CSE 302: Compilers | Project

Register Allocation

Starts: 2023-11-20

1 SYNOPSIS

In this project you will implement register allocation for TAC that will generate more efficient x64 using the full complement of hardware general purpose registers (GPR). The following will be expected of you.

- Crude SSA generation and its minimization
- Construction of the interference graph
- Register allocation by graph coloring using max cardinality search
- (Stretch Goal) Register coalescing
- Producing x64 from ETAC

Warning

Stretch goals are worth extra credit, but should be attempted only after completing all other goals.

2 STATIC SINGLE ASSIGNMENT FORM (SSA)

2.1 *Crude SSA Generation*

SSA is represented using *versioned temporaries* and *phi-functions*. In order to define them, we slightly enlarge the TAC language as follows:

⟨instr⟩ ::= ··· | ⟨var⟩ '=' 'phi' '(' ⟨phi-args⟩ ')' ';'

⟨phi-args⟩ ::= (⟨phi-arg⟩ (',' ⟨phi-arg⟩)*)?

⟨phi-arg⟩ ::= LABEL ':' TEMP

TEMP ::≈ @(0|[1-9][0-9]*|[A-Za-z][A-Za-z0-9_])*(\.[0-9]+)?

To represent a phi-function definition, you can use an tac instruction class instance with opcode phi, taking one argument which is the argument list as a dictionary mapping labels to temporaries. So, for example, `@0.0 = phi(@.L1:@0.1, @.L2:@0.3);` can be represented as:

```
TAC('%0.0', 'phi', {'@.L1': '%0.1', '@.L2': '%0.3'}, None)
```

As mentioned in the lecture, the crude SSA generation algorithm has the following steps:

Algorithm: Crude SSA Generation

1. Add `phi`-function definitions for all temporaries that are live-in at the start of each block. These `phi`-function definitions must come before any other instruction in the block.
2. Uniquely version every temporary that is def'd by any instruction in the entire CFG.
3. Update the uses of each temporary within the same block to their most recent versions.
4. For every edge in the CFG fill in the arguments of the `phi` functions. For an edge from block `@.L1` to `@.L2`, there would be a `phi`-argument (in the `@.L2` block) of the form `@.L1:@n` for every temporary `@n` that comes to `@.L2` from `@.L1`.

An example of crude SSA generation was worked out in detail in lecture 8.

2.2 *SSA Minimization*

Once you have a crude SSA, you will need to perform a number of minimization transformations (in any order) to simplify and eliminate redundancies in the SSA. For this course we will only implement two minimization procedures: *null choice elimination* and *rename elimination*.

NULL CHOICE ELIMINATION This step removes redundant `phi`-function definitions from the SSA.

Null Choice Elimination

The following kind of instruction may be removed anywhere it occurs.

```
@1.i = phi(@.L1:@1.i, @.L2:@1.i, ..., @.Lk:@1.i);
```

Here `@1` stands for any temporary root, and i stands for any version.

RENAME ELIMINATION A *rename* is a `phi`-function definition where, once you remove the temporary in the LHS from the arguments of the `phi`-function, you are left with only a single temporary in the arguments (possibly repeated). This case is almost exactly like a `copy` instruction, and can be eliminated by uniting the LHS and RHS versions.[1]

Rename Elimination

Given an instruction of the following form

```
@1.u = phi(@.L1:@1.v1, @.L2:@1.v2, ..., @.Ln:@1.vn);
```

where `@1` stands for any temporary root and $u, v_1, v_2, \ldots, v_n$ are versions:

- if there exists $v \neq u$ such that $\{v_1, v_2 \ldots, v_n\} \subseteq \{u, v\}$,
- then replace every occurrence of `@1.`u with `@1.`v in every instruction.

[1] You can use a union-find data structure to implement this efficiently.

3 REGISTER ALLOCATION

Next, you will allocate as many registers as possible to temporaries. In order to do this, you will use a graph coloring algorithm that is described in this section.

Note: Procedure Calls

If your CFG contains procedure calls (i.e., the instructions `param` and `call`), then you are not required to perform register allocation. In other words, all temporaries in such CFGs can be left on the stack like you already did in lab 4.

COLOR MAPPING In the register allocation algorithms to follow, you will be trying to color the temporaries with a color set $K = \{1, 2, \ldots, k\}$ for some $k \in \mathbb{N}$. This color set is usable for `x64` only if $k \leq 13$. When such a coloring has been obtained, it is then important to map color numbers to the registers in some canonical way. Unless you have a strong reason to prefer a different mapping, use this one:

```
color_map = (
    '%%rax', '%%rcx', '%%rdx', '%%rsi', '%%rdi', '%%r8', '%%r9', '%%r10',
    '%%rbx', '%%r12', '%%r13', '%%r14', '%%r15', '%%r11', '%%rbp', '%%rsp'
)
reg_map = {reg: color_map.index(reg) + 1 for reg in color_map}
def color_to_reg(col):   return color_map[col - 1]
def reg_to_color(reg):   return reg_map[reg]
```

It gives lower color values to caller-save registers, so the lower the register pressure in a function the less likely you are to need a callee-save register (which would need an additional `pushq`/`popq`).

OUTLINE Register allocation proceeds in the following order.

0. Start from SSA (see sec. 2)
1. Compute the interference graph (sec. 3.1)
2. Use max-cardinality search to find a simplicial elmination ordering (SEO) (sec. 3.2)
3. Use greedy coloring based on the SEO (sec. 3.3)
4. If needing > 13 colors, spill some temporary and redo from step 2 (sec. 3.4)
5. Finally, compute the allocation record (sec. 3.5)

3.1 *Interference Graph*

From a CFG, the first step is to compute the *interference graph*, which is an undirected graph where:

- the nodes consist of TAC temporaries; and
- an edge between two nodes means that the nodes *cannot* be allocated to the same machine register.

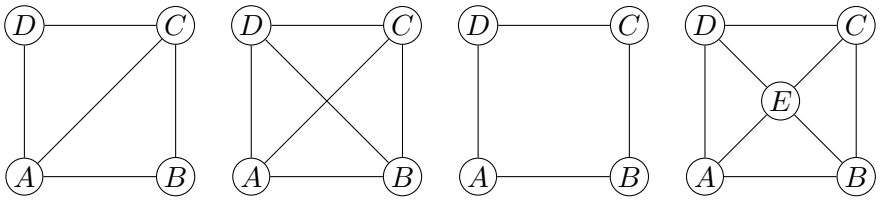
The register allocation problem is equivalent to coloring the interference graph using as many colors as available machine registers (13).

To create an interference graph, you first need to compute liveness information for every instruction. As you saw in lecture, these sets are defined in terms of the sets: use( ), def( ), livein( ), and liveout( ). Based on these definitions, we can say that an edge in the interference graph is created if any of the following rules apply for any instruction I in the control flow graph.

- If I is a `copy` instruction, then every temporary $x \in \mathsf{liveout}(I)$ interferes with every temporary $y \in \mathsf{use}(I) \cup \mathsf{def}(I)$.
- If I is any other instruction, then every temporary $x \in \mathsf{liveout}(I)$ interferes with every temporary in $y \in \mathsf{def}(I)$.

Note that in these rules, we implicitly assume that no self-loop is created in the interference graph.

It turns out that the SSA guarantees that the inteference graph is *chordal*. That is, every cycle with 4 or more nodes in the interference graph has a *chord*, which is an edge that links nodes that are not immediate neighbors in the cycle. The following are four graphs of which the first two are chordal but the second two are not because the cycle $A - B - C - D$ does not have a chord.



3.2 *Simplicial Elimination Orderings (SEO) and Max-Cardinality Search*

The graph coloring algorithm in the next section is based on computing an ordering of the nodes of the interference graph. As explained in lecture, we are looking for a *simplicial elimination ordering*. A node in the interference graph is *simplicial* if the nodes it is connected to are all connected to each other by single edges.[2] An ordering of nodes $v_1, v_2, \ldots, v_n$ is said to be a *simplicial elimination ordering* (SEO) if each v_i is simplicial in the subgraph of the interference graph formed by the nodes $v_1, \ldots, v_i$.

To find a SEO, we will use the so called *max-cardinality search* algorithm. In this algorithm each node v of the graph is assigned a *weight* $\mathsf{wt}(v)$, which is initialized to 0 and updated during the algorithm. (Intuitively, $\mathsf{wt}(v)$ stands for the number of neighbors of v that have been selected for the final ordering before v itself.) The algorithm is given in figure 1. The proof that it produces a SEO for chordal graphs is non-trivial.

3.3 *Greedy Coloring*

Using an elimination ordering to color the interference graph is a simple matter. The algorithm is shown in fig. 1. If the ordering is a SEO, then the greedy algorithm is guaranteed to find the minimal number of colors required to color the chordal interference graph. Obviously this property is not guaranteed to hold for other kinds of interference graphs and elimination orderings. However, the algorithm always returns a valid coloring.

The greedy algorithm takes a *partial coloring* as input, which is a color assigned to some of the temporaries. These color assignments come from the calling conventions. In particular, the temporaries that contain the input arguments of the procedure are preassigned the colors corresponding to the input registers. Likewise, the temporar(y/ies) containing the return value of the procedure should be precolored to the color mapped to RAX.

[2] That is, the successors of the node form a clique.

Algorithm: Max Cardinality Search

Input: $G = \langle V, E \rangle$, an interference graph with $|V| = n$.

Output: a simplicial elimination ordering $[v_1, v_2, \ldots, v_n]$ where $V = \{v_1, v_2, \ldots, v_n\}$.

1. Create a mapping $\mathsf{wt}() : V \to \mathbb{N}$ and initialize $\mathsf{wt}(v) \leftarrow 0$ for all $v \in V$.
2. Let $W \leftarrow V$ (i.e., initially a copy of the vertex set V).
3. For i from 1 to n:
   - Set $v_i \leftarrow \operatorname{argmax}_{w \in W} \mathsf{wt}(w)$. (*i.e., the ith element of the SEO.*)
     (If more than one node can be picked for v_i, pick one arbitrarily.)
   - For all $w \in W \cap \mathsf{next}(v_i)$: set $\mathsf{wt}(w) \leftarrow \mathsf{wt}(w) + 1$
   - $W \leftarrow W \setminus \{v_i\}$

(a) max-cardinality search

Algorithm: Greedy Coloring

Input: $G = \langle V, E \rangle$ and an elimination ordering $[v_1, \ldots, v_n]$ of V.
Input: A partial coloring $\mathsf{col} : V \rightharpoonup K$ where $K = \{1, 2, \ldots, k\}$ for some $k \in \mathbb{N}$.
Output: A complete coloring $\mathsf{col} : V \to K$

1. For every $u \in V$, if $u \notin \mathsf{dom}(\mathsf{col})$, then initialize $\mathsf{col}(u) \leftarrow 0$.
2. For i from 1 to n:
   - If $\mathsf{col}(v_i) \neq 0$, skip to the next iteration of this loop.
   - Let $c \in K$ be the smallest color that is not used as a neighbor of v_i by col; i.e.,
     $$c = \operatorname*{argmin}_{c \in K} \text{for every } w \text{ with } (v_i, w) \in E\text{: } \mathsf{col}(w) \neq c.$$
     Note that $c \neq 0$ because $0 \notin K$.
   - Set $\mathsf{col}(v_i) \leftarrow c$.

(b) coloring based on elimination orders

Figure 1: Graph algorithms

3.4 *Spilling Temporaries*

If the greedy coloring step returns a minimal coloring col : $V \to K$ with $|K| > 13$, then the interference graph cannot be colored without *spilling* some temporaries to the stack. Specifically, $|K| - 13$ temporaries need to be spilled. To spill a register, perform the following steps:

1. Pick a temporary and assign it a stack slot. This temporary will not be allocated to a register. Note that you cannot pick one of the precolored temporaries (input arguments, output value) to spill, because they are required to have a color.
2. Remove that temporary from the interference graph. This will remove that node and all edges that have the node as an endpoint from the graph.
3. Recompute the SEO with max-cardinality search and retry coloring.

This process will have to be repeated until you obtain a coloring with ≤ 13 colors.

Selecting good temporaries to spill can be tricky. A good rule of thumb is to spill those temporaries that stay live for the shortest ranges. However, any spillable temporary can be chosen.

3.5 *Allocation Record*

After you have successfully computed the coloring for the registers, you will have to compute an *allocation record* for this CFG. The allocation record contains the following fields:

- How much stack space to allocate for spilled temporaries
- Which registers are mapped to which temporaries

More specifically, the activation record is represented as additional keys to the `'proc'` object with the following mappings:

- `'stacksize'`: an integer, representing the number of bytes that will be taken up by temporaries on the stack in the current (callee's) stack frame.
- `'alloc'`: a dictionary that maps each temporary to one of the following:
  - one of the x64 GPRs, written i.e., one of the following strings:

    ```
    {'%%rax', '%%rcx', '%%rdx', '%%rsi', '%%rdi', '%%r8', '%%r9', '%%r10',
     '%%rbx', '%%r12', '%%r13', '%%r14', '%%r15', '%%r11', '%%rbp', '%%rsp'}
    ```

  - an RBP offset, which is an integer. Negative values indicate temporaries, while positive values indicate stack arguments.

An example of the allocation record is given in figure 2, with the extra fields shown as a comment. In the TAC (JSON) representation, you would see something like:

```
{ 'proc': '@_fib',
  'args': ['%n'],
  'stacksize': 8,
  'alloc': { '%n': '%%rdi', ... },
  'body': [
    { 'opcode': 'label', 'args': [ '@.L0' ], 'result': null },
    ...
  ] }
```

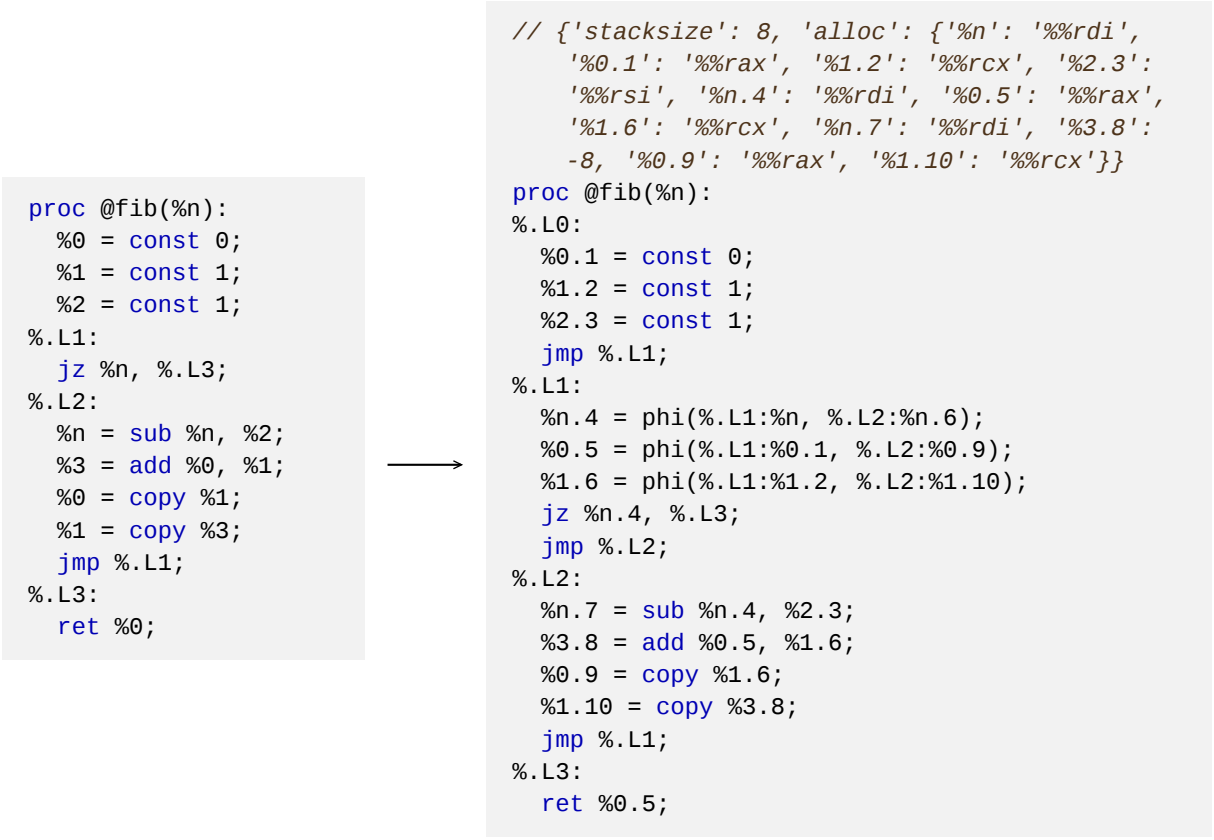


Figure 2: An example of allocation

4 REGISTER COALESCING (STRETCH GOAL)

A copy between two non-interfering temporaries can be removed if there is a free color among their neighbors. Specifically, the free color is used to create a new pre-colored temporary and then both the source and destination temporary in the entire program is rewritten to this new temporary.

Algorithm

Input: An interference graph $G = \langle V, E \rangle$ and a coloring col : $V \to K$.
Input: The instruction sequence that produced G.
For every instruction `%b = copy %a;`:
    If col(%a) = col(%b): delete the instruction
    Else if %a $\notin$ next(%b) and $\exists c \in K$ such that $c \notin \{\text{col}(u) : u \in \text{next}(\%a) \cup \text{next}(\%b)\}$:
        Create a new temporary %c with color c
        Connect %c to all the neighbors of %a and %b in G
        Remove %a and %b from G
        Replace %a and %b with %c in the instruction sequence.

5 GENERATING X64

You will also need to implement the ETAC →X64 pass of your compiler. In order to implement this, you will first have to destruct SSA form using the SSA destruction algorithm presented in the lecture. Note that you may assume that your SSA is *conventional*, i.e., that in every `phi`-function the temporaries in the arguments are versions of the same root as the result temporary. This will not be the case if you implement

global copy propagation, so for this project you should turn off that dataflow optimization. (You may still perform DSE and control flow optimizations.) (As a strech goal, you can implement the SSA destruction algorithm that works on non-conventional SSA, as seen during the lecture)

When compiling ETAC to x64, you should make use of the allocation record. This will require you to handle many more cases of the conversion of ETAC instructions to x64 instructions, but most of the cases will be straightforward. As usual, use `R11` as your book-keeping register, if needed; this will mean that you cannot use it as a color.

Notes

- Some instructions such as `mul`, `div`, `mod`, `shl`, and `shr` make use of particular hardware registers. Therefore, you should add these hardware register temporaries into the use/def sets for these instructions during liveness computations. For instance, the `shl` instruction should have `%%rcx` in both its use and def sets.

6 DELIVERABLES

At the end of this project you should produce a program called `bxc.py` that would convert a BX source file specified in the command line, say `prog.bx`, into x64 assembly, `prog.s`. You may choose to emit the intermediate `prog.tac` if you wish (can be controlled with command line options).