Lecture Notes on Optimizations of Register Allocation

15-411: Compiler Design
Frank Pfenning, Rob Simmons, and Jan Hoffmann

Lecture 18
October 25, 2018

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. The idea behind (i) is that colors that are used for fewer nodes will result in the spilling of fewer temps. Strategy (ii) is easier to implement and slightly more efficient. The idea is that (ii) is an approximation of (i). To understand why, recall how greedy graph coloring works: We successful select uncolored nodes and color them with the *lowest color* that is not used by its neighbors. As a result, there is a tendency to use lower colors more often.

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. To take this into account when using strategy (i), you can for instance introduce a weight for each node/temp that depends on the nesting depth of the loops in which the respective temp is used.

It can also be advantageous to add heuristics to step 2 of Pereira and Palsberg's algorithm, maximum cardinality search. This is important if you decide to implement strategy (ii) since nodes that are picked earlier tend to have lower colors. In practice, this algorithm encounters many "ties" where multiple different nodes could be chosen as the next node. 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.

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:

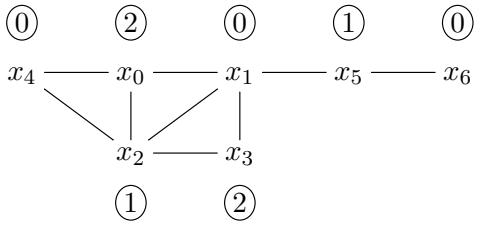
1. Consider each move between variables $t \leftarrow s$ occurring in the program in turn.
2. If there is an edge between t and s, that is, they interfere, they cannot be coalesced.
3. Otherwise, if there is a color $c \leq c_{\max}$ which is not used in the neighborhoods of t and s, i.e., $c \notin N(t) \cup N(s)$, then the variables t and s are coalesced into a single new variable u with color c:
   (a) Create a new node u with color c and create edges from u to all vertices in $N(t) \cup N(s)$.
   (b) Remove t and s from the graph.
   (c) Replace t and s with u in the program.

The color $c_{\max}$ is the maximal color that has been used in the coloring of the original 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.

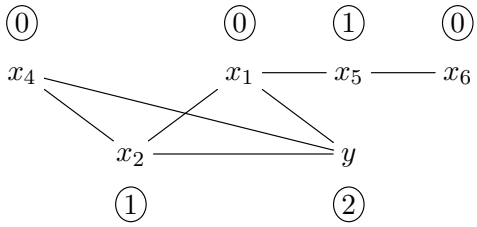
The requirement for coalescing s and t in the algorithm is that there exists a $c \notin N(t) \cup N(s)$ with $c \leq c_{\max}$. However, you should consider some variations of the requirement. It would be for instance still be beneficial to coalesce if c is bigger than $c_{\max}$ in case $c_{\max}$ is small than the number of available registers. If the graph was K colored before coalescing s and t it will be $K + 1$ colored afterwords. However, this does not hamper efficiency since we have $K + 1$ registers. On the other side, if we know already that all colors $c \notin N(t) \cup N(s)$ correspond to temps that will be spilled then coalescing might not make sense; particularly if s and t will be assigned to registers. This situation can arise when using strategy (ii) for spilling the highest colors.

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:

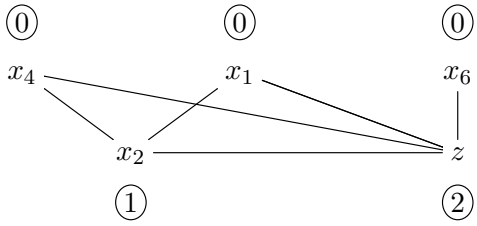


We can always coalesce a move between two registers of the same color. For instance, we can coalesce a move $x_0 \leftarrow x_3$ by creating a new temp y with the same color (2). We would then want to substitute y for x_1 and x_3 everywhere in the program. This new temp will have all the neighbors that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

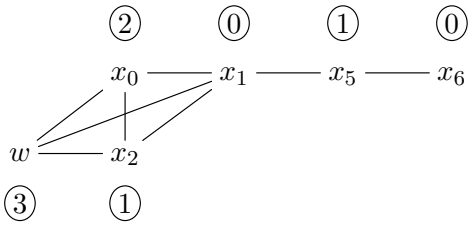
As a more interesting example, consider the move $y \leftarrow x_5$ in our rewritten program. (Before rewriting, this would have been either $x_0 \leftarrow x_5$ or $x_3 \leftarrow x_5$.) We can eliminate that move by replacing both y and x_5 with z everywhere in our program. The register y has neighbors 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.

4 Splitting Live Ranges

Another popular register allocation optimization is *splitting live ranges*. This optimization can be easily integrated in some register allocation algorithms such as *linear scan* but it is not easy to find good heuristics that work for our chordal graph coloring approach.

To split the live range of a temp t we pick a line in the live range of t and insert a new move instruction $t' \leftarrow t$. We then rename all the occurrences of t that are reached by the new definition to t'. Of course, this is only sound if the new move instruction is on a path from all definitions of t that reach the replaced occurrences (where readability is defined as in the reaching definitions analysis). This can for example be ensured by splitting live ranges only at the beginning of a basic block.

In some sense, splitting live ranges is the complementary optimization to register coalescing. The idea of coalescing is to remove move instructions at the cost of making the interference graph more dense. The idea of splitting live ranges is to make the interference graph more sparse at the cost of introducing additional move instructions. The rational is that a move instruction is a low price to pay if we can avoid to spill a temp since spilling could easily introduce a large number of move instructions that involve expansive memory accesses.

How splitting live ranges works is best explained by example. Consider the following code snipped.

			live variables
		...	
		$x \leftarrow y$	y, u, v
		$n \leftarrow u + v$	x, u, v
		$i \leftarrow n$	x, n
l_1	:	if $(i \leq 0)$ then done else l_2	x, i
l_2	:	$i \leftarrow i - 1$	x, i
		$x \leftarrow x * x$	x, i
		goto l_1	x, i
done	:	$a \leftarrow x + 8128$	x
	:	$b \leftarrow a + a$	x, a
		return $x * b$	x, b

The code is a bit contrived and it is not important what it computes. The crucial point is that the temp x is live throughout the whole snipped. As a result, it interferes with almost all other temps in the code: we haven edge in the interference graph between x and u, v, n, i, a, b. Assume that there are additional constraints so that greedy graph coloring leads to the following coloring.

$$\begin{aligned} u &\mapsto 2 \\ v &\mapsto 0 \\ n &\mapsto 3 \\ i &\mapsto 1 \\ a &\mapsto 2 \\ b &\mapsto 3 \end{aligned}$$

Then x gets assigned color 4 and we assume that this leads to the spilling of x because we only have 4 registers that we use for the lowest colors. This is sub-optimal because it means we will have expensive memory operations inside the loop. So we decide to split the live range of x right before the loop by inserting a

move instruction $x' \leftarrow x$.

			live variables
		...	
		$x \leftarrow y$	y, u, v
		$n \leftarrow u + v$	x, u, v
		$i \leftarrow n$	x, n
split	:	$x' \leftarrow x$	x, i
l_1	:	if $(i \leq 0)$ then done else l_2	x', i
l_2	:	$i \leftarrow i - 1$	x', i
		$x' \leftarrow x' * x'$	x', i
		goto l_1	x', i
done	:	$a \leftarrow x + 8128$	x'
	:	$b \leftarrow a + a$	x', a
		return $x' * b$	x', b

Now, x only interferes with i,n,u, and v. The new temp x' interferes with i,a, and b. Assume that the other temps are still colored as before. Then we can color x with color 1 and x' with color 0. In this way, we avoided the spilling of x at the cost of one additional move.

While the greedy coalescing algorithm from the previous section is a good heuristic for deciding whether we should eliminate a move, we do not have good heuristic for splitting live ranges. One problem is that splitting live ranges has to happen before the graph coloring. However, we cannot know if the splitting will indeed lead to a better coloring. It is for instance not a good idea to aggressively split live ranges of all temps that appear in a loop body before every loop and to hope that coalescing will remove unneeded moves. This will lead to quite inefficient code in practice. Instead, we recommend to use splitting only in rare cases or not at all since it is difficult to integrate with chordal graph coloring.

References

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

[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.