Code Generation for Data Processing

Lecture 9: SSA Destruction and Liveness Analysis

Alexis Engelke

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

Winter 2025/26

Register Allocation: Overview

- Destruct SSA form = resolve ϕ-nodes
- Map unlimited/virtual registers to limited/architectural registers
- When running out of registers, move values to stack

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

```
gauss(X0) {
  X1 = SUBXri X0, 1
  X0 = MADDXrrr X0, X1, XZR
  X1 = MOVXconst 2
  X0 = SDIVrr X0, X1
  ret X0
}
```

Register Allocation: Strategy Overview

- "Conventional:" ϕ-node elimination before register allocation
  - ϕ-nodes replaced with sequences of copies
  - RegAlloc input is no longer in SSA form
  - $\rightsquigarrow$ RegAlloc is a $\mathcal{NP}$-hard problem
  - Widely used in practice

- "More modern:" register allocation on SSA form
  - ϕ-nodes eliminated after RegAlloc
  - $\rightsquigarrow$ RegAlloc becomes easier, but remains $\mathcal{NP}$-hard in practice
  - Mainly academic research

SSA Destruction

- Goal: eliminate ϕ-nodes
- For now, continue to assume unlimited registers

- Remember: ϕ-nodes are executed on the edge

- Idea: predecessors write their value to that location at the end

SSA Destruction: Example 1

```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %loop ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %loop, %ret
ret:
  ret %3
```

```
identity(%0)
e:
  %3 = copy 0
  br %loop
loop:
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  %3 = copy %4
  br %5, %loop, %ret
ret:
  ret %3
```

- Original value lost in return block!

Lost Copy Problem

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

- *Break* critical edges: insert an empty block

SSA Destruction: Example 1 – Critical Edge Split

```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %critedge ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %critedge, %ret
critedge:
  br %loop
ret:
  ret %3
```

```
identity(%0)
e:
  %3 = copy 0
  br %loop
loop:
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %critedge, %ret
critedge:
  %3 = copy %4
  br %loop
ret:
  ret %3
```

- Problem fixed, but one extra branch per loop iteration

Handling Critical Edges

Breaking Edges

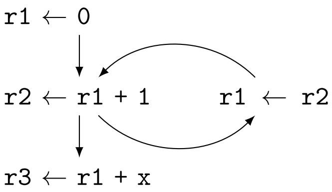
- Insert new block for moves

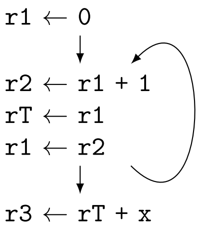
\+ Simple, no analyses needed
– Bad performance in loops

Copy Used Values

- Move used values to new reg.

\+ Performance might be better
– Needs more registers





SSA Destruction: Example 1 – Value Copy

```
identity(%0)
e:
  br %loop
loop:
  %3 = phi [ 0, %e ], [ %4, %loop ]
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  br %5, %loop, %ret
ret:
  ret %3
```

```
identity(%0)
e:
  %3 = copy 0
  br %loop
loop:
  %4 = add %3, 1
  %5 = cmp ult %4, %0
  %3c = copy %3
  %3 = copy %4
  br %5, %loop, %ret
ret:
  ret %3c
```

- Problem fixed; register allocator can hopefully omit a copy

SSA Destruction: Example 2

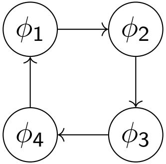
```
odd(%0)
e:
  br %loop
loop:
  %3 = phi [ %0, %e ], [ %8, %body ]
  %4 = phi [ 1, %e ], [ %5, %body ]
  %5 = phi [ 0, %e ], [ %4, %body ]
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  br %loop
ret:
  ret %4
```

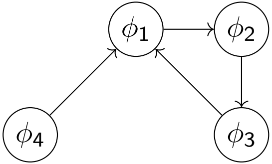
```
odd(%0)
e:
  %3 = copy %0
  %4 = copy 1
  %5 = copy 0
  br %loop
loop:
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3 = copy %8
  %4 = copy %5
  %5 = copy %4
  br %loop
ret:
  ret %4
```

- Value of ϕ (%5) is wrong!

Swap Problem / ϕ Dependencies

- Problem: ϕ-nodes can depend on each other
- Can be chains (ordering matters) or cycles (need to be broken)
- Note: only ϕ-nodes defined in same block are relevant/problematic





$$\begin{aligned} \phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(v, \dots) \end{aligned}$$

$$\begin{aligned} \phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(\phi_4, \dots) \\ \phi_4 &= \phi(\phi_1, \dots) \end{aligned}$$

$$\begin{aligned} \phi_1 &= \phi(\phi_2, \dots) \\ \phi_2 &= \phi(\phi_3, \dots) \\ \phi_3 &= \phi(\phi_1, \dots) \\ \phi_4 &= \phi(\phi_1, \dots) \end{aligned}$$

Handling PHI Cycles

1. Compute number of other ϕ-nodes reading other ϕ on same edge
2. For each ϕ with 0 readers: handle node/chain
   - No readers $\rightsquigarrow$ start of chain
   - Handling node may unblock next element in chain
3. For all remaining ϕ-nodes: must be cycles, reader count always 1
   - For trivial cycles (single node), do nothing
   - Choose arbitrary node, load to temporary register, unblock value
   - Handle just-created chain
   - Write temporary register to target

$\rightsquigarrow$ Resolving ϕ cycles requires an extra register (or stack slot)

SSA Destruction: Example 2 – Fixed

```
odd(%0)
e:
  br %loop
loop:
  %3 = phi [ %0, %e ], [ %8, %body ]
  %4 = phi [ 1, %e ], [ %5, %body ]
  %5 = phi [ 0, %e ], [ %4, %body ]
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  br %loop
ret:
  ret %4
```

```
odd(%0)
e:
  %3 = copy %0
  %4 = copy 1
  %5 = copy 0
  br %loop
loop:
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3 = copy %8
  %4t = copy %4
  %4 = copy %5
  %5 = copy %4t
  br %loop
ret:
  ret %4
```

- ϕ-cycle on edge body→`loop` broken with temporary register

SSA Destruction: Exercise

```
fn(%0, %1) {
b1:
  %2 = add %0, %1
  br %b2
b2:
  %3 = phi [%1, %b1], [%4, %b3]
  %4 = phi [%0, %b1], [%3, %b3]
  %5 = phi [%2, %b1], [%3, %b3]
  %6 = phi [0, %b1], [%8, %b3]
  %7 = icmp lt %3, %6
  br %7, %b3, %b4
b3:
  %8 = add %6, 1
  %9 = icmp gt %8, %1
  br %9, %b4, %b2
b4:
  %10 = phi [%4, %b2], [%3, %b3]
  %11 = phi [%5, %b2], [%8, %b3]
  %12 = add %10, %11
  ret %12
}
```

1. Dependencies between ϕ-nodes?
2. Critical Edges? (Draw CFG)
3. Destruct SSA into form with unlimited registers.
   3.1 ... by breaking critical edges
   3.2 ... by copying used values

SSA Destruction: Alternative Approach

- Other approach: generate a lot of copies, clean up later

```
odd(%0)
e:
  br %loop
loop:
  %3 = phi [ %0, %e ], [ %8, %body ]
  %4 = phi [ 1, %e ], [ %5, %body ]
  %5 = phi [ 0, %e ], [ %4, %body ]
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  br %loop
ret:
  ret %4
```

```
odd(%0)
e:
  %3in = copy %0
  %4in = copy 1
  %5in = copy 0
  br %loop
loop:
  %3 = copy %3in
  %4 = copy %4in
  %5 = copy %5in
  %6 = cmp ne %3, 0
  br %6, %body, %ret
body:
  %8 = sub %3, 1
  %3in = copy %8
  %4in = copy %5
  %5in = copy %4
  br %loop
ret:
  ret %4
```

Register Allocation: Simple Approaches

- Simplest thing that could possibly work:
  allocate a one stack slot for every (SSA) variable/argument
  - Reload all operands immediately before the instruction
  - Store to stack slot after computation

+ Simple, always works, debugging easy
– Extremely slow: values are stored and immediately reloaded
– Extremely inefficient memory usage: huge stack frames

Register Allocation: Simple Approaches

- Next simplest thing that could possibly work:
  avoid reload if value is still in a register
  - When assigning target register: choose any register
  - Across basic blocks, spill everything

\+ Still simple, always works, debugging easy

– Very slow: unnecessary stores, loop variants in memory

– Extremely inefficient memory usage: huge stack frames

Register Allocation: Problems of Simple Approaches

- Many avoidable spills
  - Spill of last use of a value can/should be omitted
  - Stack slots for unused variables should be reused (why?)
- 2-address instructions (destructive source) require value copy
- Bad eviction decisions: value might be used immediately afterwards
- Loop-variants should kept be in registers if possible

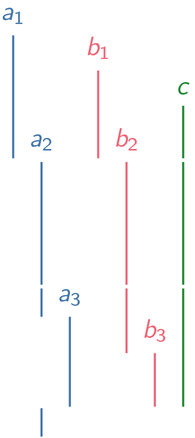
$\rightsquigarrow$ Need information about what variables need to be preserved

Liveness: Definitions

- *Live:* value might be used by later operation
  - 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-in*/*Live-out*: values live at begin/end of basic block/instruction
  - For ϕ nodes: ϕ is live-in in block, operands are live-out in predecessors

  (Note: different literature uses different definitions)

Liveness: Example



	$a_1 \leftarrow \ldots$	liveIn=$\emptyset$
	$b_1 \leftarrow 0$	liveOut=$\{a_1, b_1, c\}$
	$c \leftarrow \ldots$	
	`br loop`	
`loop:`	$a_2 \leftarrow \phi(a_1, a_3)$	liveIn=$\{a_2, b_2, c\}$
	$b_2 \leftarrow \phi(b_1, b_3)$	liveOut=$\{a_2, b_2, c\}$
	`br` $b_2 < c$`, body, ret`	
`body:`	$a_3 \leftarrow a_2 + b_2$	liveIn=$\{a_2, b_2, c\}$
	$b_3 \leftarrow b_2 + 1$	liveOut=$\{a_3, b_3, c\}$
	`br body`	
`ret:`	`ret` a_2	liveIn=$\{a_2\}$
		liveOut=$\emptyset$

Liveness: Definitions[57]

- $\mathit{Defs}(B)$: values defined in B (no defs from ϕ)
- $\mathit{Uses}(B)$: values used in B (no uses in ϕ)
- $\mathit{UpwardExposed}(B)$: values used in B before a definition in B
  - SSA: this is $\mathit{Uses}(B) \setminus (\mathit{Defs}(B) \cup \mathit{PhiDefs}(B))$
- $\mathit{PhiDefs}(B)$: values defined by ϕ at entry of B
- $\mathit{PhiUses}(B)$: values used in ϕs in successors of B

$$\mathit{LiveOut}(B) = \mathit{PhiUses}(B) \cup \bigcup_{S \in \mathit{succs}(B)} (\mathit{LiveIn}(S) \setminus \mathit{PhiDefs}(S))$$

$$\mathit{LiveIn}(B) = \mathit{PhiDefs}(B) \cup \mathit{UpwardExposed}(B) \cup (\mathit{LiveOut}(B) \setminus \mathit{Defs}(B))$$

[57] B Boissinot and F Rastello. "Liveness". In: *SSA-based Compiler Design*. Ed. by F Rastello and F Bouchez Tichadou. 2022, pp. 107–122. DOI: 10.1007/978-3-030-80515-9_9.

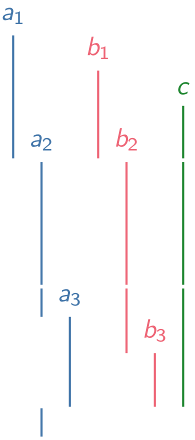
Liveness: Iterative Data Flow Algorithm[58]

- Precompute per-basic block sets
  - Requires backwards pass over instructions of basic block
- Initialize *LiveIn*/*LiveOut* to empty sets
- Iteratively recompute *LiveIn*/*LiveOut* until convergence

- Block order has strong impact on runtime; post-order preferable
- Exact live ranges can be tracked
  - LiveOut $\Rightarrow$ live range extends to end of block; otherwise ends at last use inside block

[58] GA Kildall. "A unified approach to global program optimization". In: *POPL*. 1973, pp. 194–206.

Liveness: Iterative Algorithm Example



$a_1 \leftarrow \dots$
$b_1 \leftarrow 0$
$c \leftarrow \dots$
$\texttt{br loop}$

ue$=\emptyset$
liveIn$=\emptyset$
liveOut$=\{a_1, b_1, c\}$

$\texttt{loop:}$ $a_2 \leftarrow \phi(a_1, a_3)$
$b_2 \leftarrow \phi(b_1, b_3)$
$\texttt{br } b_2 < c\texttt{, body, ret}$

ue$=\{c\}$
liveIn$=\{a_2, b_2, c\}$
liveOut$=\{a_2, b_2, c\}$

$\texttt{body:}$ $a_3 \leftarrow a_2 + b_2$
$b_3 \leftarrow b_2 + 1$
$\texttt{br body}$

ue$=\{a_2, b_2\}$
liveIn$=\{a_2, b_2, c\}$
liveOut$=\{a_3, b_3, c\}$

$\texttt{ret:}$ $\texttt{ret } a_2$

ue$=\{a_2\}$
liveIn$=\{a_2\}$
liveOut$=\emptyset$

Liveness: Exercise

Compute *liveIn*, *liveOut* for all blocks and live ranges for values.

$$
\begin{aligned}
b1:\quad & a = f(0) \\
& \texttt{br}\ \ b1 \\
b2:\quad & b = f(a) \\
& \texttt{br}\ \ b3 \\
b3:\quad & \texttt{br}\ \ b > 0, b2, b4 \\
b4:\quad & c = phi(b, e) \\
& d = f(c) \\
& \texttt{br}\ \ c > 0, b3, b5 \\
b5:\quad & e = f(d) \\
& \texttt{br}\ \ e > 0, b4, b6 \\
b6:\quad & \texttt{ret}\ \ e
\end{aligned}
$$

Liveness: Iterative Algorithm Analysis

- Convergence: must converge, liveness sets grow monotonically
- Max. iterations: $d(G) + 3$[59]
  - $d(G)$ is depth = max. number of backedges on cycle-free path in G

- Variation: worklist instead of round-robin[60]

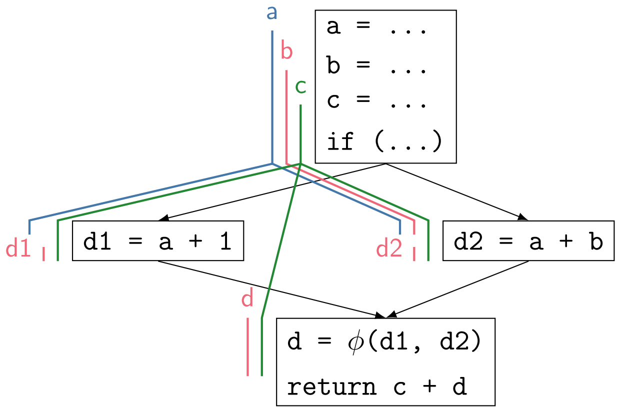
[59] JB Kam and JD Ullman. "Global data flow analysis and iterative algorithms". In: *JACM* 23.1 (1976), pp. 158–171.

[60] KD Cooper, TJ Harvey, and K Kennedy. "An empirical study of iterative data-flow analysis". In: *CIC*. 2006, pp. 266–276

Liveness on SSA

- SSA values have one definition; uses must be dominated by that definition

$\Rightarrow$ Live ranges are sub-trees of dominator tree



Liveness: Two-Pass Algorithm on SSA[61]

- Pass 1: Compute live-in/live-out in post-order, ignoring backedges
  - This is the first iteration of the iterative algorithm
- Pass 2: Extend live-in of loop headers to entire loops
  - Intuition: loop is SCC, so values live in header must be live in entire loop
  - Traverse loop forest in DFS
  - Add $\mathit{LiveIn}(\mathit{Hdr}) \setminus \mathit{PhiDefs}(\mathit{Hdr})$ to LiveIn and $\mathit{LiveOut}$ of loop blocks

- Complexity: $\mathcal{O}(|E| \cdot \#\mathit{vars} + \#\mathit{instrs})$
- Limitation: only works for reducible graphs

[61] B Boissinot et al. "A non-iterative data-flow algorithm for computing liveness sets in strict SSA programs". In: *ASPLAS*. 2011, pp. 137–154.

Liveness: Two-Pass Algorithm on SSA for Irreducible CFG[62]

- Two-pass algorithm can be adapted for irreducible CFGs
- Adjust side entry to loop $s \rightarrow t$ to outermost loop containing t excluding s
  - Side entry: entry not going through header block (remember: header is ambiguous)
- Adjusted CFG doesn't need to be materialized, different CFG traversal

- No change in asymptotic complexity

[62] B Boissinot and F Rastello. "Liveness". In: *SSA-based Compiler Design*. Ed. by F Rastello and F Bouchez Tichadou. 2022, pp. 107–122. DOI: 10.1007/978-3-030-80515-9_9.

Liveness: Over-Approximation

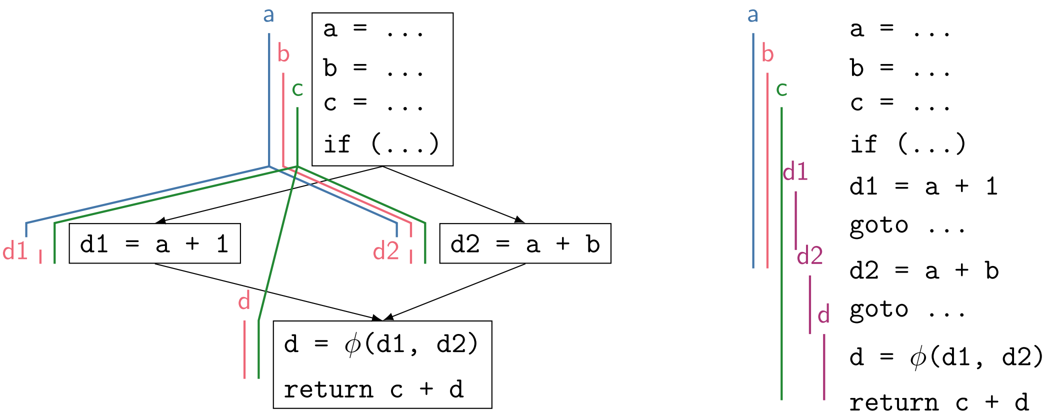
- Precise liveness analysis is somewhat expensive for large functions
- Also requires super-linear amount of memory to store
- For JIT compilation, not always feasible

$\rightsquigarrow$ Over-approximation of liveness information

- Typical approach: single live interval per value[63]
  - Define fixed block order (typically RPO or similar)
  - Store begin and end location of live interval
  - Value might be marked as live in between even if it actually isn't

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

Liveness: Live Ranges vs. Single Live Interval



- Single live interval can be substantially worse

Live Interval Computation

- Can be done in single pass over program[64]
- Also can omit computation of live-in/live-out sets[65]

- Conceptually no large difference to two-pass algorithm
  except: growing live ranges simply adjusts end of interval

[64] C Wimmer and M Franz. "Linear scan register allocation on SSA form". In: *CGO*. 2010, pp. 170–179.

[65] A Kohn, V Leis, and T Neumann. "Adaptive execution of compiled queries". In: *ICDE*. 2018, pp. 197–208.

Liveness Through Path Exploration[66]

- Alternative approach: trace uses back to their definition
- For every use:
  - If defined in the current block, stop (defined)
  - If in live-in of current block, stop (already propagated)
  - Add value to live-in
  - If value is a ϕ-node, stop
  - Add value to live-out of all predecessors
  - Recursively continue in all predecessors

- Complexity: $\mathcal{O}(|E| \cdot \#vars + \#instrs)$
- Used in LLVM

[66] F Rastello. "On Sparse Intermediate Representations: Some Structural Properties and Applications to Just-In-Time Compilation". Habilitation thesis. Inria Grenoble Rhône-Alpes, 2012.

SSA Destruction and Liveness Analysis – Summary

- SSA form must be destructed before machine code generation
- ϕ-nodes executed concurrently on edges
  - Critical edges need special handling: splitting or value copying
  - ϕ-nodes can have dependencies on each other
- Good register allocation needs liveness analysis
- Iterative data flow algorithm can need many iterations
- SSA form permits two-pass liveness analysis
- For faster analysis, liveness can be over-approximated

SSA Destruction and Liveness Analysis – Questions

- What are the two main problems when destructing ϕ-nodes?
- Why are critical edges problematic and how to deal with them?
- Which dependencies between ϕ-nodes can occur? How to handle?
- Why is liveness information critical for reasonable code quality?
- When is a single-use SSA value live after its use?
- Why does SSA form make liveness analysis easier?
- How to compute the live ranges of values on SSA form?
- What are two benefits of storing just a single live interval?