Lecture Notes on Register Allocation

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

Lecture 3 and Lecture 4
Sept. 4 and Sept. 6, 2018

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 extensively studied. 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 different 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 large number of temps and potentially some specific registers, like `%eax` and `%edx`, that are used

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

in the `idiv` and `ret` instructions. Our goal is to turn that program into an *equivalent* program that as few temps as possible. We 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).

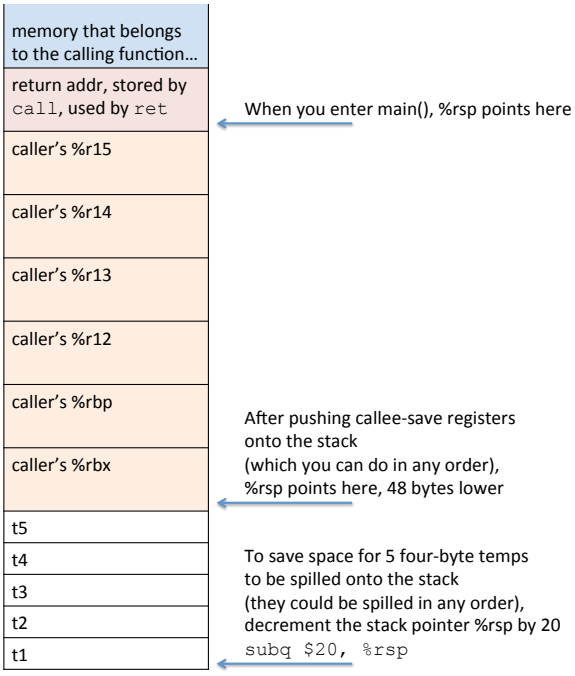
If we can transform a program into another program that uses `%eax`, `%edx`, and no more than 13 other temps, then register allocation is trivial: we can arbitrarily assign those 13 temps to the 13 other general-purpose x86-64 registers (don't mess with `%rsp`). At that point, there is very little distance 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 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 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 used at the same time.

In our simple straight-line expression language, this is relatively 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.

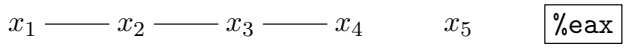
			live-in	
x_1	$\leftarrow$	1	$\cdot$	
x_2	$\leftarrow$	1	x_1	
x_3	$\leftarrow$	$x_2 + x_1$	x_2, x_1	
x_4	$\leftarrow$	$x_3 + x_2$	x_3, x_2	
x_5	$\leftarrow$	$x_4 + x_3$	x_4, x_3	
`%eax`	$\leftarrow$	x_5	x_5	
`ret`			`%eax`	return register

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 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 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.

[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.

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 7.

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.

4 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: $\textcircled{1}$, $\textcircled{2}$, $\textcircled{3}$, $\ldots\textcircled{K}$. 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 $\textcircled{1}$ and associate the register `%edx` with the color $\textcircled{2}$.

Unfortunately, the problem whether an arbitrary graph is K-colorable is NP-complete for $K \geq 3$. Chaitin [Cha82] proved that for any graph G there exists some program which has G as its interference graph.

Sometimes you will see the statement that this proof shows that optimal register allocation is also NP-complete. In other words, one cannot hope for a theoretically optimal and efficient register allocation algorithm that works on all machine programs. However, there are a few caveats that we should keep in mind. First, it is undecidable if two variables interfere. So if G *is the interference graph* of a program depends on the approximation that we pick. Second, register allocation with graph coloring requires that we use one register for one temp throughout the program. However, it is also possible to use multiple registers for one temp (splitting live ranges) and ending up with fewer required registers. Similarly, in the context of a compiler, we can rewrite the program using a compiler phase (such as translation to SSA form) and arrive at a equivalent program that needs fewer registers.

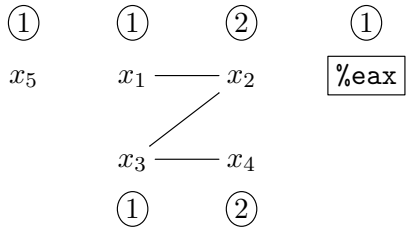
A good formulation of register allocation as a decision problem would be as follows. Given a program P and K registers. Is there an equivalent program P' that

uses only the registers (but no temps or memory locations). For a Turing complete language (for example, assembly with jumps) this problem is undecidable.

Fortunately, the situation is not so dire in practice. 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. In the algorithm, $\Delta(G)$ is the number of colors that is used by the algorithm to color the graph G. We write $N(v)$ for the neighborhood of v, that is, the set of all adjacent nodes.

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_1, x_2, x_3, x_4, x_5, we will end up with an optimal 2-coloring using greedy coloring:



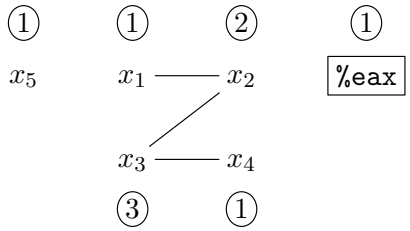
This graph coloring would cause us translate our original program into:

```
%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.

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_1, x_4, x_2, x_3, x_5 is an example of a pessimal ordering and results in the following 3-coloring:

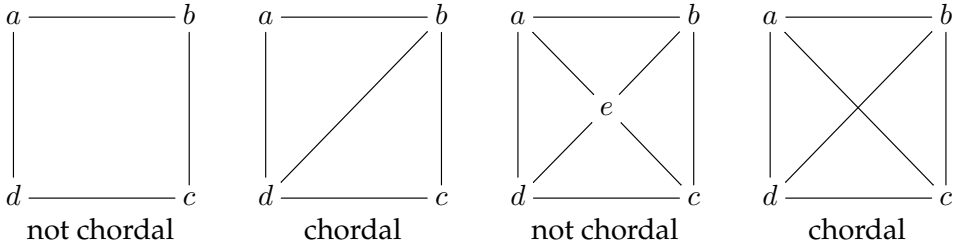


This still represents taking a program that used 6 destinations (x_1, x_2, x_3, x_4, x_5, and `%eax`) and turning it into a program that uses 3 (`%eax`, `%edx`, and whatever register we map color (3) to), a significant improvement.

5 Chordal Graphs

It's not possible[3] 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 Appel's 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:



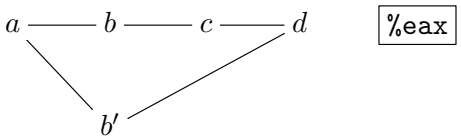
Only the second and fourth are chordal (how many cycles need to be checked for chords?). In the other two, the cycle $abcd$ does not have a chord. In both cases, the effect of the non-chordality is that a and c as well as b and d, respectively, can safely use the same color, unlike in the chordal case.

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

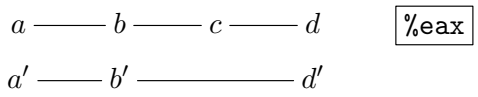
All the interference graphs we've looked at so far are chordal! Creating a non-chordal interference graph requires us to *re-use* temps in a somewhat unusual way, as in the following program and corresponding chordal graph:

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

$a \leftarrow c + d$
$b' \leftarrow 7$
$d \leftarrow a + b'$
%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 would have an equivalent program with this interference graph, which only requires two colors:



Such a transformation is also called *splitting live ranges* (see Section 9.2). Interestingly, 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.

6 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. The proof of this statement is not trivial.

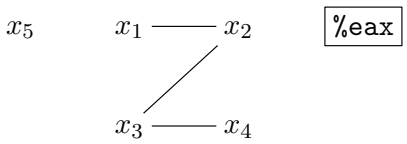
If we use a simplicial elimination ordering in the greedy graph coloring algorithm then we will not make suboptimal decisions by pretending that all previously occurring neighbors need to be assigned different colors. It is not difficult, to prove by induction that a simplicial elimination ordering leads to an optimal greedy coloring and that the the minimal number of colors needed for every subgraph that arises is the size of the largest clique. (More generally, chordal graphs are so-called *perfect graphs*.)

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 again $N(v)$ for the neighborhood of v. The proof that maximum cardinality search returns a simplicial elimination ordering for a graph G if and only if G is chordal is not straightforward.

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 $W \leftarrow W \setminus \{v\}$

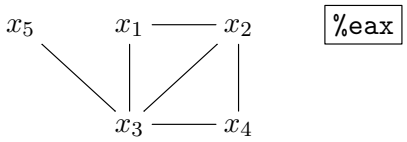
In our running example,



if we pick x_1 first, the weight of x_2 and x_3 will become 1 and has to be picked

second, followed by x_3 and x_4. Only x_5 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_1 first in this interference graph



then both x_2 and x_3 will be given weight 1, and either one of them can be picked second. Alternatively, if we picked x_5 first in this graph, we would be forced to pick x_3 second but could then pick any of x_1, x_2, or x_4 third.

Complexity It is not immediately clear how the algorithm can be implemented to achieve a running time linear on $|V| + |E|$. In the original paper by Tarjan and Yannakakis, the algorithm is described a slightly different. Instead of maintaining a weight for each vertex, we use an array of buckets B so that $B[i]$ contains all vertices of weight i. At the beginning, all vertices are in bucket $B[0]$. We also maintain a counter c that indicates the largest non-empty bucket. We then pick a vertex from bucket $B[c]$ and if it is empty we decrement c and look at the next bucket until we find something (and if not then we terminate). After we picked a vertex, we move all neighbors of that vertex to a higher bucket. So for instance, if we picked v and u is a neighbor of v in bucket $B[i]$ then we move u to $B[i + 1]$.

In the formulation above, we could say that we do not have a map $V \to \{1, ..., K\}$ but rather an array W of size K that contains the doubly linked lists of vertices of that weight. Depending on the representation we use, we also need pointers to find the position of vertices in the lists to move them around in constant time.

7 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.

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 described by Appel [App98, Chapter 11] 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.

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.

9 Optimizations

There are several optimizations that can be performed during register allocation. They are not required to generate correct code but can improve performance. We will talk more about optimizations later in the course. For now, we briefly discuss two *coalescing* and *splitting live ranges* which are dual optimizations that have somewhat conflicting goals.

9.1 Coalescing

Coalescing is a technique for reducing code seize at the expense of putting more stress on the register allocation. Consider a program and its corresponding conflict graph. Assume we have a move instruction $t \leftarrow s$ in the code so that there is no edge between s and t in the conflict graph. Consequently, s and t can be assigned the same register or memory location.

The idea if coalescing is to merge (coalesce) the nodes s and t in the conflict graph by creating a new node that has all the edges of s and t. The color that gets assigned to the new node during register allocation is then considered the color of s and t. In this way, we force the register allocation to assign the same register to both temps. Now the instruction $t \leftarrow s$ becomes a self move that can be safely removed from the program.

The prize that we pay for coalescing is that the coloring of the interference graph becomes more difficult as the graph becomes denser. It is possible that coalescing results in additional colors during the coloring phase and in additional spills. In general, it is hard to decide when coalescing is possible without additional spills. As a rule of thumb, coalescing works well in programs that require only few registers. Appel's textbook contains strategies that can be used to decide when coalescing does not result in additional spills. However, these strategies are tight to specific algorithms for register allocation in the sense that they do not result in additional spills only if you use a particular algorithm.

9.2 Splitting Live Ranges

Splitting live ranges is the dual strategy to coalescing. We insert additional moves to make the interference graph less dense and register allocation easier. The idea is that the cost of the additional move is offset by the gains of reduced spilling.

While coalescing combines notes in the interference graphs, splitting live ranges splits a node into two. Transforming a program to SSA form can be seen as a form of splitting live ranges. We introduce a new name for a temp and in this way create a new node in the interference graph. Transforming a program to SSA form does not necessarily introduce additional move instruction. But even if a program is in SSA form, it is possible to further split live ranges.

Assume a temp t is initialized at the beginning of the code and read at multiple places throughout the code. Then t has potentially a long live range and interferes with many other temps. To reduce the interference, we introduce a new move instruction $t' \leftarrow t$ somewhere in the live range of t and rename t to t' in the instructions after the new move. The result in the interference graph is that the node t is split into two nodes t and t' that do not interfere. The edges of t in the old graph become the edges of t and t' in the new graph.

Similar as for coalescing, it is not immediately clear if splitting the live range of a variable helps. It can for instance be the case that many edges in the original graph get duplicated and become edges of t and t'. This might then not reduce the number of spills.

Questions

1. Why does register allocation take such a long time? It is polynomial isn't it?
2. Given an optimal graph coloring, how would you construct an ordering such that greedy graph coloring would re-create that coloring?
3. 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?

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

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.