Lecture Notes on
Register Allocation Optimizations

15-411: Compiler Design
Frank Pfenning, Rob Simmons

Lecture 17
October 27, 2015

1 Introduction

In this lecture we'll look way back at the lecture on register allocation, and consider the ways in which register allocation can be optimized to improve program performance. The most important operation we'll consider is *register coalescing*, which gets rid of register-register moves when doing doesn't lead to spilling more temps.

One of the advantages of Pereira and Palsberg's chordal graph coloring algorithm [PP05] is that it lends itself to a register coalescing approach that is independent of the actual register allocation process. In contrast, register allocation algorithms like the one covered in the textbook [App98, Chapter 11] tend to tightly integrate register allocation and register spilling, making both more complicated. Recall that this process has five steps, only four of which were considered in our initial presentation:

1. **Build** the interference graph from the liveness information.
2. **Order** the nodes using maximum cardinality search.
3. **Color** the graph greedily according to the elimination ordering.
4. **Spill** if more colors are needed than registers available.
5. **Coalesce** non-interfering move-related nodes greedily.

2 Register Allocation Heuristics

Pereira and Palsberg's algorithm for register allocation is notable in that it does *not* tell us which registers to spill in step 4, it only tells us *how many* registers we will need to spill.

Pereira and Palsberg suggest two heuristics for deciding which colors should be spilled and which colors should be mapped to registers: (i) spill the least-used color, and (ii) spill the highest color assigned by the greedy algorithm. For programs with loops and nested loops, it may also be significant *where* in the programs the variables or certain colors are used: keeping variables used frequently in inner loops in registers may be crucial for certain programs.

It can also be advantageous to add heuristics to step 2 of Pereira and Palsberg's algorithm, maximum cardinality search. In practice, this algorithm encounters many "ties" where multiple different registers could be chosen as the next register. If the algorithm prefers to break ties by selecting more frequently used temps (or temps used inside of more nested loops), then those temps will be considered earlier by the greedy graph coloring algorithm and potentially assigned lower-numbered colors.

Other options that can be used for heuristically ordering or spilling temps include:

- values that rematerialize easily, i.e., that can be recomputed easily (say with 1 or 2 instructions) from other registers or at least loaded from or recomputed easily from few memory accesses. When rematerializing from memory, the placement of the instruction needs to be scheduled appropriately for cache and pipeline efficiency reasons.
- values that (approximately) will not be used quickly again when following the (likely) control flow, counting loop bodies as "closer" than loop exits.
- values that interfere with many others.

3 Register Coalescing

The most important optimization related to register allocation is eliminating register-to-register moves with *register coalescing*. Algorithms for register coalescing are usually tightly integrated with register allocation. In contrast, Pereira and Palsberg describe a relatively straightforward method that is performed entirely after graph coloring called *greedy coalescing*.

Greedy coalescing is based on two simple observations:

1. If we have a move $u \leftarrow u$, it won't change the meaning of the program if we delete it.
2. If two temps do not have an interference edge between them, then the two *different* temps could both be renamed to be the *same* temp without changing the meaning of the program. (This is simply what it means for two temps to not interfere!)

Therefore, if t and s do not interfere, then we can *always* eliminate the move $t \leftarrow s$ by creating a new temp u, replacing both t and s with u everywhere in the program, and eliminating the move.

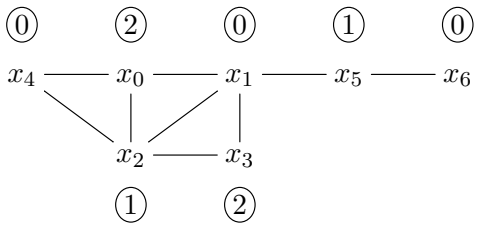
We wouldn't want to do this *before* graph coloring, because it tends to make a chordal graph non-chordal and it also tends to increase the number of colors needed to color the graph. But with a little bit of care, we can coalesce registers t and s for some moves $t \leftarrow s$ *after* we have colored the interference graph but *before* we have rewritten the program to replace temps with registers. The algorithm is as follows:

1. Consider each move between variables $t \leftarrow s$ occurring in the program in turn.
2. If there is an edge between t and s, that is, they interfere, they cannot be coalesced.
3. Otherwise, if there is a color c which is not used in the neighborhoods of t and s, i.e., $c \notin N(t) \cup N(s)$, and which is smaller than the number of available registers, then the variables t and s are coalesced into a single new variable u with color c. Then create edges from u to any vertex in $N(t) \cup N(s)$ and remove t and s from the graph.

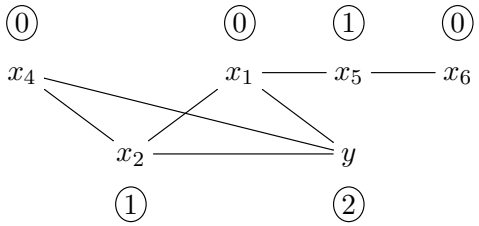
Because of the tested condition, the resulting graph is still K-colored, where K is the number of available registers. Of course, we also need to eventually rewrite the program appropriately by replacing both t and s with u everywhere so that the program remains in correspondence with the graph.

It's important to realize that this is *not* an optimal register coalescing algorithm, in that it won't necessarily remove the maximum number of moves. Optimal register allocation can be done using a reduction to integer linear programming, but this would be too slow.

Let's look at an example, considering the interference graph below, which can be colored with three colors as follows:

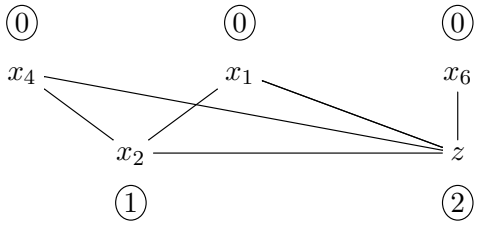


We can always coalesce a move between two registers of the same color. For instance, we can coalesce a move $x_0 \leftarrow x_3$ by creating a new temp y with the same color (2). We would then want to substitute y for x_1 and x_3 everywhere in the program. This new temp will have all the neighbors that x_0 had (x_1, x_2, and x_3) as well as all the neighbors that x_3 had (x_1 and x_2).



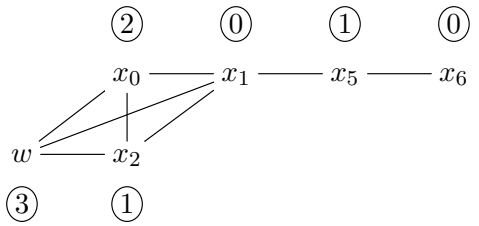
Of course, coalescing two temps that are *already* the same color isn't the interesting case. If that's all we wanted to do, we should have just rewritten the program completely and eliminated obviously redundant self-moves from the register associated with (2) to itself.

As a more interesting example, consider the move $y \leftarrow x_5$ in our rewritten program. (Before rewriting, this would have been either $x_0 \leftarrow x_5$ or $x_3 \leftarrow x_5$.) We can eliminate that move by replacing both y and x_5 with z everywhere in our program. The register y has neighbors with both the color (0) and the color (1), and x_5 has only neighbors with the color (0). We will give z the color (2), the lowest color not in the neighborhoods of y and x_5.



To demonstrate a bit about why doing *optimal* register coalescing is not straightforward, consider what would happen if the original program contained the move $x_3 \leftarrow x_4$. In our new program, this would have been rewritten to $z \leftarrow x_4$, and because there is an interference edge between z and x_4, this move cannot be eliminated.

In the original graph, however, we could have eliminated the move $x_3 \leftarrow x_4$ by coalescing x_3 and x_4 into a new temp w. However, because x_4 in the original graph has neighbors colored (1) and (2), and because x_3 in the original graph has nodes colored (0) and (1), we can only color our new temp w with a color that isn't present in the original graph.



Would we want to perform this step? Almost certainly: even though we're increasing the number of colors needed to color the graph, we have at least 3 caller-save registers available, and it's always worthwhile to use those if possible. In the opposite direction, we might wish to *avoid* coalescing registers if one of the temps had a low color that would be assigned to a temp and the other had a high color that would be assigned to a stack location.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Cha82] Gregory J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 1982. ACM Press.

[Hac07] Sebastian Hack. *Register Allocation for Programs in SSA Form*. PhD thesis, Universität Karlsruhe, October 2007.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In K.Yi, editor, *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05)*, pages 315–329, Tsukuba, Japan, November 2005. Springer LNCS 3780.