Color Flipping

Felipe L. Silva(✉), Marcelo F. Luna, and Wesley Attrot

State University of Londrina, Londrina, Brazil
{felipe.lds.88,marcelofernandesdeluna}@gmail.com
wesley@uel.br

Abstract. Spill code minimization is an important problem in register allocation because it affects the quality of the code produced by the compiler and program performance. This work presents a new technique to reduce spill code, called color flipping. Differently of other techniques, color flipping prevents all load/store instructions insertion when avoiding spill. Nevertheless, color flipping can be used in combination with other spill minimization techniques to achieve an overall better result. To evaluate the impact of using color flipping, experiments with a set of interference graphs and with the benchmark SPEC CPU2006, showed over 12 % of spill reduction.

Keywords: Spill minimization · Register allocation · Color flipping

1 Introduction

Register allocation [10,16,18,23] is one of the most important compiler optimizations. It directly affects the quality of the code produced. The goal of register allocation is to keep as many as possible temporary values created by a program in machine registers. The problem in register allocation occurs when the finite number of available machine registers can not fit the unbounded temporary values. When this occurs some values must be kept in memory, which decreases the speed of the generated code. To keep the temporaries in memory, load/store instructions are inserted into the code; this process is called spill code generation.

The most widely used algorithm to perform register allocation is graph coloring [7,10,15]. In this approach, the compiler builds an interference graph G, where each node represents a live range and edges connecting two live ranges l_i and l_j symbolizes an interference and means that l_i and l_j will be live at the same time in the future and should not occupy the same register. The problem then is to find a proper K-coloring for G, such that no two adjacent nodes receive the same color. By representing the colors as machine registers we can perform register allocation with a coloring algorithm.

An ideal register allocator should produce the minimum amount of spill code possible to avoid unnecessary memory accesses, and therefore slowdown the executable code. However, introducing the minimum spill code as possible is an NP-complete problem.

© Springer International Publishing Switzerland 2015
A. Pardo and S.D. Swierstra (Eds.): SBLP 2015, LNCS 9325, pp. 81–95, 2015.
DOI: 10.1007/978-3-319-24012-1_7

Several efforts have been made to find efficient techniques to reduce the impact of spills in the code. In 1989 Bernstein *et al.* [5] improved the Chaitin's allocator with new heuristics to select the spill node known as *best-of-three*. In the same year, Briggs *et al.* [6] developed a stronger coloring heuristic, called *optimistic coloring*. In 1992 Briggs *et al.* [8] also extended the *rematerialization* notion of Chaitin by dealing with multi-valued live ranges. The rematerialization recomputes constant values when it is cheaper than to store and reload it. In 1997 Bergner *et al.* [4] developed a new minimization technique, known as *interference region spilling* that was able to spill partially a live range. Later in 1998, Cooper and Simpson [13] developed a new technique to globally split live ranges similar to that developed by Bergner *et al.* [4] known as *live range splitting*. In 2003, Govindarajan *et al.* [17] developed a heuristic to reduce the numbers of registers used by instruction sequencing, called *Minimum Register Instruction Sequence (MRIS)*. In the same year, Koseki *et al.* [20] developed a new technique for partial spilling called *spill code motion*. In 2005, Gao and Shi [14] created a method, named *merge* that allows two interfered nodes in the interference graph occupy the same machine register. Finally in 2013, Barany and Krall [3] developed a *global code motion* to order basic blocks with the aim of reduce overlaps among live ranges.

The majority of previous spill code minimization research efforts have been focused on studying spilling heuristics to select the live range with the smallest spill cost [5,9] and finer spilling/splitting mechanisms to reduce the number of load/store instructions inserted [4,7,13]. Unlike these techniques we introduce a technique called *color flipping* which focuses on the coloring stage of graph coloring algorithm, where if *color flipping* succeeds no load/store instructions are inserted because a register is assigned for the entire live range. The main idea is to attempt to recolor [19] the interference graph, such that a used color becomes available for spill node.

2 Color Flipping

To demonstrate how *color flipping* works, we present a simple example where the spill is successfully avoided. The interference graph and its corresponding node costs are shown in Fig. 1. In this example, we will assume that we have 3 colors available, that is, $K = 3$.

After coloring the interference graph, we are left with the 3-colored sub-graph shown in Fig. 2(a) and the uncolored live range F. Normally we would spill the live range F. However, observing this graph we notice that F has three neighbors with unique colors: A : *green*, B : *blue* and E : *red*. So, if we change the color of any of these nodes, then we will make a color available for F. By extending this idea to one more level of the interference graph, i.e., searching for nodes with unique color in the neighborhood of A, B and E, we can start to flip colors. Analyzing the neighbors of A, we find that A has no neighbor with unique color, so we proceed our analysis to B. We observe that B has one neighbor with unique color, that is, A : *green*, C : *green* and D : *red*. As B is the only *blue*

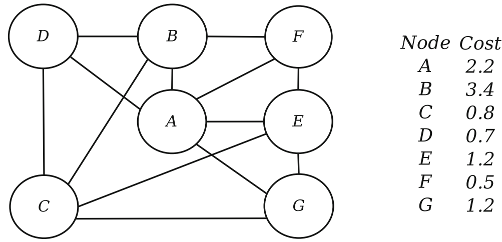
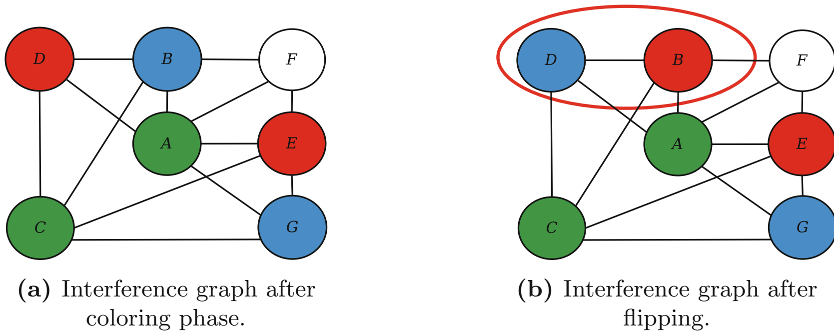


Fig. 1. Interference graph and its spill costs.



(a) Interference graph after coloring phase.

(b) Interference graph after flipping.

Fig. 2. Flipping colors in the graph (Color figure online).

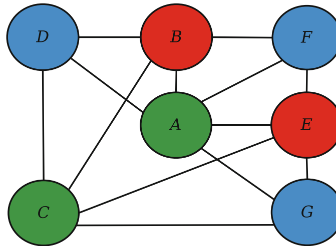
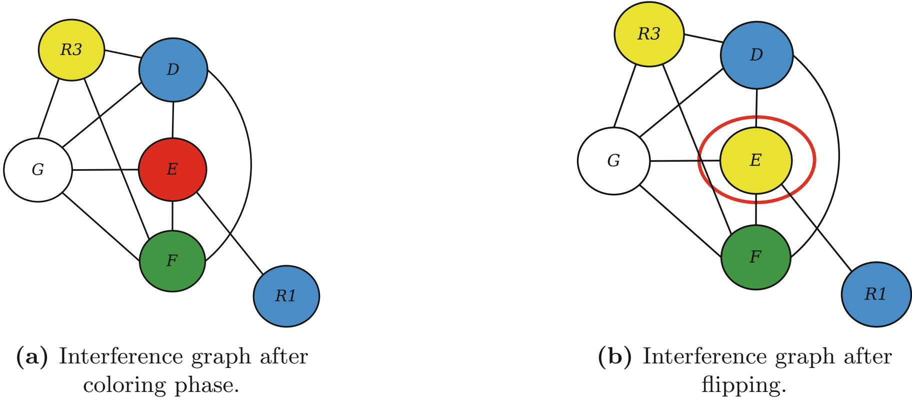


Fig. 3. Final result after applying color flipping in Fig. 2(a) and continuing register allocation (Color figure online).

node connected to D, so, it's possible to flip B and D colors, as seen in Fig. 2(b). Now we are free to color F with *blue* as shown in Fig. 3.

In Fig. 2(a) we flipped colors between two neighboring nodes. But it is also possible to recolor a node if it has another color available. In the next example we present a situation where recolor a node in this way makes a color available for the spill node. The interference graph after the coloring phase and after color flipping is shown in Fig. 4. We assume that $K = 4$. There are two physical registers $R1$ and $R3$ already in the graph. Node F interferes with $R1$; nodes



(a) Interference graph after coloring phase.

(b) Interference graph after flipping.

Fig. 4. Flipping colors in the graph (Color figure online).

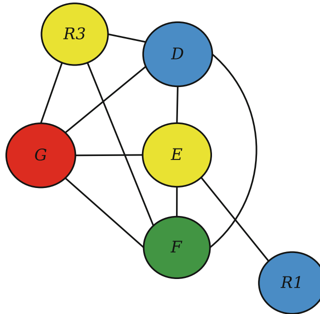


Fig. 5. Final result after applying color flipping in Fig. 4(a) and continuing register allocation (Color figure online).

D, F and G interfere with $R3$. There is no color available for live range G. By observing this graph, we notice that E has another color available, because it can be recolored with *yellow*. Recoloring E in this way, makes *red* available for G. The Fig. 5 shows the result after applying color flipping in the graph.

The main advantage of color flipping over other spill minimization techniques is that when avoiding a live range spill, no load/store instructions are inserted. The color flipping avoids completely the spill, not only partially.

3 Color Flipping Algorithm

To implement color flipping we added an additional stage after the coloring phase. This stage attempts to assign a register for each spilled live range. If color flipping succeeds the live range is removed from the spill list and added to the colored nodes list, otherwise no modification is made on the interference

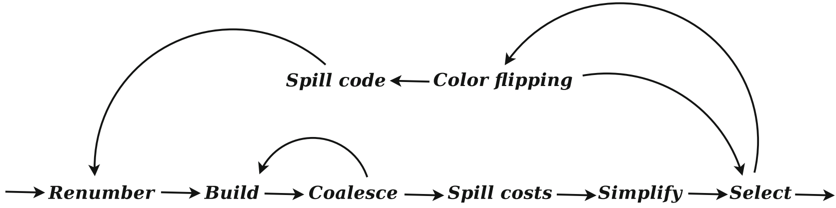


Fig. 6. Color flipping added to Briggs' allocator.

graph and the live range is spilled. The Fig. 6 shows the Briggs' allocator [7] with color flipping stage added.

Given an interference graph G, and a spill node $s \in G$ the color flipping algorithm tries to recolor G such that a valid register R is made available for s. To do so we divided color flipping into two modules: `FindFlippingCandidates` and `TryFlipping`.

The aim of the first module is to find a set of *flpping candidates*, i.e., nodes that may have their colors flipped. It begins analyzing each neighbor n_i of the spill nodes to determine if n_i satisfies three constraints called **flipping restrictions**. A list - `flippingCandidates` - containing the neighbors that meet all flipping restrictions is created.

Once the first module has finished the algorithm starts `TryFlipping`. In this module each flipping candidate $f_i \in$ `flippingCandidates` is analyzed to determine if f_i satisfies one of two **flipping conditions**. In positive case f_i is recolored, such that, a color is made available to the spill node and the algorithm stops. Otherwise the *color flipping* algorithm calls `FindFlippingCandidates` but with I and f_i (not s) as input. This process is repeated until there is no more flipping candidates, that is, `flippingCandidates` $= \emptyset$. We can stop `TryFlipping` before setting a max level of recursion - `maxLevel` - such that *color flipping* stops trying to find new flipping candidate when `maxLevel` is reached. Figure 7 shows a simple flowchart of `TryFlipping`.

The flipping restrictions and the flipping conditions are constraints imposed to a node n_i to guarantee that is safe to flip n_i color. By safe, we mean that all constraints of the interference graph after *color flipping* are preserved. When n_i satisfies all flipping restrictions, then n_i is a *potential* flip node. The next step is to analyze n_i to determine if n_i is an *actual* flipping node, that is, determine if n_i satisfies one flipping condition. In order to understand how the *color flipping* algorithm works, one needs a deeper understanding of the criteria used in flipping restrictions and those used in flipping conditions.

Flipping Restrictions: The `FindFlippingCandidates` module is responsible for finding nodes that satisfy the three flipping restrictions. The input is a node in the interference graph, which we call the target node T, and the output is a list of nodes that meets all three restrictions, which we call `flippingCandidates`.

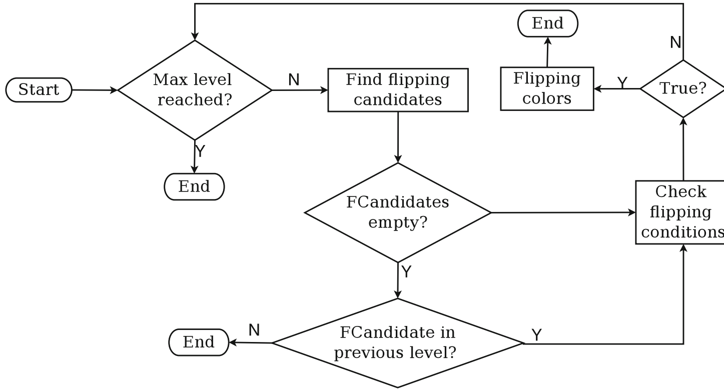


Fig. 7. Flowchart of `TryFlipping` module.

- *First flipping restriction*: this restriction must ensure that the flipping candidate has a unique color among the neighbors of the target node. In Fig. 8(a), T contains three neighbors of the same color. Therefore X, Y and Z do not satisfy the first flipping restriction. In Fig. 8(b) Z satisfies the first flipping restriction. With this restriction we guarantee that if a flipping candidate change its color, then T is free to receive its old color. For our example, in Fig. 8(b) if Z is recolored, we are free to color T with *green*.
- *Second flipping restriction*: this restriction ensures that the flipping candidate is colored with a proper register R_i for T. By proper register we mean that R_i does not interferes with T. In the sub-graph of Fig. 9 the node Z is the unique among the neighbors of the target node T colored with *blue*. However, $R1$ interferes with T, which makes Z to violate the second restriction. If we remove $R1$ from the interference graph, then Z satisfies the second flipping restriction.
- *Third flipping restriction*: we say that `FindFlippingCandidates` is on the first level of an interference graph if T is a spill node. If T has flipping candidates, then each one of them may be target nodes, if so we say that we are at level > 1 of the interference graph. Once `FindFlippingCandidates` begins to operate at a level > 1 of the interference graph, the third flipping restriction is triggered. Otherwise this restriction is always satisfied. Consider the graph in Fig. 10, when we begin searching for flipping candidates of T, we find that W satisfies the first and the second flipping restrictions. As we are in the first level, it's unnecessary to check the third flipping restriction, so W is a flipping candidate of T. The algorithm proceeds to determine the flipping candidates of W and finds that Z satisfies the first and second the flipping restrictions. But Z is neighbor of T violating the third flipping restriction.

So the aim of the third flipping restriction is ensure that a flipping candidate does not interfere with a target node of the previous flipping candidate. In the example of Fig. 10, it must ensure that the flipping candidates of the

Algorithm 1. Finds flipping candidates

```
1:  procedure FindFlipCandidates(T)
2:      for all i ∈ T.Adjs do
3:          if i.color ∈ T.PreColored then
4:              continue
5:          for all j ∈ T.Adjs − {i} do
6:              if !(i.color = j.color) then
7:                  continue
8:              if !(T.ancestor ∉ i.Adjs) then
9:                  continue
10:             i.ancestor ← T
11:             flippingCandidates.insert(i)
12:     return flippingCandidates
```

target node W do not interfere with T. If we remove the interference between Z and T, then Z becomes a flipping candidate of W.

The Algorithm 1 shows the implementation of `FindFlippingCandidates`. The line 2 checks if i satisfies the second flipping restriction, line 6 checks if i satisfies the first flipping restriction, finally line 8 checks if i satisfies the third flipping restriction. If i meets all flipping restrictions, then it's added to the list of flipping candidates on line 11.

(a) First restriction unsatisfied.

(b) First restriction satisfied.

Fig. 8. First flipping restriction example (Color figure online).

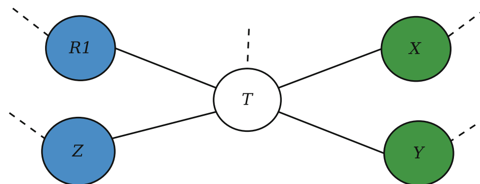


Fig. 9. Second flipping restriction unsatisfied (Color figure online).

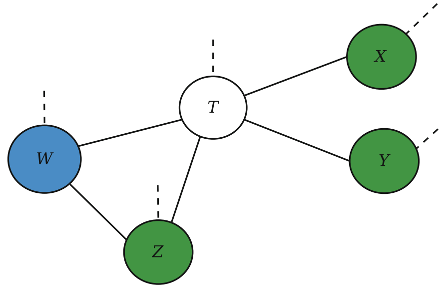


Fig. 10. Third flipping restriction unsatisfied.

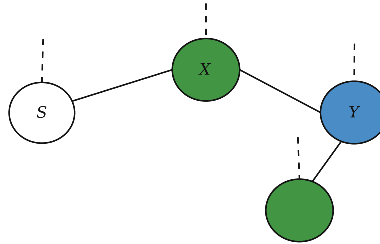


Fig. 11. An interference graph where the second flipping condition fails (Color figure online).

Flipping Conditions: The `TryFlipping` module is responsible for finding nodes that satisfy one of two flipping conditions. The input is a spill node in the interference graph and the desired level of recursion. The output is a valid color for the spill node if color flipping succeeds or -1 if color flipping fails.

- *First flipping condition*: The first flipping condition deals with abusive using of colors that pre-colored nodes may lead in the interference graph. The Fig. 4(a) shows an example of interference graph, which satisfies the first flipping condition. After the coloring phase, it is found that G is a spill node. So we triggered the color flipping algorithm and we found that D, E and F are marked nodes of G. Since E can also be colored with *yellow*, the color *red* is made available for G.
- *Second flipping condition*: The second flipping condition operates at least in three nodes. So it is only triggered from a recursion level > 1. For example, consider the interference graph fragment shown in Fig. 11, if S is a spill node, X a flipping candidate of S, and Y flipping candidate of X, the second flipping condition must ensure that Y has no neighbor with the same color of X. In Fig. 11 flipping the colors of X and Y will not be possible because Y has a neighbor colored with *green*.

An implementation of `TryFlipping` is shown in Algorithm 2. The line 2 checks if the max level of recursion was reached and stops the algorithm in

Algorithm 2. Tries to flipping some nodes colors

```
 1: procedure TryFlipping(upNode, level)
 2:     if level = 0 then
 3:         return -1
 4:     FlipCandidates ← FindFlipCandidates(upNode)
 5:     if FlipCandidates.size() = 0 then
 6:         return -1
 7:     for all i ∈ FlipCandidates do
 8:         if i.allowed.size() > 0 then
 9:             flipColor ← i.color
10:             i.color ← i.allowed.next()
11:             return flipColor
12:         else if upNode.color ≠ −1 then
13:             IAdjs ← AdjList(i) − upNode
14:             if upNode.color ∉ IAdjs.colors then
15:                 flipColor ← i.color
16:                 i.color ← upNode.color
17:                 return flipColor
18:         else if level > 0 then
19:             downFlipColor ← TryFlippingColor(i, level − 1)
20:             if downflipColor > −1 then
21:                 upFlipColor ← i.color
22:                 i.color ← downFlipColor
23:                 return upFlipColor
24:     return -1
```

positive case. The line 4 calls the module `FindFlippingCandidates` and stores its results in `FlipCandidates`. Lines 7–23 loop through each element of the list `FlipCandidates`, to determine if one of them satisfies one of the flipping conditions. Lines 8–11 check the first flipping condition and lines 12–17 check the second flipping condition.

Complexity: The most costing operation in color flipping algorithm is the computation of the first restriction. Given an interference graph G with n nodes, the first restriction needs $(n-1)(n-2)$ comparisons in the worst case, i.e., the cost is $O(n^2)$. Where $(n-1)$ is the number of nodes in G less the spill node and $(n-2)$ is the number of nodes in G less the spill node and the node that is under evaluation in the first restriction. On the other hand, in the best case it's not necessary to compute the first restriction, because the algorithm stops in the second restriction analysis. The number of comparisons to calculate the second flipping restriction is bounded by the number of registers in the target machine. If we represent the number of registers as c, the second restriction needs $c(n-1)$ comparisons to be computed, which gives to color flipping a cost of $\Omega(n)$ in the best case. Based in some of our experimental analysis of color flipping execution, we noticed that the second restriction occurs with considerable frequency, which makes color flipping cost similar to the best case.

4 Experimental Results and Discussion

There are many reasonable ways to measure the quality of a good register allocator - compile time, space requirements, produced executable code efficiency. The main objective of color flipping is to improve code efficiency of allocators that use graph coloring approach. Although an additional cost of space and time is introduced when the color flipping is added to the framework of these allocators, the trend is not to cause severe damage in the performance, since it operates on very limited portions of the graph. This section presents a series of comparisons to measure the impact on the quality of the code when the color flipping is added to Briggs' allocator [7].

To evaluate the efficiency of color flipping two main experiments have been made. The first one takes a set of $27,921$ interference graphs made available by Appel and George [2] to measure how many live range spills were possible to avoid using the color flipping technique. The second experiment, implements the Briggs' allocator with color flipping stage added in LLVM framework [21]. Several comparisons were made between the existing allocators of LLVM. The tests were performed in a Core i5 machine, with 8 GB of RAM in Ubuntu 14.04 64 bits.

4.1 Appel and George Graph Experiments

The set of graphs available by Appel and George [2] were generated from the self-compilation of SML/JN (Standard ML of New Jersey) [1] - a compiler for the language Standard ML '97 - to test new allocation techniques for graph coloring, without relying on any specific framework.

The samples assume that $K = 21$ or $K = 29$, and also provide information about moves between nodes in each graph, which allows the use of coalescing in the allocation process. However, no spill cost information is provided, nor the code that represents the interference graph. This limits the tests in two ways. First when spill occurs, we can not know which variable will be spilled. To work around this problem we assumed that all nodes in the interference graph have $cost = 1$, therefore the node with higher degree is always chosen to spill. The second limitation is that we can not reconstruct the interference graph when spill occurs because there is no code information to make live analysis. In this way, the experiment only computes the effect of color flipping in the first round of the graph coloring algorithm if any spill occurs.

In order to test the efficiency of color flipping a Briggs' allocator without coalescing and where is possible enable color flipping was implemented without any framework dependence. The tests were performed assuming $K = 4, 8, 12, 16, 21/29$. The recursion level of the color flipping was set to 2, we try a recursion level > 2, but there was no significant improvement in the results - less than 0.5 %. The results are shown in Table 1. We observed that as the number of available register grows, the color flipping avoids more spills, this is due to the fact that more flipping opportunities become possible when there are more possibilities of coloring. However, even with $K = 4$ the reduction in

Table 1. Number of live range spills avoided for the Appel and George 27,921 interference graph samples.

K	Briggs - Total spills	Color flipping - Spills avoided	Reduction (%)
4	159,308	6,996	4.37
8	31,417	2,174	6.92
12	10,170	853	8.39
16	3,931	498	12.67
21/29	1,265	146	11.54

the number of live range spills is considerable. Another important observation is that our measurements in Table 1 are in terms of live range spills avoided, not in terms of load/store instructions reduction. As for each live range spilled some load/store instructions are inserted, if we were able to perform our measurements with Appel and George graph samples in terms of load/store reduction, an even better result would be obtained.

4.2 LLVM Experiments

To evaluate the quality of the code produced, the benchmark SPEC CPU2006 was compiled for architectures x86_64 and ARM-Cortex9. A comparison was made with the three main allocators of LLVM: `basic`, `greedy` and `pbqp`. It is difficult to talk about allocators `basic` and *greedy* because there is no official documentation about them. The best material found was an informal mail list between the author of both allocators and the LLVM community [22]. Based on this discussion and code itself, we can infer that both are hybrids allocators, using ordered intervals as the *extended linear scan* [24] but using allocation mechanisms similar of those used in the graph coloring. The `basic` uses a priority queue to separate unrestricted live ranges ($degree < k$) from restricted live ranges ($degree \geq k$) which is similar to the algorithm proposed by Chow and Hennessy [11,12]. The `greedy` is an extension of `basic`, which uses a form of iterative coalescing similar to George and Appel [15] with split on demand. This is the default allocator of LLVM. The `pbqp` allocator is based on quadratic problem solving implemented by Hames Scholz [18].

The results of SPEC CPU2006 benchmark compilation are shown in Tables 2 (x86_64) and 3 (ARM-Cortex9). Unlike the experiments performed in Sect. 4.1, in LLVM experiments the measurements are in terms of spill instructions (load/store). We notice that our allocator produced code with similar quality to LLVM allocators. In some cases much less spill code was inserted, e.g., `403.gcc`, `400.perlbench`. In `403.gcc` for X86_64 (Table 2) was inserted $6,356$ spills with *color flipping*, while all LLVM allocators inserted > 7300 spills. In `400.perlbench` for ARM-CortexA9 (Table 3) was inserted $2,643$ spills, while all LLVM allocators inserted > 3200 spills. We also notice that one of the best performances of *color flipping* was on the `gcc` benchmark. This may be due to the nature of the interfer-

ence graph of a compiler, since the samples of graphs of Appel and George, where we achieve better results, also represented a compiler. Another important observation is that the *color flipping* was more effective in ARM-Cortex9 architecture,

Table 2. Amount of spill code inserted by each benchmark of SPEC CPU 2006 for x86_64 architecture using Briggs', Color Flipping and LLVM's allocators.

Benchmark	Briggs	Color flipping	Reduction (%)	Greedy	Basic	PBQP
400.perlbench	2,957	2,943	0.47	3,789	3,568	3,192
401.bzip2	323	318	1.55	531	329	309
403.gcc	6,422	6,356	1.03	7,352	7,527	7,396
429.mcf	21	21	-	17	20	22
433.milc	663	663	-	612	693	677
444.namd	4,813	4,802	0.23	5,055	5,087	4,731
445.gobmk	2,230	2,227	0.13	2,365	2,325	2,230
450.soplex	1,255	1,255	-	1,127	1,310	1,261
456.hmmer	1,389	1,389	-	1,205	1,424	1,388
458.sjeng	196	196	-	236	217	196
464.h264	2,908	2,897	0.38	3,068	3,014	2,867
470.lbm	89	89	-	41	89	91
471.omnetpp	737	737	-	583	759	724
473.astar	190	190	-	176	197	189

Table 3. Amount of spill code inserted by each benchmark of SPEC CPU 2006 for ARM-CortexA9 architecture using Briggs', Color Flipping and LLVM's allocators.

Benchmark	Briggs	Color flipping	Reduction (%)	Greedy	Basic	PBQP
400.perlbench	2,684	2,643	1.53	3,337	3,271	3,260
401.bzip2	571	556	2.63	739	573	539
403.gcc	6,661	6,536	1.88	7,589	7,605	7,694
429.mcf	30	30	-	31	36	31
433.milc	466	466	-	491	462	486
444.namd	3,655	3,652	0.08	4,926	3,759	3,569
445.gobmk	2,000	1,985	0.75	2,311	2,216	2,148
450.soplex	772	766	0.78	902	840	829
456.hmmer	723	721	0.28	855	755	783
458.sjeng	415	413	0.48	492	464	422
464.h264ref	3,799	3,779	0.53	3,984	3,981	3,818
470.lbm	28	28	-	22	32	28
471.omnetpp	192	191	0.51	239	196	236
473.astar	230	230	-	216	236	226

this suggests that *color flipping* may has a better performance in an environment with more *alias* [25]- while ARM has 289 register units, the X86_64 architecture has 241 register units. Finally we observe that *color flipping* always produced $\leq$ spills when compared to Briggs' allocator.

Based on the experiment of Sect. 4.1 we expected a greater spill reduction. There are two main causes for the results have been affected negatively. The first one is the register class issue. Most of modern architectures are irregular. This means that each live range can only be assigned to a specific set of registers. For example, a variable `int` can not be allocated to a class of registers of type `float`. The graph coloring algorithms are too abstract and do not deal with these issues in their original design. Modern research has sought to make this strategy generic enough to deal with these modern problems [25]. Unfortunately, the tests in Sect. 4.1 do not simulated this behavior. The second one is the spill cost issue. The tests with Appel and George graphs always spills the live range with greater degree, which differs from the spill heuristic used in real programs. This may causes unpredictable results.

5 Conclusion

In this work we presented a new spill code minimization technique called *color flipping*. Rather than try to partially spill a live range, the *color flipping* tries to recolor the interference graph, such that, a color is made available to the live range spilled. If *color flipping* succeeds no load/store instructions are inserted, that is, a machine register is assigned to the entire live range. Otherwise, the graph coloring algorithm proceeds normally with no change in the coloring of the interference graph. Another important advantage of using *color flipping* is that it can combined with other spill minimization techniques easily, which can improve the overall result of the final code.

Our experiments with the samples of Appel and George shown over 12 % of live range spills reduction, suggesting that *color flipping* is an effective technique to avoid spill code. However, in the experiments with the LLVM framework the performance of *color flipping* was not as effective: in most benchmarks there was a reduction $< 1\%$ of spill code.

In further tasks, we should investigate the causes of such performance. We notice that the second restriction occurred much more often in the LLVM experiment, then we will study ways to work around this restriction to achieve better results.

References

1. Appel, A.W.: Standard ml of New Jersey (1996). http://www.smlnj.org/. Accessed 18 Nov 2014
2. Appel, A.W., George, L.: Sample graph coloring problems (1996). https://www.cs.princeton.edu/appel/graphdata/. Accessed 18 Nov 2014

3. Barany, G., Krall, A.: Optimal and heuristic global code motion for minimal spilling. In: Jhala, R., De Bosschere, K. (eds.) Compiler Construction. LNCS, vol. 7791, pp. 21–40. Springer, Heidelberg (2013)
4. Bergner, P., Dahl, P., Engebretsen, D., O'Keefe, M.: Spill code minimization via interference region spilling. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI 1997, pp. 287–295. ACM, New York (1997). http://doi.acm.org/10.1145/258915.258941
5. Bernstein, D., Golumbic, M., Mansour, Y., Pinter, R., Goldin, D., Krawczyk, H., Nahshon, I.: Spill code minimization techniques for optimizing compliers. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. PLDI 1989, pp. 258–263. ACM, New York (1989). http://doi.acm.org/10.1145/73141.74841
6. Briggs, P., Cooper, K.D., Kennedy, K., Torczon, L.: Coloring heuristics for register allocation. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation. PLDI 1989, pp. 275–284. ACM, New York (1989). http://doi.acm.org/10.1145/73141.74843
7. Briggs, P.: Register allocation via graph coloring. Ph.D. thesis, Rice University (1992)
8. Briggs, P., Cooper, K.D., Torczon, L.: Rematerialization. In: Feldman, S.I., Wexelblat, R.L. (eds.) PLDI, pp. 311–321. ACM (1992)
9. Chaitin, G.J.: Register allocation & spilling via graph coloring. In: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1982, pp. 98–105. ACM, New York (1982). http://doi.acm.org/10.1145/800230.806984
10. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Comput. Lang. **6**(1), 47–57 (1981)
11. Chow, F.C., Hennessy, J.L.: The priority-based coloring approach to register allocation. ACM Trans. Program. Lang. Syst. **12**(4), 501–536 (1990). http://doi.acm.org/10.1145/88616.88621
12. Chow, F., Hennessy, J.: Register allocation by priority-based coloring. In: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1984, pp. 222–232. ACM, New York (1984). http://doi.acm.org/10.1145/502874.502896
13. Cooper, K.D., Simpson, L.T.: Live range splitting in a graph coloring register allocator. In: Koskimies, K. (ed.) CC 1998. LNCS, vol. 1383, pp. 174–187. Springer, Heidelberg (1998)
14. Gao, L., Shi, C.: An improved approach of register allocation via graph coloring. In: Proceedings of the SPIE, vol. 5683, no. 5, pp. 113–123, May 2005
15. George, L., Appel, A.W.: Iterated register coalescing. ACM Trans. Program. Lang. Syst. **18**(3), 300–324 (1996). http://doi.acm.org/10.1145/229542.229546
16. Goodwin, D.W., Wilken, K.D.: Optimal and near-optimal global register allocations using 0–1 integer programming. Softw. Pract. Exper. **26**(8), 929–965 (1996)
17. Govindarajan, R., Yang, H., Amaral, J.N., Zhang, C., Gao, G.R.: Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures. IEEE Trans. Comput. **52**(1), 4–20 (2003). http://dx.doi.org/10.1109/TC.2003.1159750
18. Hames, L., Scholz, B.: Nearly optimal register allocation with PBQP. In: Lightfoot, D.E., Ren, X.-M. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 346–361. Springer, Heidelberg (2006)
19. Kempe, A.B.: On the geographical problem of the four colours. Am. J. Math. **2**(3), 193–200 (1879)

20. Koseki, A., Komatsu, H., Nakatani, T.: Spill code minimization by spill code motion. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques 0, p. 125 (2003)
21. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004, p. 75. IEEE Computer Society, Washington (2004). http://dl.acm.org/citation.cfm?id=977395.977673
22. Olesen, J.S.: Greedy register allocation in llvm 3.0 (2011). http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-September/043511.html. Accessed 25 Aug 2014
23. Poletto, M., Sarkar, V.: Linear scan register allocation. ACM Trans. Program. Lang. Syst. **21**(5), 895–913 (1999). http://doi.acm.org/10.1145/330249.330250
24. Sarkar, V., Barik, R.: Extended linear scan: an alternate foundation for global register allocation. In: Adsul, B., Odersky, M. (eds.) CC 2007. LNCS, vol. 4420, pp. 141–155. Springer, Heidelberg (2007)
25. Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI 2004, pp. 277–288. ACM, New York (2004). http://doi.acm.org/10.1145/996841.996875