Lecture Notes on Register Allocation

15-411: Compiler Design
Frank Pfenning, André Platzer

Lecture 3
September 2, 2014

1 Introduction

In this lecture we discuss register allocation, which is one of the last steps in a compiler before code emission. Its task is to map the potentially unbounded numbers of variables or "temps" in pseudo-assembly to the actually available registers on the target machine. If not enough registers are available, some values must be saved to and restored from the stack, which is much less efficient than operating directly on registers. Register allocation is therefore of crucial importance in a compiler and has been the subject of much research. Register allocation is also covered thoroughly in the textbook [App98, Chapter 11], but the algorithms described there are complicated and difficult to implement. We present here a simpler algorithm for register allocation based on *chordal graph coloring* due to Hack [Hac07] and Pereira and Palsberg [PP05]. Pereira and Palsberg have demonstrated that this algorithm performs well on typical programs even when the interference graph is not chordal. The fact that we target the x86-64 family of processors also helps, because it has 16 general registers so register allocation is less "crowded" than for the x86 with only 8 registers (ignoring floating-point and other special purpose registers).

Most of the material below is based on Pereira and Palsberg [PP05][1], where further background, references, details, empirical evaluation, and examples can be found.

2 Building the Interference Graph

Two variables need to be assigned to two different registers if they need to hold two different values at some point in the program. This question is undecidable in

[1] Available at http://www.cs.ucla.edu/~palsberg/paper/aplas05.pdf

general for programs with loops, so in the context of compilers we reduce this to *liveness*. A variable is said to be *live* at a given program point if it will be used in the remainder of the computation. Again, we will not be able to able to accurately predict at compile time whether this will be the case, but we can approximate liveness through a particular form of *dataflow analysis* discussed in the next lecture. If we have (correctly) approximated liveness information for variables then two variables cannot be in the same register wherever their live ranges overlap, because they may both be then used at the same time.

In our simple straight-line expression language, this is particularly easy. We traverse the program backwards, starting at the last line. We note that the return register, `%eax`, is live after the last instruction. If a variable is live on one line, it is live on the preceding line unless it is assigned to on that line. And a variable that is used on the right-hand side of an instruction is live for that instruction.[2]

As an example, we consider the simple straight-line computation of the fifth Fibonacci number, in our pseudo-assembly language. We list with each instruction the variables that are live *before* the line is executed. These are called the variables *live-in* to the instruction.

			live-in	
f_1	$\leftarrow$	1	$\cdot$	
f_2	$\leftarrow$	1	f_1	
f_3	$\leftarrow$	$f_2 + f_1$	f_2, f_1	
f_4	$\leftarrow$	$f_3 + f_2$	f_3, f_2	
f_5	$\leftarrow$	$f_4 + f_3$	f_4, f_3	
`%eax`	$\leftarrow$	f_5	f_5	
`ret`			`%eax`	return register

The nodes of the *interference graph* are the variables and registers of the program. There is an (undirected) edge between two nodes if the corresponding variables interfere and should be assigned to different registers. There are never edges from a node to itself, because, at any particular use, variable x is put in the same register as variable x. We distinguish the two forms of instructions.

- For an $t \leftarrow s_1 \oplus s_2$ instruction we create an edge between t and any different variable $t_i \neq t$ live after this line, i.e., live-in at the successor. t and t_i should be assigned to different registers, because otherwise the assignment to t could destroy the proper contents of t_i.
- For a $t \leftarrow s$ instruction (move) we create an edge between t and any variable t_i live after this line different from t *and* s. We omit the potential edge between

[2]Note that we do not always have to put the same variable in the same register at all places, but could possibly choose different registers for the same variables at different instructions (given suitable copying back and forth). But SSA already takes care of this issue as we will see later.

t and s because if they happen to be assigned to the same register, they still hold the same value after this (now redundant) move. Of course, there may be other occurrences of t and s which force them to be assigned to different registers.

For the above example, we obtain the following interference graph.

Here, the register `%eax` is special, because, as a register, it is already predefined and cannot be arbitrarily assigned to another register. Special care must be taken with predefined registers during register allocation; see some additional remarks in Section 9.

We could consider another condition, namely create an interference edge if two variables *have overlapping live ranges*, that is, they are both live in to some line in the program. This is overly conservative in that if we have a variable-to-variable move (which frequently occurs as the result of translation or optimizations) then both variables may be live at the next line and automatically be considered interfering. Instead, it is often actually beneficial if they are assigned to the same register because this means the move becomes redundant. So it is not the fact that both variables are live at the same point, but that they are live at the same program point *and* must hold different values which creates the interference.

3 Register Allocation via Graph Coloring

Once we have constructed the interference graph, we can pose the register allocation problem as follows: construct an assignment of K colors (representing K registers) to the nodes of the graph (representing variables) such that no two connected nodes are of the same color. If no such coloring exists, then we have to save some variables on the stack which is called *spilling*.

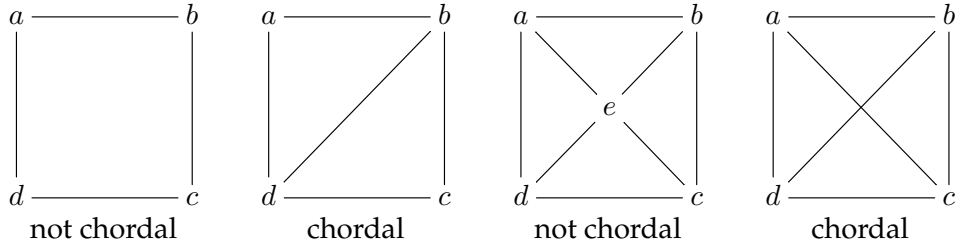
Unfortunately, the problem whether an arbitrary graph is K-colorable is NP-complete for $K \geq 3$. Chaitin [Cha82] has proved that register allocation is also NP-complete by showing that for any graph G there exists some program which has G as its interference graph. In other words, one cannot hope for a theoretically optimal and efficient register allocation algorithm that works on all machine programs.

Fortunately, in practice the situation is not so dire. One particularly important intermediate form is *static single assignment* (SSA). Hack [Hac07] observed that for programs in SSA form, the interference graph always has a specific form called *chordal*. Coloring for chordal graphs can be accomplished in time $O(|V| + |E|)$ (hence at most quadratic in size) and is quite efficient in practice. Better yet, Pereira

and Palsberg [PP05] noted that as much as 95% of the programs occurring in practice have chordal interference graphs anyhow. Moreover, using the algorithms designed for chordal graphs behaves well in practice even if the graph is not quite chordal, which will just lead to unnecessary spilling, not incorrectness. Finally, the algorithms needed for coloring chordal graphs are quite easy to implement compared, for example, to the complex algorithm in the textbook. You are, of course, free to choose any algorithm for register allocation you like, but we would suggest one based on chordal graphs explained in the remainder of this lecture.

4 Chordal Graphs

An undirected graph is *chordal* if every cycle with 4 or more nodes has a chord, that is, an edge not part of the cycle connecting two nodes on the cycle. Consider the following three examples:



Only the second and fourth are chordal (how many cycles need to be checked for chords?). In the other two, the cycle $abcd$ does not have a chord. In particular, the effect of the non-chordality is that a and c as well as b and d, respectively, can safely use the same color, unlike in the chordal case.

On chordal graphs, optimal coloring can be done in two phases, where optimal means using the minimum number of colors. In the first phase we determine a particular ordering of the nodes in which we proceed when coloring the nodes. This order is called *simplicial elimination ordering*. In the second phase we apply *greedy coloring* based on this order. These are explained in the next two sections.

5 Simplicial Elimination Ordering

A node v in a graph is *simplicial* if its neighborhood forms a clique, that is, all neighbors of v are connected to each other, hence all need different colors. An ordering $v_1, \ldots, v_n$ of the nodes in a graph is called a *simplicial elimination ordering* if every node v_i is simplicial in the subgraph $v_1, \ldots, v_i$. Interestingly, a graph has a simplicial elimination ordering if and only if it is chordal. That is, we will not be making a suboptimal decision on those graphs by pretending that all previously

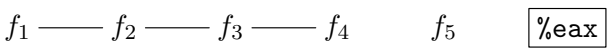
occurring neighbors need to be assigned different colors. Furthermore, the number of colors needed for a chordal graph is at most the size of its largest clique.

We can find a simplicial elimination ordering using *maximum cardinality search*, which can be implemented to run in $O(|V| + |E|)$ time (so at most quadratic in the size of the program). The algorithm associates a weight $\mathrm{wt}(v)$ with each vertex which is initialized to 0 updated by the algorithm. The weight $w(v)$ represents how many neighbors of v have been chosen earlier during the search. We write $N(v)$ for the neighborhood of v, that is, the set of all adjacent nodes.

If the graph is not chordal, the algorithm will still return some ordering although it will not be simplicial. Such an ordering from a non-chordal graph can still be used correctly in the coloring phase, but does not guarantee that only the minimal numbers of colors will be used. Essentially, for non-chordal graphs, generating an elimination ordering in the way described here amounts to pretending that all nodes of the neighborhood are in conflict, which is conservative but suboptimal. For chordal graphs the assumption is actually justified and the correctly allocated registers are also optimal.

Algorithm: Maximum cardinality search
Input: $G = (V, E)$ with $|V| = n$
Output: A simplicial elimination ordering $v_1, \ldots, v_n$
For all $v \in V$ set $\mathrm{wt}(v) \leftarrow 0$
Let $W \leftarrow V$
For $i \leftarrow 1$ to n do
    Let v be a node of maximal weight in W
    Set $v_i \leftarrow v$
    For all $u \in W \cap N(v)$ set $\mathrm{wt}(u) \leftarrow \mathrm{wt}(u) + 1$
    Set $W \leftarrow W \setminus \{v\}$

In our example,



if we pick f_1 first, the weight of f_2 will become 1 and has to be picked second, followed by f_3 and f_4. Only f_5 is left and will come last, ignoring here the node `%eax` which is already colored into a special register. It is easy to see that this is indeed a simplicial elimination ordering.

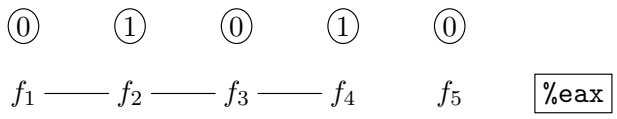
In contrast, $f_2, f_4, f_3, \ldots$ is not, because the neighborhood of f_3 in the subgraph f_2, f_4, f_3 does not form a clique. Indeed, when giving arbitrary (let's say different) colors to f_2 and f_4 in this order, they would require f_3 to assume a third color, which is suboptimal.

6 Greedy Coloring

Given an ordering, we can apply greedy coloring by simply assigning colors to the vertices in this order, always using the lowest available color. Initially, no colors are assigned to nodes in V. We write $\Delta(G)$ for the maximum out-degree of a node in G. The algorithm will always assign at most $\Delta(G)+1$ colors. If the ordering is a simplicial elimination ordering, the result is furthermore guaranteed to be optimal, i.e., use the fewest possible colors.

Algorithm: Greedy coloring
Input: $G = (V, E)$ and ordered sequence $v_1, \ldots, v_n$ of nodes.
Output: Assignment $\mathrm{col} : V \to \{0, \ldots, \Delta(G)\}$.
For $i \leftarrow 1$ to n do
    Let c be the lowest color not used in $N(v_i)$
    Set $\mathrm{col}(v_i) \leftarrow c$

In our example, we would just alternate color assigments:



Of course, `%eax` is represented by one of the colors. Assuming this color is 0 and `%edx` is the name of register 1, we obtain the following program:

```
%eax ← 1
%edx ← 1
%eax ← %edx + %eax
%edx ← %eax + %edx
%eax ← %edx + %eax
%eax ← %eax                    // redundant self move
ret
```

It should be apparent that some optimizations are possible. Some are immediate, such as the redundant move of a register to itself. We discuss another one called *register coalescing* in Section 8.

7 Register Spilling

So consider that we have applied the above coloring algorithm and it turns out that there are more colors needed than registers available. In that case we need to save some temporary values. In our runtime architecture, the stack is the obvious place. One convenient way to achieve this is to simply assign stack slots instead

of registers to some of the colors. The choice of which colors to spill can have a drastic impact on the running time. Pereira and Palsberg suggest two heuristics: (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.

Once we have assigned stack slots to colors, it is easy to rewrite the code using temps that are spilled if we reserve a register in advance for moves to and from the stack when necessary. For example, if `%r11` on the x86-64 is reserved to implement save and restore when necessary, then

$$t \leftarrow t + s$$

where t is assigned to stack offset 8 and s to `%eax` can be rewritten to

$$\begin{array}{lcl} \texttt{\%r11} & \leftarrow & 8(\texttt{\%rsp}) \\ \texttt{\%r11} & \leftarrow & \texttt{\%r11} + \texttt{\%eax} \\ 8(\texttt{\%rsp}) & \leftarrow & \texttt{\%r11} \end{array}$$

Sometimes, this is unnecessary because some operations can be carried out directly with memory references. So the assembly code for the above could be shorter

```
ADDL %eax, 8(%rsp)
```

although it is not clear whether and how much more efficient this might be than a 3-instruction sequence

```
MOVL 8(%rsp), %r11
ADDL %eax, %r11
MOVL %r11, 8(%rsp)
```

We recommend generating the simplest uniform instruction sequences for spill code.

Extensions Heuristic factors that are used for register allocation especially for breaking ties in deciding which temps to spill into the memory 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.

Especially on SSA programs, deciding on register spilling can sometimes be more efficient before final register allocation, which can help the interplay with instruction selection. On SSA programs, register allocation can be done without explicitly constructing the interference graph (based on a postfix order of the dominance tree). The reason is that the central SSA relation called dominance tree defines a simplicial elimination order by doing a prefix traversal order of the dominance tree, such that register allocation is immediate. It, thus, makes sense to reconsider register allocation and interference graph construction for possible simplifications in case you later choose to implement SSA.

8 Register Coalescing

After register allocation, a common further optimization is used to eliminate register-to-register moves called *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 follows the principle

1. Consider each move between variables $t \leftarrow s$ occurring in the program in turn.
2. If t and s are the same color, the move can be eliminated without further action.
3. If there is an edge between them, that is, they interfere, they cannot be coalesced.
4. 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 to maintain a correspondence with the graph.

This simple greedy coalescing will eliminate the redundant self move in the example above. Optimal register coalescing can be done using a reduction to integer linear programming, which can be too slow.

9 Precolored Nodes

Some instructions on the x86-64, such as integer division `IDIV`, require their arguments to be passed in specific registers and return their results also in specific registers. There are also `call` and `ret` instructions that use specific registers and must respect caller-save and callee-save register conventions. We will return to the issue of calling conventions later in the course. When generating code for a straight-line program as in the first lab, some care must be taken to save and restore callee-save registers in case they are needed.

First, for code generation, the live range of the fixed registers should be limited to avoid possible correctness issues and simplify register allocation.

Second, for register allocation, we can construct an elimination ordering as if all precolored nodes were listed first. This amounts to the initial weights of the ordinary vertices being set to the number of neighbors that are precolored before the maximum cardinality search algorithm starts. The resulting list may or may not be a simplicial elimination ordering, but we can nevertheless proceed with greedy coloring as before.

10 Register Allocation in Two-Address Form

The two-address form of instruction used in the x86 family of processors is a special case of the three-address form. As such, our general algorithm should work either way, as long as special restrictions on register usage for operations such as shifts, division, or modulus are respected.

However, one should still consider if register allocation should happen on three-address form or two-address form, provided both are intermediate languages in your compiler. In some sense the "safest" way is to do it on two-address form, just before emitting actual x86-64 assembly. If you perform register allocation on the three-address form you need to make sure that the translation to two-address form does not introduce any new interferences. This requires some care. For example,

$$t \leftarrow x + y$$

might assign t and y to the same register, say, r_1 while x becomes r_0. But now

$$r_1 \leftarrow r_0 + r_1$$

cannot be simply translated to

$$\begin{aligned} r_1 &\leftarrow r_0 \\ r_1 &\leftarrow r_1 + r_1 \end{aligned}$$

because this assigns r_1 the value $x + x$ and not $x + y$. Of course, we can be smarter in the translation and exploit the commutativity of addition to generate the single

two-address instruction

$$r_1 \leftarrow r_1 + r_0$$

You would need to consider all the cases, paying particular attention to non-commutative operations.

In lecture we walked through what happens if the three-address source is in SSA, so the translation to two-address form is straightforward. Doing register allocation on the result may lead to the use of more registers simply because the translation has created additional interferences.

			live-in	interference edges
f_1	$\leftarrow$	1	$\cdot$	$\cdot$
f_2	$\leftarrow$	1	f_1	(f_2, f_1)
f_3	$\leftarrow$	f_2	f_2, f_1	(f_3, f_1)
f_3	$\leftarrow$	$f_3 + f_1$	f_3, f_2, f_1	(f_3, f_2)
f_4	$\leftarrow$	f_3	f_3, f_2	(f_4, f_2)
f_4	$\leftarrow$	$f_4 + f_2$	f_4, f_3, f_2	(f_4, f_3)
f_5	$\leftarrow$	f_4	f_4, f_3	(f_5, f_3)
f_5	$\leftarrow$	$f_5 + f_3$	f_5, f_3	$\cdot$
`%eax`	$\leftarrow$	f_5	f_5	$\cdot$
`ret`			`%eax`	

Our interference graph now requires 3 colors, because f_3 interferes with f_1 and f_2 which also interfere with each other. So our straightforward translation from three-address to two-address code has increased the register pressure.

11 Summary

Register allocation is an important phase in a compiler. It uses liveness information on variables to map unboundedly many variables to a finite number of registers, spilling temporaries onto stack slots if necessary. The algorithm described here is due to Hack [Hac07] and Pereira and Palsberg [PP05]. It is simpler than the one in the textbook and appears to perform comparably. It proceeds through the following passes:

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.

The last step, coalescing, is an optimization which is not required to generate correct code. Variants such as a separate spilling pass before coloring are described in the references above can further improve the efficiency of the generated code.

The same thing is true for find a good ordering of nodes. Instead of maximum cardinality search, we could, for example, just use the order in which they occur in the program. In that case we would have a correct register allocator, but it will sometimes use unnecessarily many registers even on SSA form. This is probably mostly significant for programs that have to spill frequently used variables.

On chordal graphs, which come from SSA programs and often arise directly, register allocations is polynomial and efficient in practice. Optimal register coalescing and optimal spilling, however, are still NP-complete. Even when using heuristics, register allocation may consume the most time during a compiler run.

Questions

1. Why does register allocation take such a long time? It is polynomial isn't it?
2. Is it safe to restrict the interference graph definition for the instruction $t \leftarrow s_1 \oplus s_2$ to the case where t is live after that line?
3. What is the advantage of working with the intuition "overlapping live ranges" compared to the construction given in Section 2?
4. Does it make a difference where we start our register allocation, i.e., where we start the construction of a simplicial order?
5. Is register allocation for programs with mixed data types more difficult than for programs with uniform types? Why or why not?
6. Why is chordality of a graph interesting for register allocation?
7. Why should one worry about allocating half registers of lower data width? Isn't accessing words out of double words etc. inefficient? Is accessing bytes out of words inefficient?
8. Will register coalescing work better on 2-address or 3-address instruction forms?

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.