescing, instruction scheduling, renaming*

[*]This work was supported by STEPI in Korea under the MOST-FOTD project.

1. Introduction

Many optimizing compilers take the approach of *graph coloring* for global register allocation [1]. Graph coloring abstracts the problem of assigning registers to live ranges in a program into the problem of assigning colors to nodes in the *interference graph.* The register allocator attempts to "color" the graph with a finite number of machine registers, with one constraint that any two nodes connected by an interference edge must be colored with different registers. If the allocator fails to color the graph, some nodes must be *spilled* to memory by inserting loads and stores. Graph coloring with a minimal number of spills is a well-known NP-complete problem and many heuristic algorithms have been used [1, 2, 3].

One important task of a register allocator is copy propagation. In the context of graph coloring, copy propagation is achieved simply by coloring the source and target node of a copy (which we call *copy-related nodes*) with a single register if they do not interfere. On the interference graph, this is implemented by *coalescing* the two nodes into a single node, with their interference edges being unioned. Since many optimization phases before the register allocation including instruction scheduling [4], store-to-copy promotion [5], and static single assignment (SSA) translation [6] leave behind many copies that slow down program execution, it is essential to minimize those copies.

Coalescing may affect the colorability of the interference graph. Since a coalesced node will have the union of interference edges of the source and target nodes being coalesced, it might not be possible to color the coalesced node. This makes the register allocation problem more challenging because we need to minimize the spill cost while maximizing the number of coalesced copies. A couple of coalescing heuristics have been proposed to handle this optimization problem.

The original Chaitin's register allocator uses *aggressive coalescing* which coalesces any pair of non-interfering, copy-related nodes [1]. It obviously achieves the best result of copy elimination yet might suffer from the worst spill. Briggs et al. have developed a heuristic called *conservative coalescing* [2] which performs coalescing only when the colorability of the graph is not affected. George and Appel's *iterated coalescing* eliminates more copies by interleaving Briggs' heuristic with a coloring simplification phase which increases the chance of conservative coalescing [7].

While conservative coalescing or iterated coalescing attempt to avoid the negative impact of coalescing, it is often ignored that coalescing may even improve the colorability of the graph: if a node x interferes with both the source and target nodes being coalesced, the number of edges of x is reduced by one after the coalescing. This may yield additional opportunities for the simplification of the graph, improving the overall colorability. In order to fully exploit this positive impact of coalescing, we prefer to use aggressive coalescing except that the spill must be controlled somehow.

This paper proposes a new coalescing heuristic called *optimistic coalescing* which optimistically coalesces all pairs of non-interfering nodes as in Chaitin's heuristic, yet when a coalesced node needs to be spilled, the node is split back into two separate nodes, giving up the coalescing. Since each split node will have less number of interference edges than the original coalesced node, it might be possible to color one of them without affecting the rest of the coloring. Coloring one of them leads to a reduction in spill cost.

The rest of this paper is organized as follows. Section 2 briefly reviews the graph-coloring register allocation and Section 3 describes previous coalescing heuristics on the graph-coloring framework. Section 4 describes the optimistic coalescing heuristic. Our experimental results are presented in Section 5. A summary follows in Section 6.

2. Graph-Coloring Register Allocation

Graph coloring is a problem of assigning colors to the nodes of a given graph such that any two nodes connected by an edge have different colors. In graph-coloring register allocation, each node in the graph corresponds to a live range in the program and the number of colors represents that of machine registers. Two overlapping live ranges are said to be interfering and the corresponding nodes are connected by an edge.

Since coloring a general graph G with K colors is NP-complete, we need a heuristic algorithm. Chaitin et al. have implemented the graph-coloring register allocation based on the following theorem [1]:

A node x having less than K neighbors is always colorable no matter how G-$\{x\}$ is colored.

Such a node x is called a *low-degree* node while a node that has more than or equal to K neighbors is called a *significant-degree* node. Chaitin's allocator repeatedly removes low-degree nodes and their associated edges from the graph and pushes the nodes on top of the stack in the *simplify* phase until either the graph is empty or every node in the graph is significant. If the graph is empty, the allocator repeatedly takes nodes back from the top of the stack and assigns machine registers according to interference constraints in the *select* phase. However, if every node is significant, the allocator heuristically selects nodes to store their values in memory and marks them for spilling. The allocator inserts spill instructions at each def/use instruction of marked live ranges in the *spill* phase after which the allocation steps are repeated from scratch.

The phase ordering of the Chaitin's coloring algorithm is depicted in Figure 1 (a). The *renumber* phase and the *build* phase are also shown where the allocator constructs live ranges and builds the interference graph respectively (the phase of *aggressive coalescing* will be described in the next section).

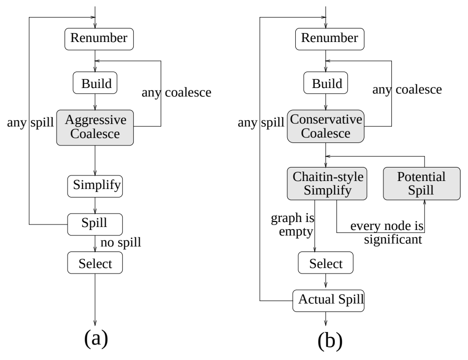


Figure 1. Phase ordering of (a) Chaitin's allocator and (b) Briggs' optimistic allocator.

In order to reduce the number of spilled nodes, Briggs et al. have introduced *optimistic coloring* [2]. When every node is significant, instead of marking a node for spilling, the allocator just removes and pushes it on the stack. This is called a *potential-spill*. During the *select* phase, if there indeed is no color for the node, it is *actually-spilled*. The idea is that there might be a chance that a color is available for the significant-degree node depending on the coloring of its neighbors, yet this opportunity is pessimistically given up too early in the Chaitin's algorithm. Figure 1 (b) shows the phase ordering of optimistic coloring[1].

[1]The original flow graph of Briggs' optimistic allocator in-

3. Approaches to Copy Coalescing

Coalescing includes both negative and positive impacts on the coloring of the interference graph. When two nodes are coalesced, the coalesced node will have a union of the edges of those being coalesced. This might make a K-colorable graph not K-colorable any more. On the other hand, if a node interferes with both nodes being coalesced, the degree of the node is reduced by one, which may expose further opportunities for simplifying the graph.

For example, consider the interference graphs in Figure 2 where a solid line represents an interference edge and a dotted line indicates a copy-related edge. If `a` and `b` are coalesced as in Figure 2 (a), a two-colorable graph becomes a graph that is not two-colorable any more. In Figure 2 (b), however, a graph that cannot be simplified when there are two colors can be simplified after coalescing `a` and `d`, making the graph two-colorable by Chaitin's coloring heuristic.

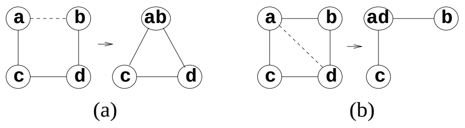


Figure 2. Examples of (a) negative impact and (b) positive impact of coalescing.

There have been two major heuristic approaches to coalescing. One is eliminating all coalescible copies, causing the side effect of fully exploiting the positive impact. Chaitin's aggressive coalescing corresponds to this. The other focuses on reducing the negative impact, which is the basic idea employed in conservative coalescing or iterated coalescing.

3.1. The Approach of Aggressive Coalescing

Chaitin's register allocator performs aggressive coalescing that coalesces any two non-interfering, copy-related nodes. From the perspective of copy elimination, aggressive coalescing achieves the best result. From the perspective of the coloring of the graph, it fully exploits the positive impact while it totally ignores the negative impact. Unfortunately, the advantage of the positive impact in aggressive coalescing has never been noted in literature, and it has never been questioned how the positive impact interacts with the negative impact, affecting the overall colorability of the graph. When aggressive coalescing is employed in Briggs' optimistic allocator, our experimental results indicate that the positive impact can well outweigh the negative impact, coloring the graph better than conservative coalescing or iterated coalescing (see Section 5).

cludes the *rematerialization* phase [2] which generates copies to be removed by conservative coalescing (see Section 3.2). It is omitted in our version of the optimistic allocator to evaluate conservative coalescing as a general coalescing heuristic, not bounded to a specific optimization.

3.2. The Approach of Conservative Coalescing

Briggs' optimistic allocator includes an optimization phase called *rematerialization* which reduces the spill cost of a live range if its value is cheaply reproducible from a constant [2]. Finding rematerializable values involves an SSA transformation which yields many copies at ϕ-nodes. These copies, unlike other copies that can be coalesced aggressively, must be coalesced "conservatively" because reckless coalescing would completely restore the transformation performed for rematerialization, and it is undesirable to coalesce the two nodes with different rematerialization tags if there is a chance for the coalesced node to be spilled. This has motivated the heuristic of conservative coalescing which is based on the following theorem [2]:

For given two nodes `x` and `y`, if the coalesced node `xy` has less than K significant-degree neighbors, `xy` does not change the colorability of the interference graph.

When every low-degree node has been removed from the graph, the coalesced node `xy` can certainly be removed also since it originally had less than K significant-degree neighbors. Consequently, a conservatively-coalesced node is never spilled.

It should be noted that some care should be taken in counting the number of significant-degree neighbors of `xy` since a simple union of significant-degree neighbors of individual `x` and `y` might include a node that becomes low-degree after coalescing `xy` because of the positive impact of coalescing. Let S_x and S_y be the sets of significant-degree neighbors of `x` and `y`, respectively. The number of elements in the set of significant-degree neighbors of `xy` ($n(S_{xy})$) is computed as follows:
$n(S_{xy}) = n(S_x \cup S_y) - n(\{z | z \in S_x \cap S_y, \text{degree } of\ z = K \text{ before the coalescing}\})^2$.

In order to complement the conservativeness of coalescing, *biased coloring* is used in the select phase [2]. Biased coloring eliminates copies by trying to assign the same register to copy-related nodes that have not

[2]It appears that the iterated coalescing algorithm in [7] ignores this term in computing the number of significant-degree neighbors, failing to perform conservative coalescing fully. In our implementation, we include this term for fair comparison.

been coalesced due to the conservativeness heuristic.

Iterated coalescing has been proposed to coalesce more copies conservatively [7]. It performs conservative coalescing iteratively, interleaved with the simplification phase which exposes more chances of conservative coalescing. Figure 3 depicts the phase ordering of iterated coalescing on top of the optimistic allocator.

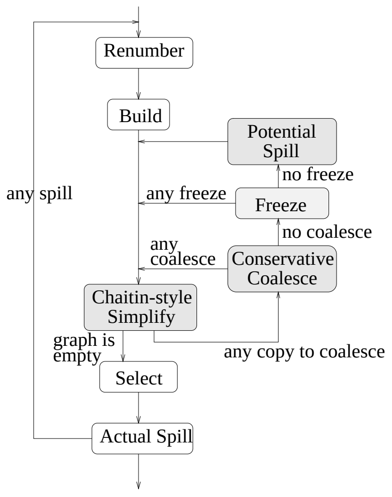


Figure 3. Phase ordering of iterated coalescing on top of optimistic allocator.

In iterated coalescing, the allocator maintains two groups of nodes. One group is a set of copy-related nodes and the other is a complementary set of non-copy-related nodes. Simplifying low-degree, non-copy-related nodes will decrease the degree of many copy-related nodes, yielding more opportunities for conservative coalescing. A coalesced node is put into the group of non-copy-related nodes if it has no more copy-related neighbor edges. When all non-copy-related nodes are significant-degree nodes and no two copy-related nodes are coalescible any more, the allocator *freezes* one of low-degree, copy-related nodes, if there is any. Freezing a copy-related node means giving up its coalescing by removing all of its copy-related edges and marking it as a non-copy-related node. A frozen node may be simplified in the next iteration. When all nodes in both groups are significant-degree nodes, a potential spill is made and the iteration of the simplification and the coalescing continues.

3.2.1 Problems with Conservative Coalescing

The problem with the heuristic of conservative coalescing is two-fold. First, the heuristic gives up coalescing too early considering that a coalescible node that violates the heuristic is not necessarily spilled. This is somewhat similar to the motivating idea of Briggs' optimistic allocator that a potentially-spilled node is not necessarily actually-spilled and thus, it is better to delay the spill decision later. We may also want to delay the spill concern later if many coalesced nodes that violate the heuristic are not actually-spilled, and if the spill cost can be held at a modest level in case they are actually-spilled. (Section 5.3 includes experimental data that justify delaying the spill concern later).

Secondly, the heuristic less exploits the positive impact of coalescing. Consider an example interference graph in Figure 4 (a). Let us assume that three colors are available and that `b` has the lowest spill cost. In this graph, since both conservative coalescing and iterated coalescing fail to coalesce `b` and `c`, `b` with the smallest spill cost is potentially spilled and pushed on the stack. Once `b` is removed from the graph, other nodes can be simplified and pushed on the stack as shown in the right. In the select phase, each node can be assigned a color (as shown in the right column of the stack) except for `b` which must be actually-spilled (if `e` were colored by 2 not 1, every node would be assigned a color, yet this is not always guaranteed).

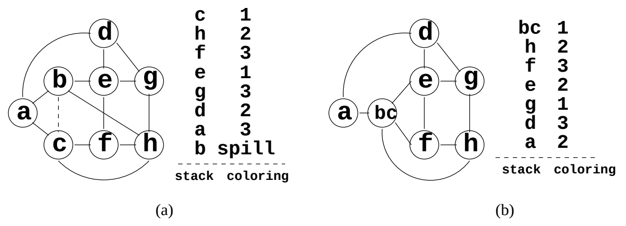


Figure 4. Allocation with (a) conservative coalescing and (b) aggressive coalescing.

However, coalescing `b` and `c` yields a low-degree node `a` as in Figure 4 (b) and allows every node in the graph to be simplified (a legal coloring without any spill is shown in the right column of the stack). The question is how often these opportunities arise in real programs.

4. Optimistic Register Coalescing

In order to fully exploit the positive impact of coalescing, we optimistically perform aggressive coalescing before the simplification phase as in Chaitin's register allocator, but we implement aggressive coalescing on top of Briggs' optimistic allocator to reduce the spill cost. When an actual spill needs to be made for a coalesced node, we attempt to reduce the spill cost by a technique called *live range splitting*.

Live range splitting is a spill cost reduction technique used by many optimizing compilers [2, 3, 8, 9, 10, 11]. A long live range is split into shorter ones by copies and load/stores inserted at carefully selected places. A register allocator can avoid spills by live range splitting since a shorter live range is more likely to be allocated registers than a longer one. The cost involved is the additional splitting instructions, yet is generally lower than that of spill instructions. Even if the allocator cannot avoid the spill completely, the spill cost is lower with live range splitting because only some splits are spilled while others are allocated registers.

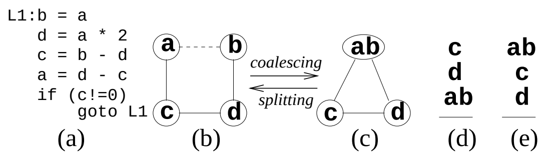
Unfortunately, live range splitting in global register allocation is not easy to implement due to several reasons [11, 12], the primary of which is the difficulty in deciding where to insert the splitting instructions.

In our context of reducing the spill cost of a coalesced node by live range splitting, it is straightforward where to insert the splitting copy: the original location where the coalesced copy was located! In this context, live range splitting is just the "undo" of coalescing. If a coalesced node `xy` needs to be spilled during the selection phase, it is split back into `x` and `y` while recovering the original interference edges of each. Now, it might be possible to color some of the two split nodes because the degree of each node is lower than that of `xy`. There are three possible cases for the coloring of those two split nodes.

First, each node is colorable with a different color. However, assigning different colors for them at this point of the select phase might ruin the rest of the coloring: if a node below `xy` in the stack has been simplified based on the fact that `x` and `y` are not distinct (e.g., a common neighbor node of `x` and `y` whose degree has been reduced after their coalescing, thus being simplified), the node might not be colorable any more because it now has more colored neighbors. Consequently, it is better to color only one node (preferably a node that has a higher spill cost) without affecting the rest of the coloring. We do not have to give up the coloring of the other node, though. Instead of marking the uncolored node for an actual spill at this moment, we can try again to find a color for it after processing all nodes left in the stack; if a color is available for the node at that point, it is safe to color it.

The second case is that `x` can be colored while `y` cannot (or vice versa). In this case, `y` is marked for the actual spill. Finally, none of `x` or `y` can be colored. In this case, we mark both of them for the actual spill (this is equal to spilling the original node `xy`).

For example, consider a loop in Figure 5 (a) whose interference graph is in Figure 5 (b). Let us assume that two colors are available. Optimistic coalescing will coalesce `a` and `b` "aggressively" as shown in Figure 5 (c). Since no nodes can be simplified, a potential spill should be made. If the coalesced node `ab` is chosen for the potential spill, the simplification result in the stack will be as in Figure 5 (d). After assigning two colors successfully to `c` and `d` in the selection phase, no color will be available for `ab`. Then, `ab` is split back which yields the same interference graph as in Figure 5 (b). Since both `a` and `b` are colorable and the stack is empty, both nodes can be colored safely.



```
L1:b = a
   d = a * 2
   c = b - d
   a = d - c
   if (c!=0)
     goto L1
```

Figure 5. An example of live range splitting.

Due to live range splitting, it is preferred to choose a coalesced node for the potential-spill. In Figure 5 (c), if `d` were chosen for the potential spill instead of the coalesced node `ab`, the stack would be formed as in Figure 5 (e). In this case, `ab` and `c` would be assigned colors first, which makes `d` be actually-spilled.

If there is more than one candidate coalesced node for the potential-spill, the one with more "ingredient" nodes is preferred because it has a better chance of being colored after the split (e.g., a coalesced node `xyz` that is composed of three nodes is preferred to `uv` that is composed of two nodes). This preference is included in our estimation of the spill cost as follows. For a given node `x`, let d, g, and n denote the degree of x, the number of ingredient nodes in x, and the number of use/def instructions of x, respectively. In addition, for each use/def instruction i, its spill cost and its loop nest are denoted by c_i and l_i, respectively. Then, the spill cost is computed as follows[3]: $\frac{\sum_i^n c_i 100^{l_i}}{10d+g}$.

When we have to split a coalesced node `xyz`, we first split it into three individual nodes `x`, `y`, and `z` and check if each can be colored individually; those nodes that cannot be colored are spilled right away at this point. For the remaining colorable nodes, we make every possible combination of a split and test if it can be colored. The order of the testing is based on the spill cost of each split. For example, if all of `x`, `y`, `z` can be colored individually and if their spill costs are 10, 5, and 3, respectively, we test splits in the

[3]In the above example, each of `ab`, `c`, `d` has one def and two use instructions after the copy is coalesced away. Assuming that the spill cost of both load and store (=c) and the loop nest level (=l) are all one, the spill cost of `ab` is $\frac{300}{22}$, which is smaller than that of either `c` or `d` which is $\frac{300}{21}$. Consequently, `ab` is chosen for the potential-spill.

following order: xy (15), xz (13), x (10), yz (8), y (5), z (3). If neither xy nor xz can be colored, only x is colored at this point while both y and z are set aside for later decision of coloring. After all nodes below the stack are processed, one of the following occurs for y and z: (1) both are colored by a single color, or (2) they are colored differently, or (3) one is colored while the other is spilled, or (4) both are spilled. If x, y, and z were colored with different colors, it means that coalescing is completely undone, with all of copies being recovered. For splitting a coalesced node with more than three nodes, the same procedure is applied. The phase ordering of optimistic coalescing on top of the optimistic register allocator is shown in Figure 6.

Regarding the spatial overhead of optimistic coalescing, the allocator should keep the information on the original interference edges of those nodes being coalesced due to possibility of live range splitting (it is freed right after coalescing in other heuristics). As to the temporal overhead, the testing of colorability for every possible combination of splits involves a computational complexity of $O(2^n)$, where n is the number of nodes being considered (in the previous example of splitting xyz, n is three). Our experiments indicate that n is only 1.87 on the average, yet it reaches up to 9. When n is large, we can avoid unnecessary tests by excluding redundant testing (e.g., we do not have to check if xyz is colorable when it is already known that xy is not colorable). The timing overhead of live range splitting can be compensated if it eliminates all actual-spills, obviating additional iterations of the allocation phases.

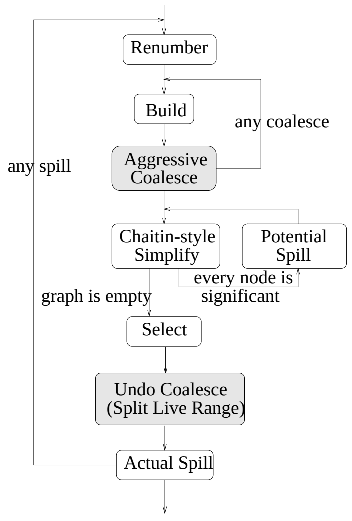


Figure 6. Phase ordering of optimistic coalescing on top of optimistic allocator.

5. Experimental Results

In order to evaluate the optimistic coalescing compared to previous coalescing heuristics, we have performed an empirical study in the context of instruction scheduling of non-numerical code. The register allocation problem in this context is highly challenging because aggressive code scheduling in the presence of complex control flows generates many copies and increases the register pressure. We have experimented with the four coalescing heuristics on this environment.

5.1. Experimental Environment

The experiments have been performed in a SPARC-based VLIW testbed [13]. The input C code is compiled into optimized SPARC assembly code (without register windows) by the *gcc* compiler. The SPARC-based assembly code is scheduled into high-performance VLIW code by aggressive scheduling techniques such as software pipelining, all-path code motion, and renaming [4]. The final VLIW code is simulated, producing execution results. Our benchmarks are composed of seven non-trivial integer programs such as eqntott, espresso, li, compress, yacc, sed, and gzip. The resource constraint of the VLIW machine is 16-ALUs and 8-way branching. The machine has 32 general-purpose registers and 16 condition registers which are targeted for both scheduling and register allocation.

For aggressive scheduling, the compiler renames registers using copies, on an as-needed basis when parallelism opportunities arise. For example, when x=x+4 cannot be moved up due to its target register x (e.g., x is live at the other target of a branch when we want to move the instruction speculatively above the branch), its target is renamed by x' and a copy instruction copy x=x' is left at the original place of the instruction. If the copy can later be eliminated, then this is equivalent to live range renaming; if not, it represents a more general way to get rid of non-true data dependences for instruction scheduling.

In the context of software pipelining, some of these copies are generated to circumvent cross-iteration register overwrites [14], and are not coalescible due to interferences in the code. In order to remove those interferences, we unroll the loop which makes many of those copies coalescible [15]. After the unrolling, we perform register allocation and coalescing which yields the same scheduled code with less copies. We experiment in this context of eliminating copies generated by instruction scheduling.

5.2. Coalescing and Spill Results

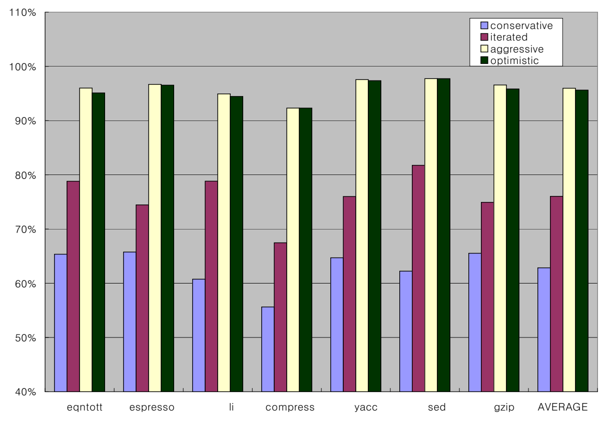


Figure 7. Ratio of eliminated copies compared to those of the base-case allocator.

We have implemented the four coalescing heuristics (conservative, iterated, aggressive, and optimistic) on top of optimistic allocator. As a base case, we have also implemented Chaitin's original register allocator which is based on aggressive coalescing.

Figure 7 depicts the ratio of the number of copies eliminated by each heuristic to that of copies eliminated by the base case allocator (i.e., 100% means those copies eliminated by the Chaitin's allocator). The graph shows that conservative, iterated, aggressive, and optimistic coalescing remove an average of 62.9%, 76.0%, 96.0%, and 95.6% of copies removed by Chaitin's, respectively. The graph indicates that optimistic coalescing removes almost the same number of copies as aggressive coalescing. Obviously, the difference of 0.4% is due to the undo of coalescing by live range splitting.

One thing to note from the graph is that aggressive coalescing on top of Briggs' allocator removes an average of 4% less copies than on top of Chaitin's allocator. This is due to the fact that Chaitin's allocator generates more spills which make otherwise uncoalescible copies coalescible. For example, consider a copy `x=y` which is not coalescible in the first round of both Chaitin's and Briggs' allocators because the live ranges of `x` and `y` interfere (e.g., there is a definition of `y` right after the copy). If Chaitin's allocator spills `x` while Briggs' does not, Chaitin's might be able to coalesce the copy in its second round because `x` is split into smaller pieces, making `x` and `y` in `x=y` not interfere any more. That is, Chaitin's allocator eliminates more copies than Briggs' at the expense of more spills.

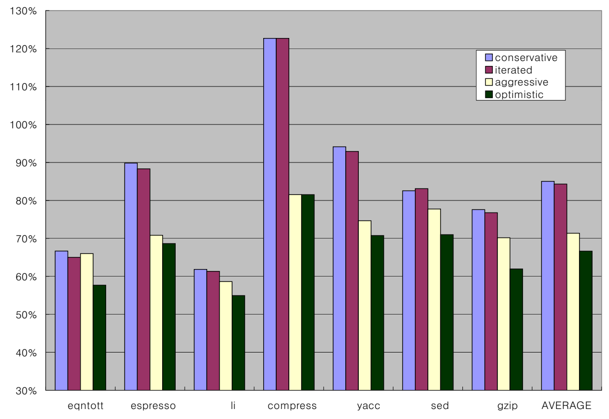


Figure 8. Ratio of spill instructions compared to those of the base-case allocator.

Figure 8 shows the ratio of the number of spill instructions generated by each heuristic to that of spill instructions generated by Chaitin's (again, 100% means spill instructions generated by Chaitin's). The graph shows that optimistic coalescing generates the smallest number of spill instructions in all benchmarks. Moreover, aggressive coalescing by itself is also better than conservative coalescing or iterated coalescing[4] (except for `eqntott`). The average difference between aggressive coalescing and optimistic coalescing is 5%, which is due to the advantage in coloring caused by live range splitting.

5.3. Evaluation of Spill Results

The spill results indicate that aggressive coalescing makes the positive impact of coalescing outweigh the negative impact, improving the overall colorability of the graph. In order to confirm this, we have analyzed the experimental results in detail to estimate the positive impact and the negative impact.

First, we estimate the negative impact of coalescing by measuring how many of nodes that have been coalesced in violation of the conservativeness heuristic are indeed actually-spilled to memory. This will evaluate if aggressive coalescing that delays the spill concern later can be justified. To avoid confusion in the following discussion, we use the term **node** for a single primitive node that has been created in the first build phase, and the term **chunk** for a coalesced node composed of several nodes.

[4]In `compress`, conservative coalescing and iterated coalescing generate more spill instructions even than the Chaitin's allocator. We found that the positive impact of coalescing well outweighs even the benefit of optimistic register allocation.

Benchmark	Num. of Nodes in Coalesced Chunks (Num. of Chunks)	Num. of Nodes in Violating Chunks (Num. of Chunks)	Aggressive		Optimistic	
			Num. of Spilled Nodes (Chunks)	Ratio %	Num. of Spilled Nodes (Chunks)	Ratio %
eqntott	1646 (561)	1042 (432)	98 (33)	9.4	53 (38)	5.1
espresso	12731 (4134)	9659 (2987)	911 (352)	9.4	629 (397)	6.5
li	1478 (492)	830 (240)	10 (4)	1.2	4 (3)	0.5
compress	104 (42)	68 (25)	0 (0)	0.0	0 (0)	0.0
yacc	5765 (1846)	3964 (1202)	186 (77)	4.7	121 (84)	3.1
sed	1172 (401)	674 (202)	24 (12)	3.6	12 (12)	1.8
gzip	4366 (1332)	3277 (968)	365 (110)	11.1	187 (128)	5.7
average				5.6		3.2

Table 1. Numbers of nodes in coalesced chunks and violating chunks, and actually-spilled nodes.

Benchmark	Num. of Nodes in Candidate Chunks (Num. of Chunks)	Spilled Nodes		Colored Nodes		Successful Coloring Ratio (%)	Successful Split Ratio (%)
		in Fully Spilled Chunks	in Partially Spilled/Colored Chunks		in Fully Colored Chunks		
eqntott	114 (40)	12 (6)	41 (32)	55 (32)	6 (2)	53.5	85.0
espresso	1065 (418)	275 (121)	354 (276)	387 (276)	49 (21)	40.9	71.1
li	12 (5)	0 (0)	4 (3)	4 (3)	4 (2)	66.7	100.0
compress	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0	0
yacc	217 (87)	48 (19)	73 (65)	85 (65)	11 (3)	44.2	78.2
sed	24 (12)	0 (0)	12 (12)	12 (12)	0 (0)	50.0	100.0
gzip	429 (134)	48 (18)	139 (110)	227 (110)	15 (6)	56.4	86.6
average						52.0	86.8

Table 2. Detailed data of live range splitting.

In Table 1, the first column shows the number of all coalesced chunks that are pushed on the stack in the first-round of optimistic register allocation with aggressive or optimistic coalescing (see Figure 6) and the number of nodes in those chunks. Many of those chunks have been formed in violation of the conservativeness heuristic, and the number of violating chunks and the number of nodes in violating chunks are shown in the next column. Among those nodes in violating chunks, the number of nodes that are actually-spilled to memory by aggressive coalescing and by optimistic coalescing are described with the number of chunks in the third and the fourth columns, respectively.

In aggressive coalescing, only an average of 5.6% of nodes among those that have been coalesced non-conservatively are actually-spilled to memory. This means that for those remaining 94.4% of nodes, conservative coalescing misses the coalescing opportunity even though they can be colored successfully. This spill rate further drops down to 3.2% in optimistic coalescing with the help of live range splitting, which is analyzed in detail in Table 2.

Among those violating chunks in Table 1, the first column in Table 2 shows the number of chunks that have been the candidate of live range splitting and the number of nodes in those chunks[5]. A candidate chunk belongs to one of the following three categories after live range splitting:

- Fully-spilled, meaning that all of its splits are spilled to memory.

- Fully-colored, meaning that all of its splits are colored successfully.

- Partially-spilled/colored, meaning that some splits are spilled while the rest are colored.

Table 2 shows the number of fully-spilled, fully-colored, and partially-spilled/colored chunks, and the number of nodes in those chunks[6]. The ratio of successfully colored nodes to all split candidate nodes is an average of 52.0% while the ratio of successfully split chunks (i.e., at least one of whose splits are colored) to all candidate chunks is an average of 86.8%. This is the reason why optimistic coloring spills less compared to aggressive coalescing.

One thing to note is that the number of actually-spilled chunks by aggressive coalescing in Table 1 is not equal to the number of split candidate chunks (which are, in fact, actual-spill candidate chunks) by optimistic coalescing in Table 2. The reason is that compared to spilling in aggressive coalescing, splitting in optimistic coalescing imposes more constraints on subsequent coloring in the selection phase, making more chunks below the stack be actual-spill candidates.

Above results indicate that the negative impact of coalescing can be alleviated by optimistic allocation,

[5]It should be noted that if the set of these candidate chunks, the set of potentially-spilled chunks, and the set of violating chunks are denoted by S, P, and V, respectively, the following relationship always holds: $S \subset P \subset V$.

[6]It should be noted that the number of spilled nodes and chunks by optimistic coalescing in Table 1 is the sum of those in partially-spilled and fully-spilled chunks in Table 2.

live range splitting, and the positive impact of coalescing. In order to analyze the positive impact of coalescing, we need to measure the number of successfully colored nodes due to the positive impact which would otherwise be spilled to memory. Unfortunately, this number is not straightforward to measure. Instead, we have measured the reduction of the interference edges due to the positive impact, as depicted in Table 3. For each benchmark, the first column shows the number of interference edges right after the first build phase. The rest of the table shows the number of edges removed due to the positive impact for each heuristic, just before the first selection phase. The table shows that aggressive coalescing removes four times as many interference edges as iterated coalescing does on the average.

Benchmark	Num. of Original Edges	Num. of Deleted Edges		
		Conservative	Iterated	Aggressive /Optimistic
eqntott	83523	1674	2506	8598
espresso	744637	7530	16818	103032
li	103457	1554	2311	6524
compress	9228	106	217	866
yacc	326276	4740	8982	40977
sed	71366	1399	3156	6614
gzip	234617	3106	5933	30111

Table 3. Number of reduced interference edges due to positive impact of coalescing.

6. Summary

In this paper, we have proposed a new coalescing heuristic which fully exploits the positive impact of coalescing via aggressive coalescing on top of optimistic allocation. It has never been evaluated separately how aggressive coalescing does compared to conservative coalescing in the context of optimistic allocation, whereas iterated coalescing, the only recent research work on coalescing, has focused only on improving conservative coalescing. Our experimental results indicate that aggressive coalescing is, in actuality, competitive in optimistic allocation. It is also shown that the negative impact of aggressive coalescing can be mitigated by the live range splitting phase which can reverse problematic coalescing.

It is too premature to conclude that the approach of optimistic coalescing is always better than the approach of conservative coalescing because the two approaches are fundamentally different and we have evaluated them in only one context of register allocation. On the other hand, our analysis of the scheduled code does not reveal any special characteristics that would make optimistic coalescing particularly more useful for its register allocation. It is left as a future work to evaluate the heuristic in other contexts, leading to a more general solution for coalescing and register allocation.

References

[1] G. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN Symp. on Compiler Construction*, pages 201–207, 1982.

[2] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM TOPLAS, Vol 16, No.3*, pages 428–455, May 1994.

[3] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM TOPLAS, Vol 12, No. 4*, pages 501–536, Oct 1990.

[4] S.-M. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM TOPLAS, Vol 19, No. 6*, pages 853–898, Nov. 1997.

[5] K. D. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the ACM PLDI '97*, pages 308–319, June 1997.

[6] R. Cytron J. Ferrante B. Rosen M. Wegman and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS, Vol 13, No. 4*, pages 451–490, Oct 1991.

[7] L. George and A. W. Appel. Iterated register coalescing. *ACM TOPLAS, Vol 18, No. 3*, pages 300–324, May 1996.

[8] P. Kolte and M. J. Harrold. Load/store range analysis for global register allocation. In *Proceedings of the ACM PLDI '93*, pages 268–277, June 1993.

[9] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocation. In *Proceedings of the CC '98*, Mar 1998.

[10] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill code minimization via interference region spilling. In *Proceedings of the ACM PLDI '97*, pages 287–295, June 1997.

[11] P. Briggs. Register allocation via graph coloring. PhD thesis, Rice Univ., Apr. 1992.

[12] T. Gross G.Y. Lueh and A. Adl-Tabatabai. Global register allocation based on graph fusion. In *Technical Report CMU-CS-96-106, School of Computer Science, Carnegie Mellon University*, March 1996.

[13] S.-M. Moon, H. Chung, J. Park, S. Shim, and J.-W. Ahn. SPARC-based VLIW Testbed. *IEE Proceedings Computers and Digital Techniques*, May 1998.

[14] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the PLDI '88*, pages 318–328, 1988.

[15] S. Kim, S.-M. Moon, J. Park, and K. Ebcioğlu. Unrolling-based copy coalescing. Tech. Rep. SNU-EE-TR-1997-7, Seoul National Univ., Seoul, Korea, 1997.