Lecture Notes on Register Allocation

15-411: Compiler Design
Frank Pfenning, Rob Simmons, André Platzer

Lecture 3
September 8, 2015

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 Spilling Temps

After instruction selection, we have a program that uses a potentially very large number of temps and some specific registers, like `%eax` and `%edx`, that are used in

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

the `idiv` and `ret` instructions. The main topic of today's lecture is turning that program into an *equivalent* program that uses the minimum number of other temps. We'll reduce the number of temps we use by reusing temps in different computations. (We'd also like to reuse `%eax` and `%edx` as much as possible).

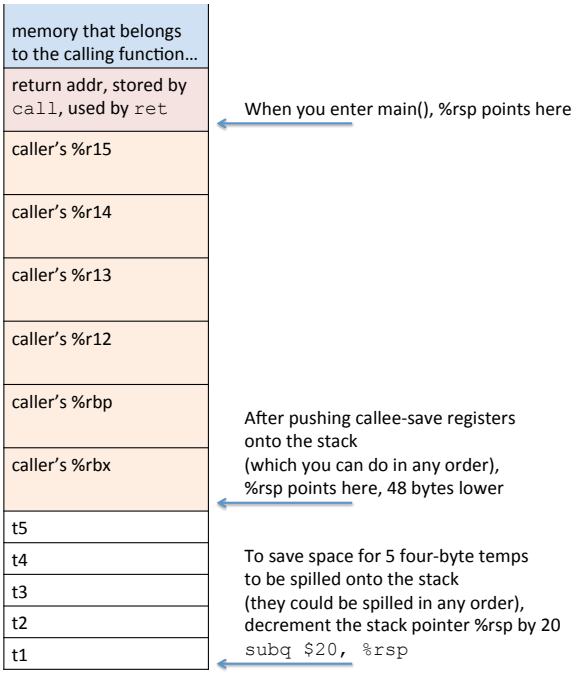
If we can transform a program into another program that uses `%eax`, `%edx`, and no more than 13 other temps, then register allocation is trivial: we can arbitrarily assign those 13 temps to the 13 other general-purpose x86-64 registers (don't mess with `%rsp`). At that point, there is very little distance between our two-address code language and x86-64 assembly: if t_9 gets assigned to `%r14d` and t_{12} gets assigned to `%esi`, then this two-address code instruction:

$$t_{12} \leftarrow t_{12} - t_9$$

can be written as this x86-64 instruction:

```
SUBL %r14d, %esi
```

For some programs, the 15 general-purpose x86-64 registers will not suffice. 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 *assign stack slots instead of registers* to some of the temps: we say that those temps have *spilled* onto the stack. If the temps $t_1 \ldots t_5$ are the temps we assign to be stored on the stack, then after we push the callee-save registers we should subtract 20 from the stack pointer, making room for the (not yet initialized) temps.



Once we have assigned as many temps as possible to registers, and assigned remaining temps to stack slots, 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

$$t_3 \leftarrow t_3 + t_{12}$$

where t_3 is assigned to stack offset 8 as in the example above and t_{12} is assigned to `%edi`, can be rewritten to

$$\begin{array}{lcl} \texttt{\%r11d} & \leftarrow & 8(\texttt{\%rsp}) \\ \texttt{\%r11d} & \leftarrow & \texttt{\%r11d} + \texttt{\%edi} \\ 8(\texttt{\%rsp}) & \leftarrow & \texttt{\%r11d} \end{array}$$

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 %edi, 8(%rsp)
```

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

```
MOVL 8(%rsp), %r11d
ADDL %edi, %r11d
MOVL %r11d, 8(%rsp)
```

We recommend generating the simplest uniform instruction sequences for spill code. It will probably be necessary to dedicate at least one register to spilling: x86-64 does not allow instructions like `ADDL 4(%rsp), 16(%rsp)` that manipulate two memory addresses simultaneously, so to implement $t_5 \leftarrow t_5 + t_2$ in the example above, we would need a spare register and several instructions:

```
MOVL 16(%rsp), %r11d
ADDL 4(%rsp), %r11d
MOVL %r11d, 16(%rsp)
```

3 Interference

We have seen how to perform register allocation after we have already done our best to minimize the number of temps in a program; we will now begin our discussion on how to minimize the number of registers. The first concept we need to talk about is the *interference*. Two destinations (registers or temps) in a program *interfere* if they need to contain different values at the same point in the program. If two temps interfere, then obviously those two temps cannot be assigned to the same register without changing the meaning of the program.

We will not be able to able to accurately predict at compile time whether two destinations interfere (it is undecidable in general), but we will nevertheless see how to construct interference graphs in the next lecture. We manage by thinking about interference as an *over-approximation*. It will be a correctness problem to claim that two destinations do not interfere when they do, because if we record that those destinations do not interfere, we might later on assign them to the same register. We must be careful never to claim that registers do not interfere when they do! It is merely an efficiency problem to claim that two destinations do interfere when they, in reality, do not. We can always claim that two registers interfere if it makes our analysis easier.

$x_0 \leftarrow 0$
$x_1 \leftarrow 1$
$x_2 \leftarrow x_1 + x_0$
$x_3 \leftarrow x_2 + x_1$
$x_4 \leftarrow x_3 + x_2$
`%eax` $\leftarrow x_4$
ret `%eax`

In the program above, which computes the fourth Fibonacci number, x_0 and x_1 interfere. There are severaly ways to look at this. One observation is that, when we compute x_2, we need to have both the values x_0 and x_1 on hand, so those values need to be in distinct registers. A better way of thinking about the problem, though, is that between where x_0 is defined (the first line) and where it is used (the third line), we move a new value into x_1. If we assigned both x_0 and x_2 to the same register `%edx`, then we would have

`%edx` $\leftarrow 0$
`%edx` $\leftarrow 1$
$x_2 \leftarrow$ `%edx` $+$ `%edx`

which would result in x_2 getting the wrong value, 2 instead of 1.

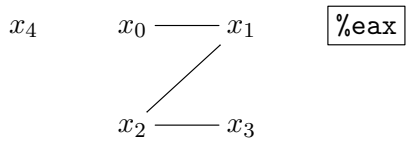
While it is true that in the program above, x_1 and x_2 only ever contain the value 1, any conceivable interference analysis will conclude that x_1 and x_2 interfere for the same reasons that x_0 and x_1 interfere, and for the same reasons that x_2 and x_3 interfere. On the other hand, x_4 and `%eax` don't interfere with anything: we could map x_4 to any register, *including* `%eax`, without changing the meaning of the program.

4 Interference Graphs

Since the work of Chaitin [Cha82] in the early 80s, compiler designers have generally thought about interference in terms of an *interference graph*, a graph where

the vertices are all the registers and temps in 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.

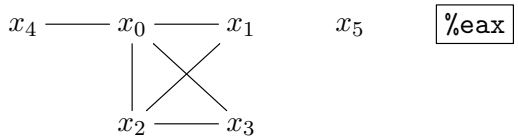
Let's look at some examples of interference graphs. The example of computing Fibonacci in the previous section had three interference. It has a simple interference graph with three edges:



If we add a new instruction $x_5 \leftarrow x_1 + x_4$ near the end of our Fibonacci example, the interference graph changes dramatically.

$x_0 \leftarrow 0$
$x_1 \leftarrow 1$
$x_2 \leftarrow x_1 + x_0$
$x_3 \leftarrow x_2 + x_1$
$x_4 \leftarrow x_3 + x_2$
$x_5 \leftarrow x_4 + x_0$
`%eax` $\leftarrow x_5$
ret `%eax`

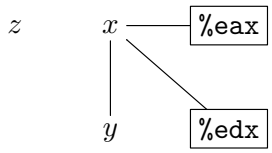
Not only do x_0 and x_4 interfere because they're used at the same time, x_0 also now interferes with x_1, x_2, and x_3 because those temps are written to in between where x_0 is defined and when it is used to compute x_5. Here is the resulting interference graph:



In the examples above, the designated register `%eax` does not interfere with any temps, because the only time we used a designated register was at the very end, for a return statement.

Uses of designated registers in the middle of code (for instance, for division), may cause other interference edges, as this program and its corresponding interference graph demonstrates:

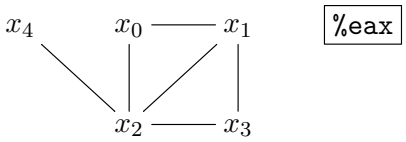
```
%eax ← 11
x ← 4
%edx ← %eax % x
y ← %edx
z ← x + y
%eax ← z
ret %eax
```



The point of designated registers is that they cannot be changed to other registers, so all designated registers automatically interfere with one another – we may to draw the edges or not.

We also may up with more interference edges after we translate code from three-address form to two-address form. If we take our original example and put it into two-address form, we get the following program:

```
x0 ← 0
x1 ← 1
x2 ← x1
x2 ← x2 + x0
x3 ← x2
x3 ← x3 + x1
x4 ← x3
x4 ← x4 + x2
%eax ← x4
ret %eax
```



The interference edges (x_0, x_1), (x_1, x_2), and (x_2, x_3) all still exist, but additionally, we cannot assign x_0 and x_2 to the same register, as we assign to x_2 (on the third line) in between the definition of x_0 on the first line and its use on the fourth line. We end up with three additional edges: (x_0, x_2), (x_1, x_3), and (x_2, x_4).

5 Register Allocation via (Greedy) 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. We will refer to the colors by number: ①, ②, ③, …(K). The designated registers are treated as *pre-colored* nodes in the graph

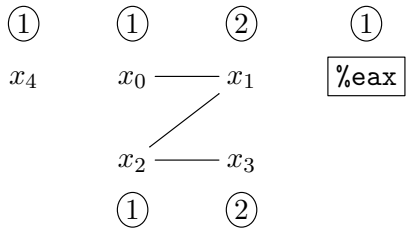
whose colors we can't change. In the examples below, we will associate the register `%eax` with the color ① and associate the register `%edx` with the color ②.

Unfortunately, the problem whether an arbitrary graph is K-colorable is NP-complete for $K \geq 3$. Chaitin [Cha82] has proved that optimal 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. Strictly speaking, we don't need the best possible graph coloring. If we use too many colors, we will end up using more stack space than necessary, an efficiency issue but not a correctness issue. The simplest greedy algorithm might be good enough, at least as a first pass.

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 $\mathrm{col}(v_i) \leftarrow c$

The ordered sequence makes all the difference here. For our original example, if we pick the ordering x_0, x_1, x_2, x_3, x_4, we will end up with an optimal 2-coloring using greedy coloring:



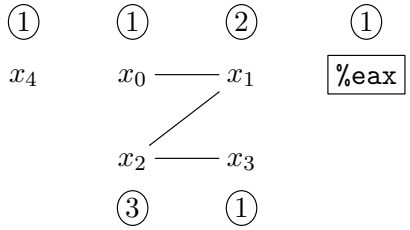
This graph coloring would cause us translate our original program into:

```
%eax ← 0
%edx ← 1
%eax ← %edx + %eax
%edx ← %eax + %edx
%eax ← %edx + %eax
%eax ← %eax
ret %eax
```

Ignoring for a moment the peculiar use of registers in for a program that is not in 2-address code, it should be apparent that some optimization is possible. Some are

immediate, such as removing the redundant move of a register to itself. We will discuss another optimization, called *register coalescing*, later in the semester.

For any graph, there is *some* ordering for which the greedy algorithm produces the optimal coloring, though we certainly shouldn't expect to easily find such an order efficiently. On some graphs, greedy graph coloring behaves quite badly, though on the graph from our first example, we cannot force greedy graph coloring to do too badly: the ordering x_0, x_3, x_1, x_2, x_4 is an example of a pessimal ordering and results in the following 3-coloring:

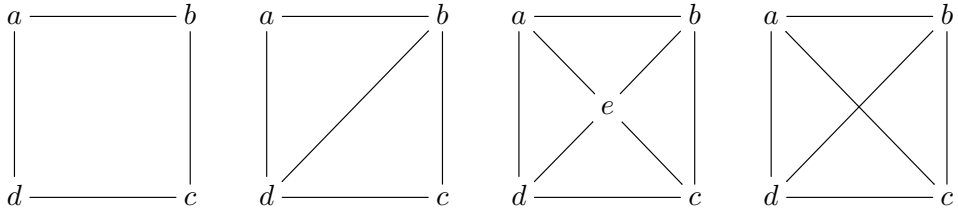


This still represents taking a program that used 6 destinations (x_0, x_1, x_2, x_3, x_4, and `%eax`) and turning it into a program that uses 3 (`%eax`, `%edx`, and whatever register we map color (3) to), a significant improvement.

6 Chordal Graphs

It's not possible[2] to efficiently come up with optimal orderings for greedily coloring arbitrary graphs. However, most programs have interference graphs with a particular form, called *chordal*. For chordal graphs, it is possible to efficiently find an optimal ordering. 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.

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:



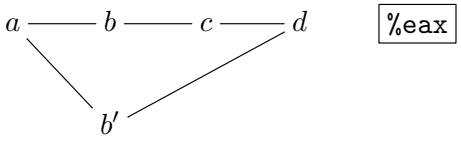
[2] Assuming that $P \neq NP$, at least.

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 both cases, 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.

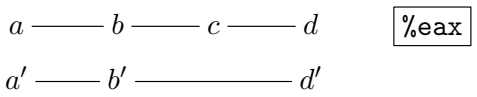
All the interference graphs we've looked at so far are chordal! Creating a non-chordal interference graph requires us to *re-use* temps in a somewhat unusual way, as in the following program and corresponding chordal graph:

$a \leftarrow 0$
$b \leftarrow 1$
$c \leftarrow a + b$
$d \leftarrow b + c$

$a \leftarrow c + d$
$b' \leftarrow 7$
$d \leftarrow a + b'$
`%eax` $\leftarrow b' + d$
ret `%eax`



This coding pattern is uncommon enough that Pereira and Palsberg [PP05] noted that something like 95% of the programs occurring in practice have chordal interference graphs already. Furthermore, note that the graph above, with a cycle of length 5, requires 3 colors. If we replaced the assignments to a and d in the lower part of the program (and the corresponding uses on the last 3 lines) with a' and d', we'd have an equivalent program with this interference graph, which only requires two colors:



In other words, allocating *more* temps led to us needing *fewer* registers! In a few weeks, we will see why we might want, in general, to transform programs into *static single assignment* (SSA) form. That transformation will have the effect of automatically rewriting the program above with a' and d', giving it the 2-colorable interference graph. Hack observed that *all* SSA programs are chordal [Hac07]. Therefore, if we transform our programs into SSA form, we can be sure that our interference graph will be chordal.

7 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

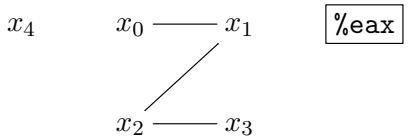
ordering $v_1, \ldots, v_n$ of the nodes in a graph is called a *simplicial elimination ordering* if every node v_i is simplicial in the subgraph $v_1, \ldots, v_i$. Interestingly, a graph has a simplicial elimination ordering if and only if it is chordal. That is, we will not be making a suboptimal decision on those graphs by pretending that all previously occurring neighbors need to be assigned different colors. That, in turn, is exactly the assumption we need in order for greedy graph coloring to give us an optimal ordering!

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 (because *any* ordering will do), but that ordering will 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.

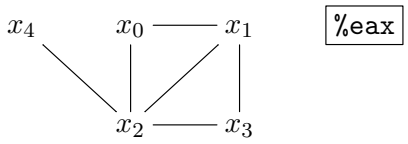
Algorithm: Maximum cardinality search
Input: $G = (V, E)$ with $|V| = n$
Output: A simplicial elimination ordering $v_1, \ldots, v_n$
For all $v \in V$ set $\mathrm{wt}(v) \leftarrow 0$
Let $W \leftarrow V$
For $i \leftarrow 1$ to n do
    Let v be a node of maximal weight in W
    Set $v_i \leftarrow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



if we pick x_0 first, the weight of x_1 and x_2 will become 1 and has to be picked second, followed by x_2 and x_3. Only x_4 is left and will come last, ignoring here the node `%eax` which is already colored into a special register.

On the other hand, if we pick x_0 first in this interference graph



then both x_1 and x_2 will be given weight 1, and either one of them can be picked second. Alternatively, if we picked x_4 first in this graph, we would be forced to pick x_2 second but could then pick any of x_0, x_1, or x_3 third.

8 Summary

Register allocation is an important phase in a compiler. It uses information about the interference of destinations to map an unbounded number of temps 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. We have covered the algorithm backwards. In an implementation, we would proceeds through the following steps:

1. **Build** the interference graph (we will learn how to do this in the next lecture).
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; we will talk about it later in the course, when we consider optimization. Variants such as a separate spilling pass before coloring are described in the references above can further improve the efficiency of the generated code. On chordal graphs, which come from SSA programs and often arise directly, register allocations is polynomial and efficient in practice.

Questions

1. Why does register allocation take such a long time? It is polynomial isn't it?
2. For all the interference graphs in Section 4, what is the a best possible ordering for greedy graph coloring, and how many colors does it use? What is the *worst* possible ordering, and how many colors does it use?

3. Given an optimal graph coloring, how would you construct an ordering such that greedy graph coloring would re-create that coloring?

4. What is the minimum number of 3-address code instructions, ending with `ret %eax`, needed to make a non-chordal graph? What is the minimum number of 2-address code instructions?

5. Does it make a difference where start the construction of a simplicial elimination order?

6. Is register allocation for programs with mixed data types more difficult than for programs with uniform types? Why or why not?

7. Why is chordality of a graph interesting for register allocation?

8. 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?

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.