Code Generation for Data Processing

Lecture 10: Register Allocation

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2025/26

Register Allocation

- Question: are there enough registers for all values?
  - *Register pressure* = number of values live at some point
  - Register pressure $>$ #registers $\Rightarrow$ move some values to stack (spilling)
- Question: when spilling, which values and where to store/reload?
  - Spilling is expensive, so avoid spilling frequently used values
- Question: for unspilled values, which register to assign?
  - Also: respect register constraints, etc.

Register Allocation Strategies

Scan-based

- Iterate over the program
- Decide locally what to do
- Greedily assign registers

\+ Fast, good for straight code
– Code quality often bad
- Used for `-O0` and JIT comp.

- Traditionally done on non-SSA form

Graph-based

- Compute *interference graph*
  - Nodes are values
  - Edge $\Rightarrow$ live ranges overlap
- Holistic approach

\+ Often generate good code
– Expensive, superlinear run-time
- Used for optimized code

Linear Scan Register Allocation[67]

Read sections 3 and 4 of the paper.

1. How does the algorithm utilize liveness information?
2. How does the selected order of blocks impact the algorithm?
3. How are values selected for spilling?
4. When are values reloaded from the stack?
5. Which issues of register allocation are not discussed?

[67] M Poletto and V Sarkar. "Linear scan register allocation". In: *TOPLAS* 21.5 (1999), pp. 895–913.

Linear Scan Register Allocation

+ low compile-time, simple
− *very* suboptimal code, live intervals grossly over-approximated

- What's missing?
  - Registers to load spilled values
  - Shuffling of values between blocks
  - Register constraints (e.g., for instructions or function calls)
- Other disadvantage: once a value is spilled, it is spilled everywhere
  - Some other approaches based on lifetime splitting[68]
- Function calls: clobber lots of registers

[68] O Traub, G Holloway, and MD Smith. "Quality and speed in linear-scan register allocation". In: *SIGPLAN* 33.5 (1998), pp. 142–151.

Scan-based Register Allocation[71]

Iterate over basic blocks[69]

- Start with register assignment from predecessor
  - Multiple predecessors: choose assignment from any one
- Iterate over instructions top-down
  - Ensure all instruction operands are in registers
    - When out of registers: move any value to stack
  - For operands in their last use: mark register as free
  - Assign instruction result to new free register
- Shuffle values back into registers where successor expects them[70]

[69] Typically: reverse post-order, so most predecessors are seen before successors, except for loops.

[70] Without critical edges, only relevant for blocks with one successor — others are visited afterwards by RPO definition.

[71] Mostly following Go: `https://github.com/golang/go/blob/5f7abe/src/cmd/compile/internal/ssa/regalloc.go`

Scan-based Register Allocation – Spilling

What to spill?

- Spill value with furthest use in future[72]
  - Frees register for longest time
  - Requires information on next use to be stored during analysis
  - But: avoid spilling values computed inside loops (esp. loop-carried dependencies), reloads are fine[73]
  - Downside: superlinear run-time

Where to store?

- Stack
- Spilling to FP/vector registers. . . occasionally proposed, not used in practice

[72] C Wimmer and H Mössenböck. "Optimized interval splitting in a linear scan register allocator". In: *VEE*. 2005, pp. 132–141.

[73] Intel Optimization Reference Manual (Aug. 2023), Assembly/Compiler Coding Rules 38 and 45

Scan-based Register Allocation – Spilling

Where to insert store?

- Option 1: spill exactly where required
  - Downside: multiple spills of same value, many reloads
- Option 2: spill once, immediately after computation
  - Later "spills" to the stack are less costly
  - May lead to spills on code paths that don't need it
- Option 3: compute best place using dominator tree
  - Spill store must dominate all subsequent loads

Scan-based Register Allocation – Register Assignment

- Merge blocks: choose predecessor with most values in registers
  - High likelihood of reducing the number of stores
  - Re-loads are pushed into predecessors
- Propagate register constraints bottom-up as hints first
  - E.g.: call parameters, instruction constraints, assignment for merge block
  - Reduces number of moves

Graph Coloring: Overview

- Analyze values that are live at the same time
- Construct *interference graph*
  - Nodes: values; edge $(a, b) \Rightarrow a$ and b have overlapping live ranges
- Idea: Find k-coloring of the graph
  - Each color corresponds to one register
- $\mathcal{NP}$-complete problem for $k > 2$

Chaitin's Algorithm[74]

- Find node with $< k$ edges
  - Such nodes can always be colored
  - Store node onto a stack
  - Remove node and its edges from interference graph
- If nodes remain: must have $\geq k$ edges
  - Spill value to stack $\rightsquigarrow$ new loads before uses
  - Update interference graph, restart
- Pop nodes from stack, assign color not used by neighbor

[74] GJ Chaitin. "Register allocation & spilling via graph coloring". In: *SIGPLAN* 17.6 (1982), pp. 98–101.

Chaitin's Algorithm: Coalescing

- Avoid reg–reg moves by assigning them the same color
  - E.g., moves resulting from register constraints or two-address instructions
- Idea: *coalescing* of graph nodes
  - Merge move-related nodes that don't interfere
- Redo liveness analysis and rebuild interference graph
  - Might result in fewer interferences, more coalescing

\+ Avoids moves

– Can make colorable graph uncolorable $\rightsquigarrow$ more spilling

Chaitin's Algorithm: Register Constraints

- Add pre-colored nodes in interference graph: one per register
  - These nodes interfere with each other
- Nodes that require a specific register interfere with all other pre-colored nodes
- Typically insert moves into constrained reg right before use

Graph Coloring: More Approaches

- Chaitin's algorithm can fail to color colorable graphs
- Several other approaches

- Interference graph on SSA is chordal
- Permits polynomial time coloring of graph
- SSA destruction done after assigning registers

- Spilling/coalescing must be done separately before coloring
- Optimal spilling remains $\mathcal{NP}$-complete
- After spill code insertion, SSA form needs to be reconstructed

- Pre-colored nodes make problem $\mathcal{NP}$-complete again
- Approaches: split into sub problems, repairing

Stack Frame Allocation

- Optionally setup frame pointer
  - Required for variably-sized stack frame
    Otherwise: cannot access spilled variables or stack parameters
- Optionally re-align stack pointer
- Save callee-saved registers, maybe also link register
- Optionally add code for stack canary
- Compute stack frame size and adjust stack pointer
  - Mainly size of `alloca`s, but needs to respect alignment
  - Ensure sufficient space for parameters passed on the stack
  - Ensure stack pointer is sufficiently aligned
- Stack pointer adjustment *may* be omitted for leaf functions
  - Some ABIs guarantee a *red zone*

Block Ordering

- Order blocks to make use of fall-through in machine code
- Avoid sequences of `b.cond; b`
  - Sometimes cannot be avoided: conditional branches often have shorter range
- Block ordering has implications for branch prediction
  - Forward branches default to not-taken, backward taken
  - Unlikely blocks placed "out of the way" of the main execution path
  - Indirect branches are predicted as fall-through

Register Allocation – Summary

- Scan-based approaches are fast, but lead to suboptimal code
- Graph coloring yields better results, but is much slower
- Coalescing important to reduce moves, esp. after SSA destruction
- SSA-based register allocation impractical
- Register allocation/spilling heavily relies on heuristics in practice

Register Allocation – Questions

- Why is register allocation a difficult problem?
- What are practical constraints for register allocation?
- What is the idea of linear scan and what are its practical problems?
- How to construct an interference graph?