Coloring-based Coalescing for Graph Coloring Register Allocation

Rei Odaira, Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, Toshio Nakatani

IBM Research – Tokyo
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan
{ odaira, nakaike, e29253, komatsu, nakatani } @jp.ibm.com

Abstract

Graph coloring register allocation tries to minimize the total cost of spilled live ranges of variables. Live-range splitting and coalescing are often performed before the coloring to further reduce the total cost. Coalescing of split live ranges, called sub-ranges, can decrease the total cost by lowering the interference degrees of their common interference neighbors. However, it can also increase the total cost because the coalesced sub-ranges can become uncolorable. In this paper, we propose coloring-based coalescing, which first performs trial coloring and next coalesces all copy-related sub-ranges that were assigned the same color. The coalesced graph is then colored again with the graph coloring register allocation. The rationale is that coalescing of differently colored sub-ranges could result in spilling because there are some interference neighbors that prevent them from being assigned the same color. Experiments on Java programs show that the combination of live-range splitting and coloring-based coalescing reduces the static spill cost by more than 6% on average, comparing to the baseline coloring without splitting. In contrast, well-known iterated and optimistic coalescing algorithms, when combined with splitting, increase the cost by more than 20%. Coloring-based coalescing improves the execution time by up to 15% and 3% on average, while the existing algorithms improve by up to 12% and 1% on average.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processor – Compilers.

General Terms Algorithms, Performance, Experimentation.

Keywords Register allocation; register coalescing.

1. Introduction

Global register allocation was formalized by Chaitin et al. [7][8] as a vertex coloring problem on an interference graph, where a node represents the live range of a variable, an edge between nodes indicates the interference between the live ranges, and a color corresponds to a physical register. The goal of the graph coloring register allocation is to minimize the total cost of uncolored nodes. The cost of a node is the sum of the execution costs of the uses and definitions in the corresponding live range. Chaitin's coloring algorithm heuristically determines the coloring order for the nodes based on the degree of interference in addition to the spill cost of each node. The larger the degree is, the lower the node is in the coloring order because it restricts the coloring of many interference neighbors.

Since the graph coloring register allocation is a simple formalization, it can only determine whether the entire live range can be assigned to a single register or must be spilled out to memory. This is because the graph does not contain any further details about the live range. When a live-range is spilled, spill-out (store) instructions are inserted after every definition and spill-in (load) instructions before every use [7]. However, it is often the case that the total cost of spill instructions can be further reduced by assigning only some parts of a live range to a register and by assigning different parts of a live range to different registers [5].

For this reason, various live-range splitting approaches have been proposed [1][5][16]. They split live ranges into shorter ranges, which we call *sub-ranges*, before the graph coloring register allocation. The sub-ranges derived from the same live range are connected by copy instructions at splitting points, and thus they are called *copy-related* sub-ranges. These live-range splitting approaches then exploit Chaitin's coloring algorithm, which is expected to determine a good coloring order of the sub-ranges to minimize the total cost of spill instructions.

In fact, it is well known that coalescing of copy-related sub-ranges is necessary between live-range splitting and register allocation to reduce the total spill cost [14][19][20]. This is because coalescing decreases the degree of the common interference neighbors of coalesced sub-ranges. If copy-related sub-ranges X1 and X2 interfere with a sub-range Y, coalescing X1 and X2 will decrease the interference degree of Y by one. The lower the degree is, the more likely the node will be assigned a color. On the other hand, coalescing can also increase the total spill cost because if the coalesced sub-ranges are spilled, we might have to pay as much as the sum of the spill costs of the sub-ranges. Although various coalescing algorithms have been proposed, none of them can effectively reduce the total spill cost. They are either too conservative [6][10] or too aggressive [7][19] in coalescing criteria.

In this paper, we propose a simple but powerful coalescing algorithm called *coloring-based coalescing*. The key idea is to perform trial coloring of the sub-ranges. We coalesce the copy-related sub-ranges that are assigned the same color together, which we call *companion* sub-ranges. After the coalescing, all of

CGO'10 April 24–28, 2010, Toronto, Ontario, Canada.

the colors are cleared, and the actual graph coloring register allocation is performed in the usual manner. The rationale is that the more interference neighbors a group of copy-related sub-ranges share, the more often such sub-ranges are assigned the same color together. This is because the coloring of the sub-ranges is restricted by their common interference neighbors. More importantly, if sub-ranges are assigned different colors, there are some interference neighbors that prevent them from being assigned the same color. That means if we forced them to be coalesced, they would be spilled in the actual coloring. Therefore we should coalesce companion sub-ranges and let non-companion sub-ranges remain split.

The benefits of coloring-based coalescing are twofold:

- It is effective in reducing the total cost of spill instructions. To the best of our knowledge, this is the first work to reveal the combined power of live-range splitting and graph coloring register allocation by coalescing companion sub-ranges.
- It is simple in its design because it can utilize the existing coloring algorithm for register allocation. All that is needed is to perform coloring twice. After the first coloring, we do not generate the actual spills but only coalesce copy-related sub-ranges with the same color.

Here is the structure of the rest of this paper. Section 2 covers graph coloring register allocation and live-range splitting. Section 3 clarifies and exemplifies our target problems. Section 4 describes our coloring-based coalescing algorithm. Section 5 explains our implementation and gives experimental results. Section 6 discusses related research in register allocation. Section 7 concludes the paper.

2. Background

In this section, we first describe the algorithm of Briggs-style graph coloring register allocation [6]. Next we show how live-range splitting transforms a program.

2.1 Graph Coloring Register Allocation

Chaitin et al. [7][8] invented the original algorithm for graph coloring register allocation that was later found to be too pessimistic. Briggs et al. [6] improved the algorithm by using the optimistic approach shown in Figure 1(a).

1. **Renumber**: Each disjoint live range is given a unique name.
2. **Build**: An interference graph is built.
3. **Spill costs**: A spill cost Cost(lr) is calculated for each live range lr. The cost is the total number of accesses to the live range weighted by instruction cost and by loop nesting level. If an execution profile is available, the cost is the total execution frequency of accesses.
4. **Simplify**: Nodes are removed from the interference graph and pushed into a coloring stack. For nodes whose interference degrees are larger than or equal to the number of physical registers, the one with the minimum value of Cost(lr) / Degree(lr) is removed first. Here, Degree(lr) is the interference degree of live range lr. Thus the smaller the degree of a node is, the more likely it is that the node is assigned a color.
5. **Select**: A node is repeatedly popped from the coloring stack and assigned a color if possible. If no color is available for the node, it is marked for spilling.
6. **Spill code**: If any node is marked for spilling, spill instructions are inserted. Because a spill instruction requires a temporary register to hold a spilled-in or spilled-out value, the whole register allocation process has to be iterated after clearing all of the assigned colors.

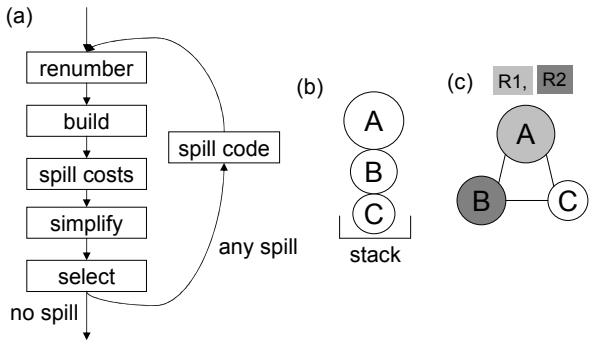

Figure 1. (a) Briggs-style graph coloring allocation. (b) Results of the simplify phase for the example in Figure 2(a). (c) Results of the select phase on the interference graph.

(a)

```
1:  A = ...
2:  B = ...
3:  while (true) {
4:      C = ...
5:      ...= A + ...
6:
7:      ...= C + ...
8:
9:      if (...) {
10:         A = ...
11:         B = C + ...
12:     } else {
13:
14:
15:         if (B) {
16:             A = ...
17:             ...= B + ...
18:
19:         } else {
20:
21:             if (A > 0) break
22:         }
23:     }
24:     A = A + ...
25:     B = B + ...
26: }
```

(b)

```
1:  A1 = ...
2:  B1 = ...
3:  while (true) {
4:      C1 = ...
5:      ...  = A1 + ...
6:      A2 = A1
7:      ...  = C1 + ...
8:      C2 = C1
9:      if (...) {
10:         A3 = ...
11:         B2 = C2 + ...
12:     } else {
13:         B2 = B1
14:         B3 = B1
15:         if (B1) {
16:             A3 = ...
17:             ...  = B3 + ...
18:             B2 = B3
19:         } else {
20:             A3 = A2
21:             if (A2 > 0) break
22:         }
23:     }
24:     A1 = A3 + ...
25:     B1 = B2 + ...
26: }
```

Figure 2. Before (a) and after (b) live-range splitting based on load-range analysis.

In Figure 2(a), suppose there are two physical registers, R1 and R2, available for three variables A, B, and C. Figure 1(c) shows the interference graph. Since these variables interfere with one another and C has the minimum number of accesses in the loop, we spill C. Figure 1(b) is the coloring stack after the simplify phase, where C is pushed first, and Figure 1(c) is the results of the select phase, where A is assigned R1 first and then B R2. As a result, we generate one spill-out and two spill-in instructions at statements 4, 7, and 11. Actually, an additional variable needs to be spilled to allocate registers to the temporary variables used by the spill instructions.

2.2 Live-range Splitting

Live-range splitting takes place before graph coloring register allocation with the expectation that graph coloring can choose the best parts of live ranges to assign to registers. Researchers have proposed various splitting algorithms such as the load-range analysis [16] and the forward-and-reverse-SSA approach [5]. In this paper, we do not assume any particular type of live-range splitting, although the performance of each coalescing algorithm described later can depend on the type used.

After splitting the live range, each part is given a unique name and copy instructions are inserted at the splitting points. We call each short part of a live range a *sub-range*. A pair of sub-ranges is *copy-related* if they are connected by a copy instruction. A copy instruction can be eliminated after register allocation if both its source and target are assigned to the same register. In this paper, we mainly focus on reducing spill instructions rather than copy instructions, because spills have larger execution costs than copies in modern CPU architectures. However, it is still important not to increase copy instructions too much while reducing spill instructions.

Figure 2(b) shows an example of live-range splitting based on load-range analysis [16], which splits the live ranges of A, B, and C at every use point. Each sub-range originates from a use and extends up to the most recent accesses to the variable. For example, B2 corresponds to the use of B at statement 25 and extends up to the uses at 17 and 15 and the definition at 11. Thus new copying definitions of B2 are inserted around 15 and 17, and the target of 11 is modified to B2. The sub-ranges A1 and A2, A2 and A3, B1 and B2, B2 and B3, B3 and B1, and C1 and C2 are all copy related.

3. Problems

In this section, we show that coalescing is necessary to reduce the total spill cost after live-range splitting. We then explain why existing coalescing algorithms are not able to decrease the cost effectively.

3.1 Biased Coloring

When used with live-range splitting, the graph coloring register allocation normally uses a biased coloring [6], where copy-related sub-ranges are assigned to the same color as often as possible. Figure 3(a) shows the interference graph of the program in Figure 2(b) and the results of biased coloring using two registers, R1 and R2. Thick lines connect copy-related nodes in the graph. All of the sub-ranges in this example are considered to have the same spill cost because each load range corresponds to a single use. In this example, A1 and A2 are spilled, so that one spill-out at statement 24 and two spill-ins at 5 and 20 are generated in the loop. The copy at statement 6 is removed. Note that no spill-in instruction is generated for the use at statement 21 because it can refer to a register that was just loaded at its previous statements. In summary, we do not benefit from live-range splitting in this example because without splitting we generate the same number of spills in the loop as described in Section 2.1.

However, if we somehow spilled B1, we could assign A1 and A2 to R2 and would generate only one spill-out at statement 25 and one spill-in at 13 in the loop. Note that the uses at 14 and 15 can refer to a register loaded by the spill-in at 13. Unfortunately, biased coloring does not offer any heuristic to choose B1 for spilling. A2 is pushed onto the coloring stack before B1 because it has a larger degree of five. After pushing A2 and B3, when the triangle of A1, B1, and C1 remains, the coloring algorithm does not have any reason to prefer B1 to A1 for spilling.

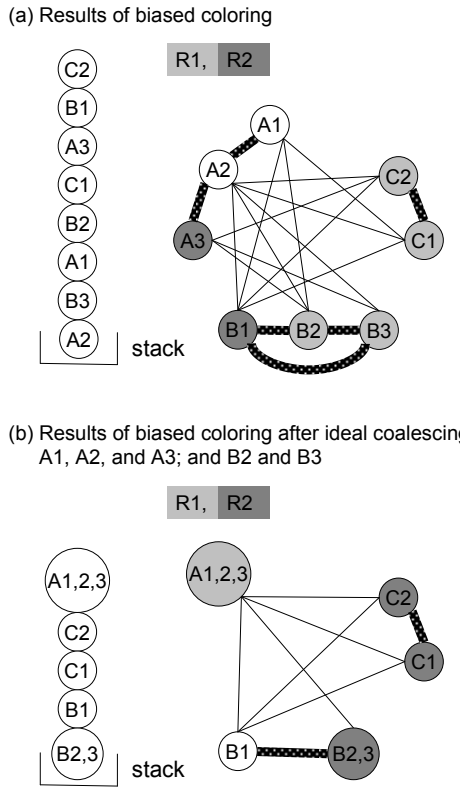

Figure 3. (a) Results of biased coloring on the interference graph of the example program in Figure 2(b). (b) Results of biased coloring after ideal coalescing. Note that neither iterated nor optimistic coalescing can generate these results.

3.2 Coalescing of Sub-ranges

Coalescing of sub-ranges can reduce the total spill cost because it can lower the degree of the common interference neighbors of the coalesced sub-ranges. Coalescing was proposed to remove copies, but is nowadays important for reducing spills, which take longer execution cycles.

Apparently, we should coalesce copy-related sub-ranges that have many common interference neighbors. In Figure 3(a), the neighbors of A1 and A3 are totally included in those of A2, and the same is true for B2 and B3. If we coalesce A1, A2, and A3, and B2 and B3, we will have the graph in Figure 3(b). The number of spilled nodes is successfully reduced from two to one.

However, further coalescing will increase the total spill cost. In Figure 3(a), B1 and B2 share two interference neighbors, and the same is true for C1 and C2. If we decide to coalesce those copy-related sub-ranges that have more than one common neighbor, we will have a graph that is the same as the one before splitting. Therefore we cannot reduce the total spill cost. In general, it is not clear to what extent we should coalesce sub-ranges if we solely use the information about the sharing of interference neighbors.

3.3 Iterated and Optimistic Coalescing

George et al. proposed iterated coalescing [10], which coalesces copy-related nodes during the simplify phase only when the coalesced nodes will not become uncolorable. It uses two criteria for coalescing: the Briggs test for non-precolored nodes and the George test for precolored nodes. A node is precolored before the graph coloring when it must be assigned to a certain physical register because of an architectural reason, for example because it is used as an argument to a function call. Under the Briggs test, two copy-related nodes can be coalesced if the node after the coalescing has fewer significant interference neighbors than the number of physical registers. A node is called significant when the number of its neighbors is equal to or greater than the number of physical registers. In fact, this is the "full" Briggs test named by Hailperin [14], which is more powerful than the one described in [10]. In the example, after pushing A2, B3, and A1 into the stack, iterated coalescing merges C1 and C2 by using the Briggs test. Iterated coalescing cannot coalesce other nodes because the resultant node might become uncolorable. Thus it does not produce better results than simple biased coloring. In general, the criteria in iterated coalescing are too conservative.

Park et al. proposed optimistic coalescing [19], which first merges all of the coalescable nodes. It then splits a merged node back into separate nodes when the node is found to be uncolorable in the select phase. In the example, optimistic coalescing first reverts the interference graph back to the original one before splitting, and then splits C again into C1 and C2 during the select phase. Unfortunately, at that point, it is too late to color any of them, because each of them interferes with nodes A and B, which are already colored. This consequence is due to the aggressiveness of optimistic coalescing.

4. Coloring-based Coalescing

So far we have shown that existing coalescing approaches can be either too conservative or too aggressive to reduce spill instructions. Our experimental results in Section 5 confirm this fact. We need new coalescing criteria with which we can coalesce as many copy-related nodes that share interference neighbors as possible without making too many coalesced nodes uncolorable.

4.1 Basic Concept

We propose a simple but powerful coalescing algorithm called coloring-based coalescing. Coloring-based coalescing first attempts to color a graph using the same coloring algorithm as the register allocation, and then coalesces all copy-related nodes that are assigned the same color. We call such nodes *companion* nodes. During the trial coloring, it uses more colors than the number of physical registers to color all of the nodes.

Coloring-based coalescing is based on our assumption that the results of the coloring reflect the essential structure of the graph: the more interference neighbors a group of nodes share, the more often such nodes are assigned the same color in the trial coloring. This is because the common neighbors impose a similar set of restrictions on the coloring of the nodes. Thus we can infer the sharing of interference neighbors from the coloring. More importantly, coalescing of differently colored nodes could result in spilling because there are some interference neighbors that prevent them from being assigned the same color. Therefore, we should not coalesce non-companion nodes.

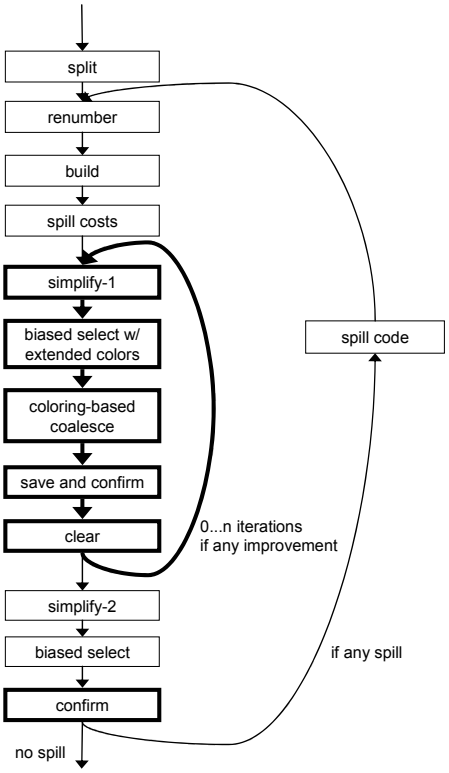

Figure 4. Coloring-based coalescing. Boxes with thick borders are our extension to the biased coloring.

4.2 Algorithm

Figure 4 shows the five main phases of the coloring-based coalescing algorithm. Those boxes with thick borders are our extension to the biased coloring:

1. **Simplify-1**: This is exactly the same as the simplify phase in the graph coloring register allocation. Simplify-2 in Figure 4 also does the same thing. We can also use iterated coalescing in this phase.
2. **Biased select with extended colors**: We extend the select phase by using more colors than the number of physical registers. Otherwise, we would not assign any color to spilled nodes, so that we could not coalesce them. The purpose of this trial coloring is not to allocate registers but to analyze the structure of the interference graph. Therefore, we need to apply the coloring algorithm to all the nodes. The algorithm is as follows, where the colors corresponding to physical registers are called real colors, while the other colors are called extended colors. Steps 1 to 4 are the same as the existing biased coloring, while Steps 5 to 7 are our extension:
   1. If the coloring stack is empty, then stop.
   2. Pop a node X from the stack.
   3. If there is a real color that is not allocated to any of its interference neighbors and is allocated to any of its copy-related nodes, then assign X to that real color and go back to Step 1.
   4. If there is a real color that is not allocated to any of its interference neighbors, then assign X to that real color and go back to Step 1.

5. If there is an extended color that is not allocated to any of its interference neighbors and is allocated to any of its copy-related nodes, then assign X to that extended color and go back to Step 1.
6. If there is an extended color that is not allocated to any of its interference neighbors, then assign X to that extended color and go back to Step 1.
7. Otherwise, introduce a new extended color, assign X to that extended color, and go back to Step 1.

It is important not to introduce extended colors at the beginning but to add them on demand. We first try to allocate real colors whenever possible because the trial coloring should resemble the actual graph coloring register allocation. We also try to reuse existing extended colors as much as possible. Otherwise, too many spilled nodes could be assigned to the same extended color.

3. **Coloring-based coalesce**: We coalesce all of the copy-related nodes that are assigned the same color, real or extended.
4. **Save and confirm**: With more iterations of coalescing, we can expect that the total spill cost will be further reduced. However, too many iterations might promote too much coalescing and increase the number of spilled nodes. Therefore, we save the results of this iteration and compare them with that of the previous iteration. Only when we confirm an improvement in the total spill cost, we go back to the simplify-1 phase. Otherwise we restore the results of the previous iteration and exit the loop. Practically, we should limit the maximum number of iterations because of the increase in the compilation time. We also confirm the improvement at the end of the graph coloring register allocation as shown in the bottom of Figure 4. Although this method does not guarantee to find the best number of iterations, it allows coloring-based coalescing to always succeed in coloring a graph that is colorable by the original graph coloring algorithm.
5. **Clear**: Before exiting or continuing the loop, we clear the colors of the nodes because the colors themselves do not matter on the changed interference graph.

Coloring-based coalescing does not distinguish a precolored node from a non-precolored one. We coalesce non-precolored and precolored nodes when they are copy-related and when the former is assigned the same color as the latter by the trial coloring.

The algorithm of coloring-based coalescing is simple. Although we illustrate the first trial coloring and the next graph coloring register allocation as separate phases, in fact we only need to iterate coloring. Except for the last iteration, after the coloring we coalesce the copy-related nodes that are assigned the same color. In the last iteration, we generate the spill instructions for any nodes that are assigned to extended colors. Our implementation of coloring-based coalescing is based on a Briggs-style allocator, but the rationale behind coloring-based coalescing is effective for other graph coloring register allocators whose heuristics are based on interference degrees.

The spatial complexity of our coalescing is the same as that of iterated coalescing. It requires a list of the coalescable pairs of copy-related nodes. The temporal complexity is the sum of the time to scan the list for coalescing and the complexity of graph coloring register allocation. The overhead of using extended colors is negligible in practice.

4.3 Example

Figure 5(a) shows the results of the trial coloring using real colors R1 and R2 plus an extended color R101. This is the same as the result in Figure 3 except for the use of the extended color. Based

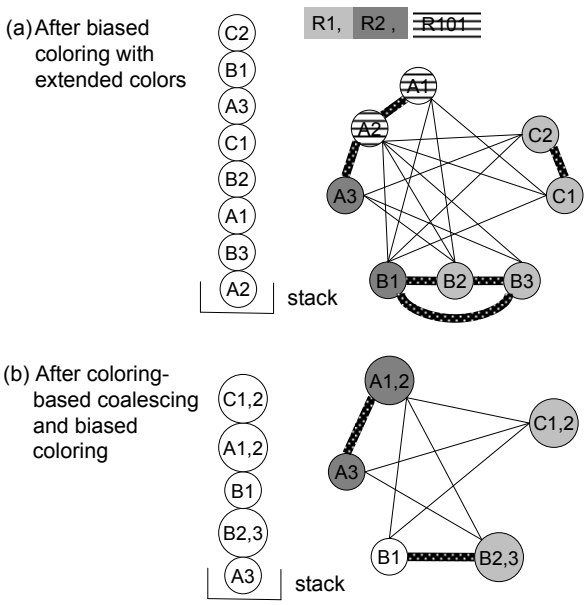

Figure 5. Results of coloring-based coalescing

on this trial coloring, we coalesce three groups of copy-related nodes that are assigned the same color: A1 and A2; B2 and B3; and C1 and C2. Thus they are companion nodes. The new nodes have the sum of the spill costs of the coalesced nodes because each node before coalescing corresponds to a single use and thus each new node represents two uses in the original program shown in Figure 2(a). Note that in general the cost of a new node may not be the sum of the costs of the coalesced nodes because it depends on the splitting algorithm used. Figure 5(b) shows the final results of the register allocation. A3 or B1 is randomly chosen first to be pushed onto the stack because they have the same spill cost and the same interference degree. Whichever is chosen first, we reach the same results, which are the optimal coloring of the graph.

This example shows how coloring-based coalescing reveals sub-ranges to be coalesced. The key to obtaining a good coloring in this example is to coalesce A1 and A2 and not to coalesce B1 with B2 or B3. Iterated coalescing cannot do anything here because it must guarantee the colorability of the coalesced nodes by taking account of their degrees. In contrast, coloring-based coalescing discovers that A1 and A2 share common interference neighbors and no other neighbors prevent them from being assigned the same color. For B1, B2, and B3, although they share a common neighbor A2, other neighbors force them to be assigned different colors. That is, B1 interferes with C2, C2 with A3, and A3 with B2 and B3. This interference chain means that coalescing of B1 with B2 or B3 creates a new triangle, which is not two-colorable. Coloring-based coalescing can sense the danger of the coalescing from the fact that they are assigned different colors.

In summary, coloring-based coalescing is more powerful than iterated coalescing because it can perform coalescing regardless of the interference degrees. When compared with optimistic coalescing, coloring-based coalescing is more effective because it can optimize the shape of an interference graph before the simplify-2 phase. It is often too late to optimize coloring in the final select phase.

Program	Number of frequently executed methods	Without live-range splitting			With forward-and-reverse-SSA live-range splitting
		Total number of nodes in frequently executed methods	Total number of spilled nodes with 8 registers (percentage of spill)	Total number of spilled nodes with 16 registers (percentage of spill)	Total number of nodes in frequently executed methods
_201_compress	4	603	131 (21.7%)	36 (6.0%)	1345
_202_jess	8	1608	223 (13.9%)	96 (6.0%)	5711
_209_db	5	309	65 (21.0%)	20 (6.5%)	1198
_213_javac	10	1423	207 (14.6%)	62 (4.4%)	5641
_222_mpegaudio	12	1307	390 (29.8%)	195 (14.9%)	2500
_227_mtrt	13	2224	341 (15.3%)	194 (8.7%)	10878
_228_jack	32	2878	29 (1.0%)	10 (0.3%)	4147
hsqldb	127	7496	784 (10.5%)	205 (2.7%)	21598
luindex	38	2623	465 (17.7%)	137 (5.2%)	7589

Table 1. Characteristics of the benchmark programs

4.4 Conservativeness of Coloring-based Coalescing

We calculate the upper bound of the chromatic number of the coalesced graph to show the worst case for our algorithm.

THEOREM 1. *The chromatic number of the graph G'' that results from coalescing companion nodes of the graph G' that was generated by splitting nodes in a graph G is less than or equal to the chromatic number of G or the number of colors used in the trial coloring of G', whichever is smaller.*

PROOF. We use the fact that splitting never increases the chromatic number. Since we limit coalescing to copy-related nodes, we can reach G'' from G by splitting. Thus the chromatic number of G'' is less than or equal to that of G. Let H be the graph that results from coalescing into one node all of the nodes that are assigned the same color by the trial coloring. The chromatic number of H is less than or equal to the number of colors used in the trial coloring. Since we can reach G'' from H by splitting, the chromatic number of G'' is less than or equal to the number of colors used in the trial coloring. Q.E.D.

The theoretical effectiveness of the algorithm is an open question: on what kind of graphs does the coalescing of companion nodes definitely reduce the total spill cost?

5. Experiments

5.1 Implementation

We implemented coloring-based coalescing in IBM J9/TR 2.4 [11], a Java™ VM with an advanced Just-In-Time (JIT) compiler. Note that our algorithm was not designed specifically for use in a JIT compiler. We used the JIT compiler mainly because it is our compiler infrastructure. In order to get stable results from run to run, we did not use the execution frequency profile of basic blocks in the compiler. For spill cost calculations, we multiplied the spill costs of definitions and uses in a loop by ten. This is common heuristics as in [2][7]. In addition, the cost was set to zero if a definition or a use was on a path that is statically regarded as a rare path, such as backup code for a devirtualized method call. We call the cost calculated in this way a *static cost*.

Our JIT compiler performs aggressive inlining first and then eliminates redundancy by value numbering and partial redundancy elimination. It also unrolls frequently executed loops. Instruction scheduling is performed twice, before and after the register allocation.

We implemented a Briggs-style graph coloring register allocator as our baseline register allocator. It is equipped with biased coloring and iterated coalescing with the full Briggs and George tests [14]. Therefore, our coloring-based coalescing uses iterated coalescing both in the trial coloring and in the actual register allocation. The baseline allocator performs spill coalescing and spill propagation [17] as post optimizations. We also implemented optimistic coalescing including copy graph optimization [19], which has a function similar to spill propagation.

We mainly show the results of using forward-and-reverse-SSA live range splitting [5] because it can effectively split live ranges around a loop. We also implemented splitting at every basic block boundary to present the effectiveness of our algorithm. Our coalescing implementations do not distinguish copies introduced by the splitting from the other copies. Even when live-range splitting is turned off, our baseline register allocator uses iterated coalescing to remove copies that exist in an original program or that are introduced by other compiler optimizations.

5.2 Evaluation

We used all seven programs in SPECjvm98 [21] and two larger programs in the DaCapo [9] benchmarks. We ran the benchmarks on an IBM System z9 2094 [15] with four 64-bit processors and 8 GB of RAM. We used a 1-GB Java heap. The machine has sixteen integer registers and sixteen floating-point registers. We also simulated an eight-register architecture by using only eight integer and eight floating point registers. Since three integer registers are reserved for special purposes, thirteen out of sixteen and five out of eight integer registers can be used for register allocation.

For execution performance, we ran each program (e.g. _201_compress) sequentially twenty times in a Java VM process and chose the run with the shortest execution time. We used the shortest time to exclude JIT compilation time from the execution time. We confirmed that JIT compilation ended early in the sequential runs. We invoked the sequential runs four times for each program and report the averages of the shortest execution times.

We summarize the characteristics about the benchmark programs in Table 1. The second column shows the number of frequently executed methods in each program. In the following results except for execution performance, we show the total statistics for these hot methods alone rather than the entire program, because a Java program executes many non-application methods during initialization. The third column is the total numbers of

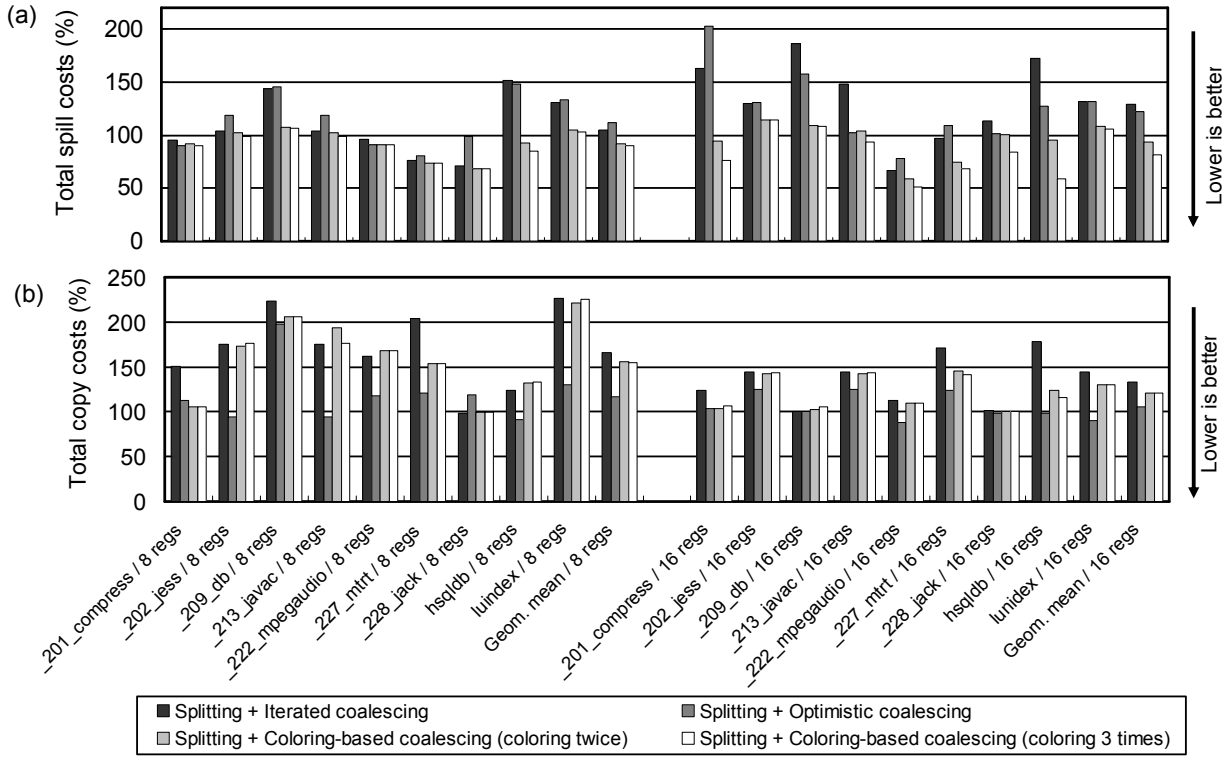

Figure 6. Comparison of (a) total static spill costs and (b) total static copy costs for frequently executed methods in the benchmark programs (100% = Register allocation w/o splitting)

nodes (including precolored ones) when live-range splitting is not used. Note that the spill costs of these nodes significantly differ from one another, depending on where and how many times the corresponding variables appear in the methods. The fourth and fifth columns show the total numbers of spilled nodes when eight and sixteen registers are used, respectively. We also include in parentheses the percentages of the spilled nodes among all of the nodes. The numbers of frequently executed methods for _208_jack, hsqldb, and luindex are larger than for the other programs, but they mostly exhibit low register pressure because they contain few computationally complex methods in their frequently executed paths. The sixth column is the numbers of nodes when the forward-and-reverse-SSA live range splitting is used. The splitting results in 2.9 times more nodes on average than the case without splitting.

In the following results, the baseline is the graph coloring register allocation without live-range splitting. With forward-and-reverse-SSA live-range splitting turned on, we compared iterated coalescing, optimistic coalescing, and our coloring-based coalescing. All the results are normalized to the results of the graph coloring allocation without splitting. For coloring-based coalescing, we experimented with no iteration and the maximum iteration of one for the inner loop of Figure 4. Note that the no iteration and one iteration actually execute coloring twice (one for coalescing and the other for actual register allocation) and 3 times at maximum, respectively. Thus we name them "coloring-based coalescing (coloring twice)" and "coloring-based coalescing (coloring 3 times)."

We first show the total static costs of spill instructions. Figure 6(a) is the results for the eight-register case on the left hand side and for sixteen registers. Note that a spill instruction in a loop is weighted by ten in the spill cost calculation. The combination of live-range splitting and our coloring-based coalescing (coloring twice) successfully reduced the total static spill costs on average by 8% with eight registers and by 6% with sixteen registers, compared with the baseline register allocation. Coloring-based coalescing (coloring 3 times) reduced the costs on average by 10% and by 18%, respectively. The large reductions are mostly due to removing spill instructions from loops. Iterated coalescing and optimistic coalescing did not reduce the total static spill costs on average, when combined with the live-range splitting. In several methods, they increased the cost by more than 50%. In contrast, coloring-based coalescing increased the cost by up to 14%. Contrary to our expectations, optimistic coalescing did not perform well in combination with live-range splitting, because it can assign only one color to sub-ranges of a spilled live range.

When we increased the maximum times for coloring, we observed consistently better results for the total static spill costs as shown in Figure 6(a). However, we found that even coloring 3 times was sometimes more than needed for some methods. These results indicate that we do not need to iterate coloring-based coalescing many times to get reasonable improvements.

Even when we used basic-block-based splitting instead of SSA-based splitting, coloring-based coalescing (coloring twice) achieved 31% reduction in the total static spill costs on sixteen registers compared with iterated coalescing, and 22% with opti-

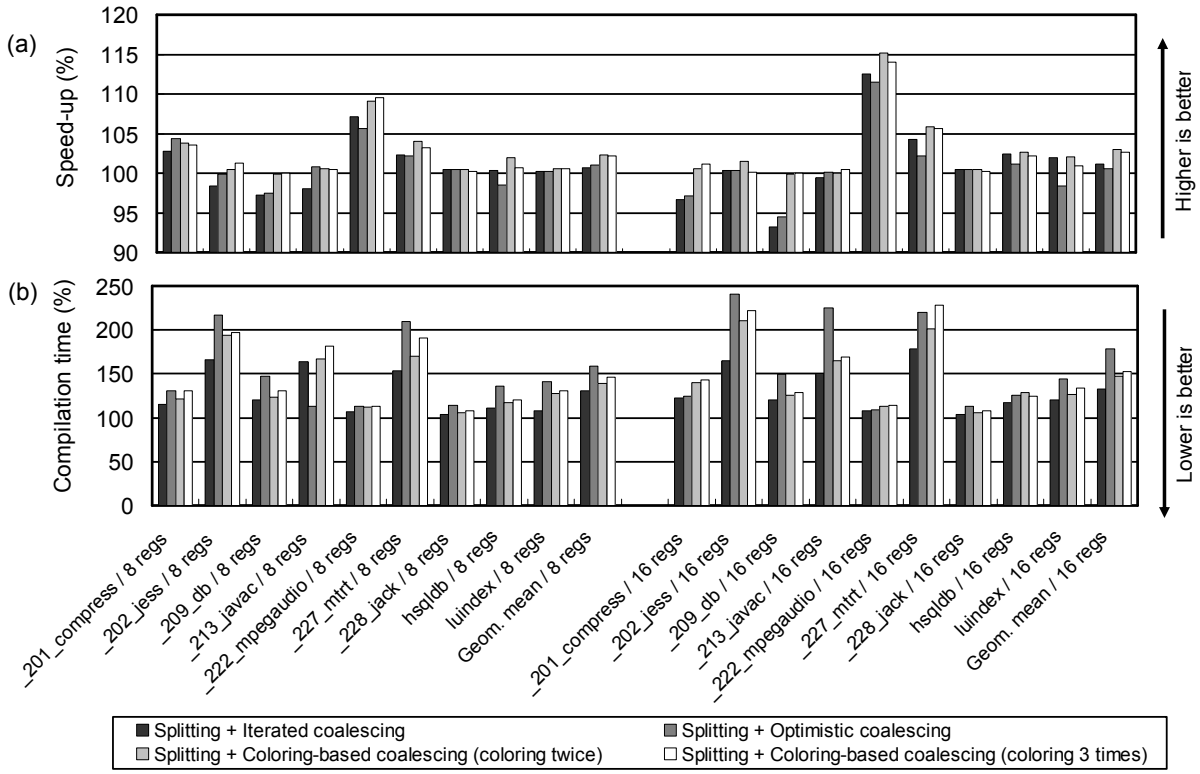

Figure 7. Comparison of (a) execution time speed-up for the benchmark programs (not including compilation time) and (b) compilation time for frequently executed methods in the benchmark programs. (100% = Register allocation w/o splitting)

mistic coalescing. Thus coloring-based coalescing is effective for different splitting algorithms.

When comparing the eight-register and sixteen-register results in Figure 6(a), we saw smaller increases or decreases in the relative costs with eight registers. This is because when only eight registers are available, there are many inevitable spill instructions that none of the coalescing algorithms can remove.

Figure 6(b) shows the total static costs of copy instructions. The copies include those from splitting, from other compiler optimizations, and from the original program source code. The combination of live-range splitting and coloring-based coalescing increased the copy cost by 56% and 21% on average with eight and sixteen registers, respectively. Coloring-based coalescing generally worked better than iterated coalescing. Optimistic coalescing is an effective technique to reduce copies, but it cannot effectively reduce the spills as shown in Figure 6(a). In general, more copies remained unremoved with eight registers than with sixteen registers because the registers were more restricted.

Figure 7(a) is the execution performance results. Coloring-based coalescing (coloring twice) achieved 2.5% speed-up on average and 9% speed-up at maximum, compared with the eight-register baseline. With sixteen registers, the speed-up was 3% on average and 15% at maximum. The speed-up in _222_mpegaudio resulted from the reduction in spill instructions in the innermost loop of hot methods. For most of these programs, it delivered better performance than iterated and optimistic coalescing. The average speed-up over the best of the existing algorithms was 1.5% with eight registers and 2% with sixteen registers. The existing algorithms degraded performance by more than 2% in several programs, while coloring-based coalescing showed at least the same performance as the baseline without splitting. Coloring-based coalescing (coloring 3 times) did not always perform better than coloring twice, because we did not use execution profile to estimate spill costs. The fact that the increase or decrease in the static costs in Figure 6(a) was not always reflected in the execution time in Figure 7(a) indicates that we need more sophisticated methods to predict runtime costs. Overall, coloring-based coalescing performs well with both eight and sixteen registers.

Figure 7(b) shows the compilation time for each program, including time spent in live-range splitting, coalescing, and register allocation. The combination of live-range splitting and coloring-based coalescing (coloring twice) increased the compilation time by 39% and 47% over the baseline when using eight and sixteen registers, respectively. To evaluate the overhead of the iterations of graph coloring, one should note the difference between coloring-based coalescing and iterated coalescing, which is 8% to 14% on average. In _202_jess and _227_mtrt, there were a few methods for which the number of nodes exploded due to splitting, causing the compilation time to increase regardless of the coalescing algorithms. The compilation time also depended on how many times the coloring algorithm iterated the loop in Figure 1(a) or the outer loop in Figure 4. For example in hsqldb with 16 registers, coloring-based coalescing with coloring 3 times took slightly less compilation time than with coloring twice, because it iterated the outer loop a fewer number of times in some methods.

6. Related Work

Hack et al. [13] proved that the colorability of a program in SSA form can be determined by the maximum number of simultaneously live variables. They also presented a quadratic-time optimal coloring algorithm for a SSA-program. This means that SSA-based live-range splitting leads to optimal register allocation for a colorable SSA-program. However, if the program is not colorable, they only provided greedy heuristics for spilling. Among the 249 hot methods we described in Section 5, 125 methods have the number of simultaneously live variables larger than five, which is the number of available physical registers in the eight-register configuration. Even in the sixteen-register configuration, 63 methods including most of the SPECjvm98 methods (except for _228_jack) have more than thirteen simultaneously live variables. Thus coloring-based coalescing can help reduce spills in these methods. Hack et al. [12] also proposed a safe coalescing algorithm for their SSA-based register allocation. The purpose of their algorithm is not to reduce spills but to reduce register-to-register copies at splitting points.

Vegdahl [20] proposed node merging to improve graph coloring register allocation. The technique maintains a pair-score for each pair of nodes, which is the ratio of the number of common interference neighbors to the total number of neighbors in the smaller-degree node. It coalesces high-pair-score nodes when there remain only significant-degree nodes in the simplify phase. It has a similarity to coloring-based coalescing in that it focuses on a node pair that has many common interference neighbors. However, it often results in overly aggressive coalescing because it does not reflect the colorability of the coalesced nodes. In contrast, coloring-based coalescing takes advantage of the trial coloring and takes account of factors that can prevent the pair from receiving the same color.

Nakaike et al. [18] proposed two-phase register allocation to be used after live-range splitting. The first phase is to spill subranges in high-register-pressure regions and also to coalesce subranges on hot paths. The second phase is the graph coloring register allocation. It is similar to coloring-based coalescing in that it performs pre-allocation and coalescing before the actual register allocation. However, it heavily relies on an execution profile, while coloring-based coalescing does not.

Appel et al. [1] used integer linear programming to find out optimal splitting points. Their approach spills variables to make no more than K variables simultaneously live at any point, where K is the number of physical registers. It does not necessarily compute a globally optimal solution, despite the fact that it requires an ILP solver in a compiler. Our coloring-based coalescing provides a reasonable reduction in spill costs by taking advantage of the existing coloring algorithm in a register allocator.

Bouchez et al. pointed out [4] that the power of iterated coalescing is limited because they can only coalesce pairs of nodes at one time. It is often the case that a better interference graph cannot be reached without coalescing a group of nodes at once. Coloring-based coalescing is effective because it can coalesce all of the copy-related nodes with the same color at one step. Bouchez et al. also proposed [3] advanced conservative and optimistic coalescing algorithms. Their purpose is to reduce register-to-register copies when coloring a greedy-k-colorable graph. Specifically, they proposed chordal-based incremental coalescing, which can merge a group of nodes at once along a "path" of non-interfering nodes in an interval graph. However, it is not clear whether or not those algorithms help reduce spills in a general non-k-colorable graph.

7. Conclusions

In this paper, we proposed a new coalescing algorithm called coloring-based coalescing. It first performs a trial coloring with an extended number of colors and then coalesces all of the copy-related nodes that are assigned the same color, which we call companion nodes. After the coalescing, all colors are cleared, and the actual graph coloring register allocation is performed. Companion nodes are worth coalescing because they share common interference neighbors and do not have other neighbors that would make the coalesced nodes uncolorable. To the best of our knowledge, this is the first coalescing alogrithm to strengthen the combined power of live-range splitting and graph coloring register allocation by focusing on companion nodes. It is simple because it utilizes the existing graph coloring function of a register allocator. Experiments on Java programs using sixteen registers showed that the combination of live-range splitting and coloring-based coalescing reduced the total static cost of spill instructions by more than 6% on average, comparing to the baseline coloring allocation without splitting. On the other hand, well-known iterated and optimistic coalescing algorithms increased the total static cost by more than 20%, when combined with splitting. Coloring-based coalescing improved the execution time by up to 15% and 3% on average, whereas iterated and optimistic coalescing improved by up to 12% and 1% on average. We also conducted experiments using only eight registers. Coloring-based coalescing provided up to 9% and on average 2.5% speed-up, which were larger than 7% maximum and 1% average speed-up by the existing coalescing algorithms.

Acknowledgments

We thank the members of the Systems group in IBM Research - Tokyo, who gave us valuable suggestions. We are also grateful to anonymous reviewers for providing us with helpful comments.

References

[1] Appel, A. W. and George, L. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 243-253, June 2001.

[2] Bernstein, D., Golumbic, M. C., Mansour, Y., Pinter, R. Y., Goldin, D. Q., Krawczyk, H., and Nahshon, I. Spill code minimization techniques for optimizing compliers, In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 258-263, July 1989

[3] Bouchez, F., Darte, A., and Rastello, F. Advanced conservative and optimistic register coalescing. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 147-156, 2008.

[4] Bouchez, F., Darte, A., and Rastello, F. On the complexity of register coalescing. In *Proceedings of the International Symposium on Code Generation and Optimization* 2007, pages 102-114, March 2007.

[5] Briggs, P. Register Allocation via Graph Coloring. PhD thesis, Rice University, April 1992.

[6] Briggs, P., Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pages 428- 455, May 1994.

[7] Chaitin, G. J. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction*, pages 201-207, SIGPLAN Notices Vol. 17, No. 6, pages 98-105, June 1982.

[8] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. Register allocation via coloring. *Computer Languages*, Vol. 6, No. 1, pages 47-57, January 1981.

[9] DaCapo Benchmarks, http://dacapobench.org/.

[10] George, L. and Appel, A. W. Iterated register coalescing. *ACM Transactions on Programming Languages and System*s, Vol. 18, No. 3, pages 300-324, May 1996.

[11] Grcevski, N., Kilstra, A., Stoodley, K., Stoodley, M., and Sundaresan, V. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 151-162, May, 2004.

[12] Hack, S. and Goos, G. Copy coalescing by graph recoloring. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 227-237, June 2008.

[13] Hack, S., Grund, D., and Goos, G. Register allocation for programs in SSA-form. In *International Conference on Compiler Construction (CC'06)*, Vol. 3923 of LNCS, pages 247-262, Springer Verlag, 2006.

[14] Hailperin, M. Comparing conservative coalescing criteria. *ACM Transactions on Programming Languages and Systems*, Vol. 27, No. 3, pages 571-582, May 2005.

[15] IBM System z9. IBM Journal of Research and Development Vol. 51, Number 1/2, 2007.

[16] Kolte, P. and Harrold, M. J. Load/store range analysis for global register allocation. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 268-277, June 1993.

[17] Leung, A. and George, L. A new MLRISC register allocator. *Standard ML of New Jersey compiler implementation notes*, 1998.

[18] Nakaike, T., Inagaki, T., Komatsu, H., and Nakatani, T. Profile-based global live-range splitting. In *Proceedings the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 216-227, June 2006.

[19] Park, J. and Moon, S. Optimistic Register Coalescing. *ACM Transactions on Programming Languages and Systems*, Vol. 26, No. 4, pages 735-765, July 2004.

[20] Vegdahl, S. R. Using node merging to enhance graph coloring. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages150-154, May 1999.

[21] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks, http://www.spec.org/osg/jvm98/.