Code Generation for Data Processing
Lecture 7: Register Allocation

Alexis Engelke

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

Winter 2022/23
Register Allocation

- Map unlimited/virtual registers to limited/architectural registers
- Assign a register to every value
  - Outputs get a (new) register, input operands often require registers
- When running out of registers, move values to stack
  - Stack spilling – save value register from to stack memory
- $\phi$-nodes: ensure all inputs are assigned to same location
- Goal: produce correct code, minimize extra load/stores
  - Regalloc affects performance in orders of magnitude
Register Allocation: Overview Example

\[
\text{gauss}(%0) \{ \\
\quad \%2 = \text{SUBXri} \%0, 1 \\
\quad \%3 = \text{MADDXrrr} \%0, \%2, 0 \\
\quad \%4 = \text{MOVXconst} 2 \\
\quad \%5 = \text{SDIVrr} \%3, \%4 \\
\quad \text{ret} \%5 \\
\}
\]

\[
\text{gauss}(%0 : X0) \{ \\
\quad \%2 = \text{SUBXri} \%0, 1 : X \\
\quad \%3 = \text{MADDXrrr} \%0, \%2, 0 : X \\
\quad \%4 = \text{MOVXconst} 2 : X \\
\quad \%5 = \text{SDIVrr} \%3, \%4 : X \\
\quad \text{ret} \%5 \\
\}
\]

- May also insert copy and stack spilling instructions
Simplest thing that could possibly work

- Idea: allocate a one stack slot for every SSA variable/argument
- Load all instruction operands into registers right before
- Perform instruction
- Write result back to stack slot for that SSA variable

+ Simple, always works, debugging easy
- *Extremely* inefficient in time and space
Regalloc Example 1

gauss(%0)
   %2 = SUBXri %0, 1
   %3 = MADDXrrr %0, %2, 0
   %4 = MOVXconst 2
   %5 = SDIVrr %3, %4
ret %5

gauss(%0 : X0)
   %spills = alloca 816243240
   STRXi %0, %spills, 0
   %10 = LDRXi %spills, 0 : X0
   %2 = SUBXri %0%10, 1 : X0
   STRXi %2, %spills, 8
   %11 = LDRXi %spills, 0 : X0
   %12 = LDRXi %spills, 8 : X1
   %3 = MADDXrrr %0, %2%11, %12, 0 : X0
   STRXi %3, %spills, 16
   %4 = MOVXconst 2 : X0
   STRXi %4,i %spills, 24
   %13 = LDRXi %spills, 16 : X0
   %14 = LDRXi %spills, 24 : X1
   %5 = SDIVrr %3, %4%13, %14 : X0
   STRXi %5, %spills, 32
   %15 = LDRXi %spills, 32 : X0
ret %5%15
Handling PHI Nodes

- $\phi$-node needs to become register or stack slot
  - Simplest thing that could possibly work: PHI becomes stack slot
- Remember: $\phi$-nodes are executed on the edge

- Idea: predecessors write their value to that location at the end
  - First pass: define/allocate storage for $\phi$-node, but ignore inputs
  - Second pass: insert move operations at end of predecessors
Regalloc Example 2

identity(%0)
   br %2
2:
   %3 = phi [ 0, %1 ], [ %4, %2 ]
   %4 = ADDXri %3, 1
   %5 = CMPXrr_BLS %4, %0
   br %5, %2, %6
6:
   ret %3

Pass 12

▶ Original value lost in %6!
Critical Edges

- Critical edge: edge from block with multi. succs. to block with multi. preds.
- Problem: cannot place move on such edges
  - When placing in predecessor, they would also execute for other successor
    ⇒ unnecessary and – worse – incorrect

- Break critical edges: insert an empty block
Regalloc Example 2 – Attempt 2

identity(%0)
   br %2
2:
   %3 = phi [ 0, %1 ], [ %4, %2 ]
   %4 = ADDXri %3, 1
   %5 = CMPXrr_BLS %4, %0
   br %5, %6, %7
6:
   br %2
7:
   ret %3

Pass 12

identity(%0 : X0)
   %spills = alloca 81624
   STRXi %0, %spills, 0
   %c0 = MOVXconst 0 : X0
   STRXi %c, %spills, 8
   br %2
2:%3 = phi [ 0, %1 ], [ %4, %2 ]
   %l0 = LDRXi %spills, 8 : X0
   %4 = ADDXri %3%l0, 1 : X0
   STRXi %4, %spills, 16
   %l1 = LDRXi %spills, 16 : X0
   %l2 = LDRXi %spills, 0 : X1
   %5 = CMPXrr_BLS %4, %0%l1, %l2
   br %5, %6, %7
6:%l4 = LDRXi %spills, 16 : X0
   STRXi %l4, %spills, 8
   br %2
7:%l3 = LDRXi %spills, 8 : X0
   ret %3
Handling Critical Edges

Breaking Edges

- Insert new block for moves
  + Simple, no analyses needed
  - Bad performance in loops

Copy Used Values

- Move values still used to new reg.
  + Performance might be better
  - Needs more registers
Regalloc Example 3

odd(%0)
   br %2
2:
   %3 = phi [ %0, %1 ], [ %8, %7 ]
   %4 = phi [ 1, %1 ], [ %5, %7 ]
   %5 = phi [ 0, %1 ], [ %4, %7 ]
   %6 = CBNZX(%3)
   br %6, %7, %9
7:
   %8 = SUBXri %3, 1
   br %2
9:
   ret %4

► Value of $\phi$ node lost!
Problem: $\phi$-nodes can depend on each other
Can be chains (ordering matters) or cycles (need to be broken)
Note: only $\phi$-nodes defined in same block are relevant/problematic

\[
\begin{align*}
\phi_1 &= \phi(\phi_2, \ldots) \\
\phi_2 &= \phi(\phi_3, \ldots) \\
\phi_3 &= \phi(\nu, \ldots)
\end{align*}
\]
Handling PHI Cycles

1. Iterate over $\phi$, handle independent, store critical $\phi$
   - Only $\phi$-nodes that read a $\phi$-node in the same block left

2. Compute number of other crit. $\phi$ nodes reading each remaining node

3. For each crit. $\phi$ with 0 readers: handle chain
   - No readers $\leadsto$ start of chain; handling node unblocks next element in chain

4. For all remaining $\phi$-nodes: must be cycles, reader count always 1
   - Choose arbitrary node, load to temporary register, unblock value
   - Handle just-created chain
   - Write temporary register to target

Resolving $\phi$ cycles requires an extra register (or stack slot)
Regalloc Example 3 – Attempt 2

Edge %1 → %2 Edge %7 → %2

Critical φ:

▶ %4 #readers: 10 – broken
▶ %5 #readers: 10

Action: break %4

odd(%0 : X0)
%spills = alloca 40
STRXi %0, %spills, 0
%13 = LDRX%spills, 0 : X0; STRXi %13, %spills, 8
%c0 = MOVX%spills 1 : X0; STRXi %c0, %spills, 16
%c1 = MOVX%spills 0 : X0; STRXi %c1, %spills, 16
br %2

2:%3 = phi [ %0, %1 ], [ %8, %7 ] // spills+8
%4 = phi [ 1, %1 ], [ %5, %7 ] // spills+16
%5 = phi [ 0, %1 ], [ %4, %7 ] // spills+24
%10 = LDRX%spills, 8 : X0
%6 = CBNZX(%10)
br %6, %7, %9

7:%11 = LDRX%spills, 8 : X0
%8 = SUBXi %12, 1 : X0; STRXi %8, %spills, 32
%14 = LDRX%spills, 40 : X0; STRXi %14, %spills, 8
%15 = LDRX%spills, 24 : X1
%16 = LDRX%spills, 16 : X0; STRXi %16, %spills, 24
STRXi %15, %spills, 16
br %2

9:%12 = LDRX%spills, 24 : X0
ret %12
Better Register Allocation

- Goal: keep as many values in registers as possible
  - Less stack spilling $\Rightarrow$ better performance

- Problem: register count (severely) limited
  - Are there enough registers? (otherwise: spilling)
  - Which register to choose?
  - Which register to kill and put on the stack?
Register Allocation: Research

- *Tons* of papers exist
- Papers often skip over important details
  - E.g., when spilling – using the value needs another register
  - E.g., temporary register for shuffling values
- Additional (ISA) constraints in practice: (incomplete list)
  - 2-address instructions with destructive source
  - Fixed registers for specific instructions
  - Computing the stack address may need yet another register
  - Different register classes, often just handled independently

- Implementations even of simple algorithms tend to be large and complex
Liveness Analysis

- **Live**: value still used afterwards
  - After last (possible) use in program flow, the value becomes dead

- **Live ranges**: set of ranges in program where value is live
  - Not necessarily contiguous, e.g. in case of branches

- **Live interval**: over-approximation of live ranges without holes
  - Depends on block order, reverse post-order often a good choice
Liveness Analysis on SSA\textsuperscript{31}

- For each block \textit{liveIn}: values that are needed at block entry
- Construct live ranges for each SSA value

- Iterate over blocks in post-order
  - \( \text{live} \leftarrow \bigcup s.\text{liveIn}, s \in b.\text{successors} \)
  - \( \text{live} \leftarrow \text{live} \cup \{ \phi.\text{input}(b) | \phi \in b.\text{successors}.\text{phis} \} \)
  - \( \forall v \in \text{live} : \text{ranges}[v].\text{add}(b.\text{start}, b.\text{end}) \)
  - For each non-\( \phi \) instruction \textit{inst} in reverse order
    - \( \text{live} \leftarrow (\text{live} \cup \textit{inst}.\text{ops}) \setminus \{\textit{inst}\} \)
    - \( \text{ranges}[\textit{inst}].\text{setStart}(\textit{inst}) \)
    - \( \forall \text{op} \in \textit{inst}.\text{ops} : \text{ranges}[\text{op}].\text{add}(b.\text{start}, \textit{inst}) \)
  - \( b.\text{liveIn} \leftarrow \text{live} \setminus b.\text{phis} \)
- Repeat until convergence

Linear Scan Register Allocation

- Idea: treat whole function as single block
  - Block order affects quality (but not correctness)
  - Only consider live intervals without holes

- Iterate over instructions from top to bottom
- For operands of instruction in their last use: mark register as free
- Assign instruction result to new free register
  - If no free register available: move a value to the stack
  - Heuristic: value whose lifeness ends furthest in future

---

Linear Scan Register Allocation

+ low compile-time, simple, used for JIT-compilers and Go
− very suboptimal code, live intervals grossly over-approximated

▶ What’s missing?
  ▶ Registers to load spilled values and shuffle values
  ▶ Register constraints (e.g., for insts. or function calls)
▶ Other disadvantage: once a value is spilled, it is always spilled
▶ Function calls: clobber lots of registers
Linear Scan – Adaption (Engelke, 2022)

- Run linear scan, but forcefully keep one free register before $\phi$-nodes
  - For register constraints, forcefully evict value occupying the register

- Emit spill code and add new live intervals
  - Spill store: immediately store to stack, adds short live interval
  - Spill loads: load operands to new reg., adds short live intervals

- Repeat until no extra intervals/spills are inserted
Making Linear Scan Non-Linear (and better)

- Don’t spill variable forever, but split life time once necessary\(^{33}\)
  - When no register is free, spill a register, but only from this point on
  - On reload, keep copy in register (but keep stack slot until end)
- Base spill decision on next use (instead of lifetime end)\(^{34}\)
  - Additionally keep track of next use distance during analysis
  - Benefit: better spill decisions; downside: superlinear run-time
- Propagate register preferences bottom-up\(^{35}\)
  - Better assignment for function calls/fixed register operands

---


\(^{35}\) https://github.com/golang/go/blob/5f7abe/src/cmd/compile/internal/ssa/regalloc.go e.g. lines 2604–2636
Graph Coloring: Overview

- Analyze values that are live at the same time
- Construct *interference graph*
  - Nodes: values; edge \((a, b) \Rightarrow a \text{ and } b\) have overlapping live ranges
- Idea: Find \(k\)-coloring of the graph
  - Each color corresponds to one register
- Easy case: all nodes have degree \(\leq k\)
Chaitin’s Algorithm\textsuperscript{36}

- Find node with fewer than $k$ edges
  - If no such node exists: pick one and spill to stack
  - Selection based on heuristics
  - Update interference graph

- Remove it from the graph
- Recursively color the rest of the graph
- Add note back in and assign valid color

\textsuperscript{36}GJ Chaitin. “Register allocation & spilling via graph coloring”. In: SIGPLAN 17.6 (1982), pp. 98–101.
Graph Coloring Approaches

- Considerably better results than greedy algorithms
- High run-time, even with heuristics

> Graph coloring in general is \( \mathcal{NP} \)-complete
> Often used in compilers (e.g., GCC, WebKit)

AD IN2053 “Program Optimization” covers this more formally
Register Selection and Spilling

- Avoid spilling values in loops
- Avoid spilling values used immediately afterwards
- Prefer callee-saved register for values live across function calls
  - Function call clobbers caller-saved regs $\leadsto$ cheaper call
- Spill slots can be reused for different values
  - Better use of stack, but higher complexity
- Spilling to FP/vector registers...
  - Occasionally proposed, rarely done in practice
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

- Map unlimited virtual registers to restricted register set
- Responsible for:
  - Assigning registers to values
  - Deciding which registers to spill to stack
  - Deciding when to spill/unspill values
- $\phi$-nodes require extra care, esp. for chains and cycles
- Liveness information is key information for register allocation
- Linear-time algorithms exist, but have suboptimal results
- Register allocation/spilling relies on heuristics in practice
Register Allocation – Questions

- Why is register allocation a difficult problem?
- How are $\phi$-nodes handled during register allocation?
- What are the two main problems when destructing $\phi$-nodes?
- Why are critical edges problematic and how to deal with them?
- What are practical constraints for register allocation?
- How to detect whether a value is still needed at some point?
- What is the idea of linear scan and what are its practical problems?