LLVM Register Allocation

Evan Cheng
Apple Inc.
August 1, 2008

LLVM Register Allocation

- Motivation
- Overview
- Optimizations
- Future Work

Isn’t It Done?

- Code generator does a reasonable job
  - LLVM code generator has proven to be quite capable
  - Roughly ~5% better than GCC 4.2 on x86 SPEC
  - About the same as GCC on x86-64
  - Even better on codecs

- But...

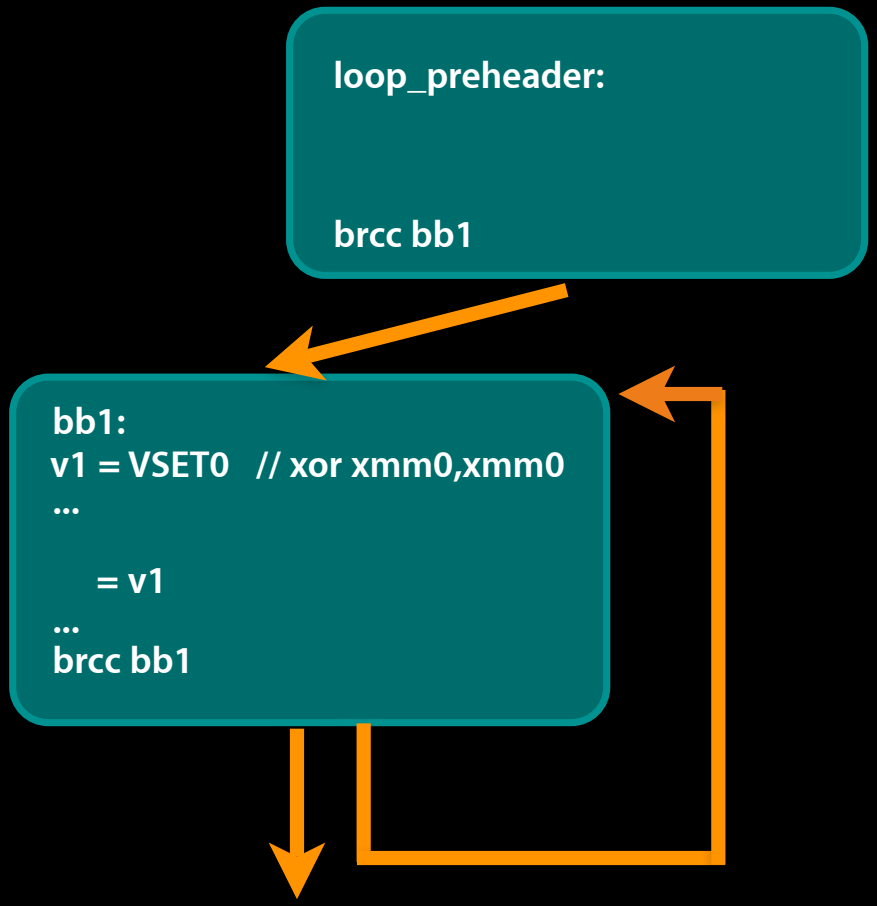
Really, Why Do We Care?

- Squeeze out that last few percentages of performance
- Fix the pathological cases
- Improve compile time for JIT and static codegen
- **Enable more aggressive optimizations**

LLVM Design Philosophy

- Each optimization pass should be as aggressive as possible
- Later passes must do *the right thing* to avoid pessimization
- Earlier optimization passes may increase register pressure
- Register allocation must be able to deal with the increased register pressure

Example: Machine LICM

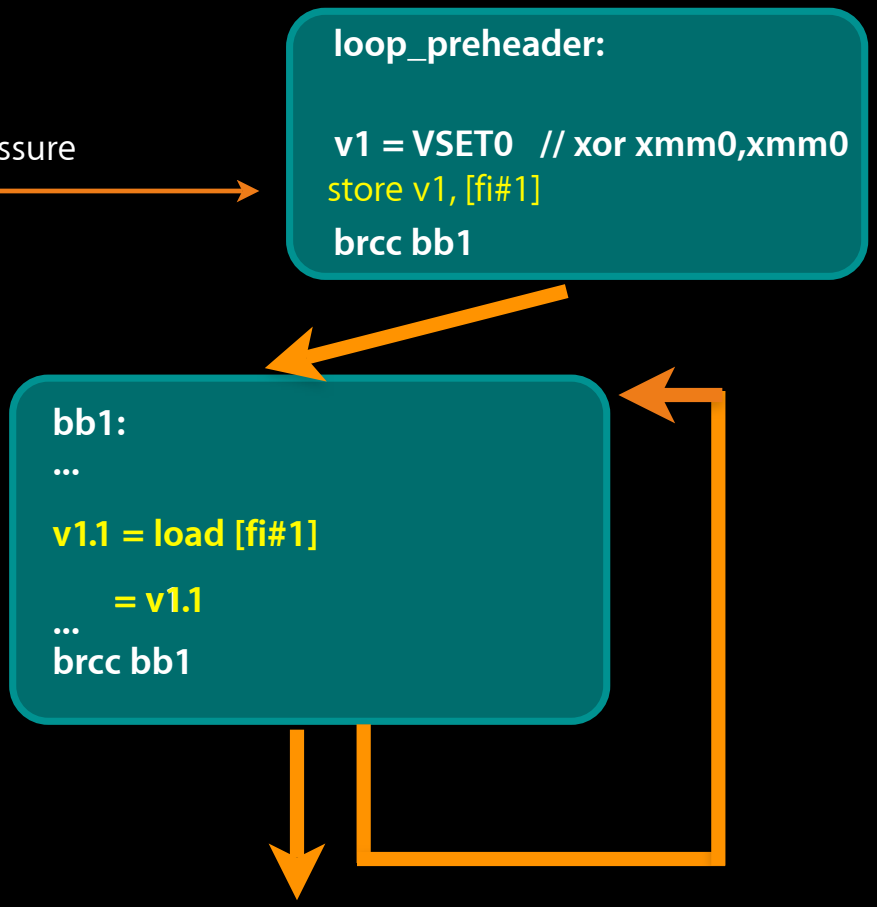


This must be good, right?

Example: Machine LICM cont.

•Not necessarily!

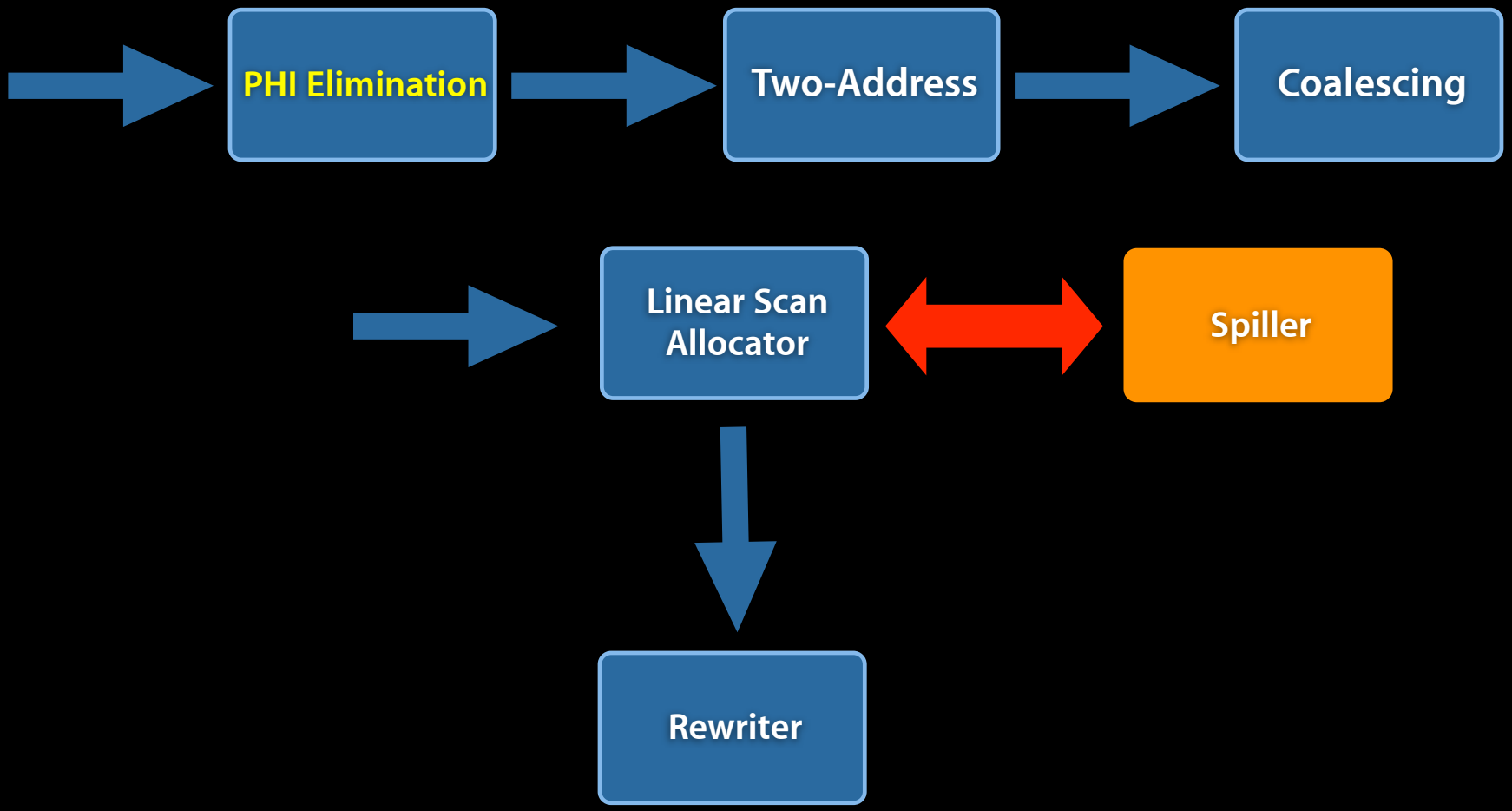
It increases register pressure
so v1 may be spilled



LLVM Register Allocation

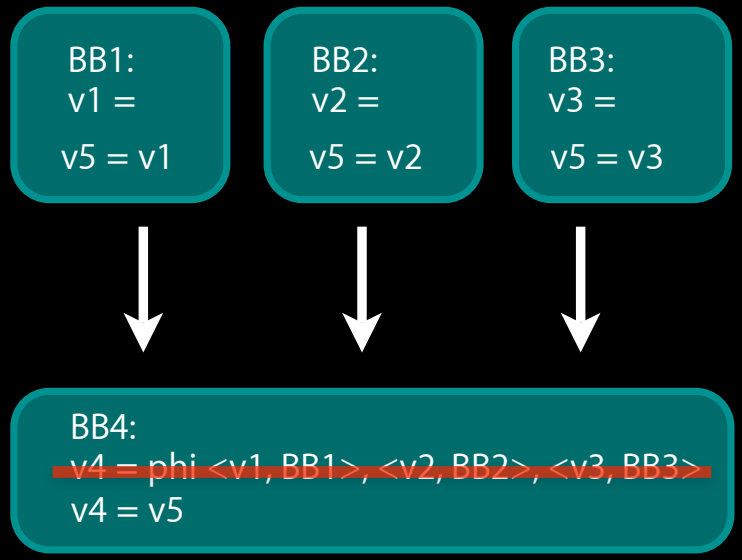
- Motivation
- Overview
- Optimizations
- Future Work

Design of the Register Allocator



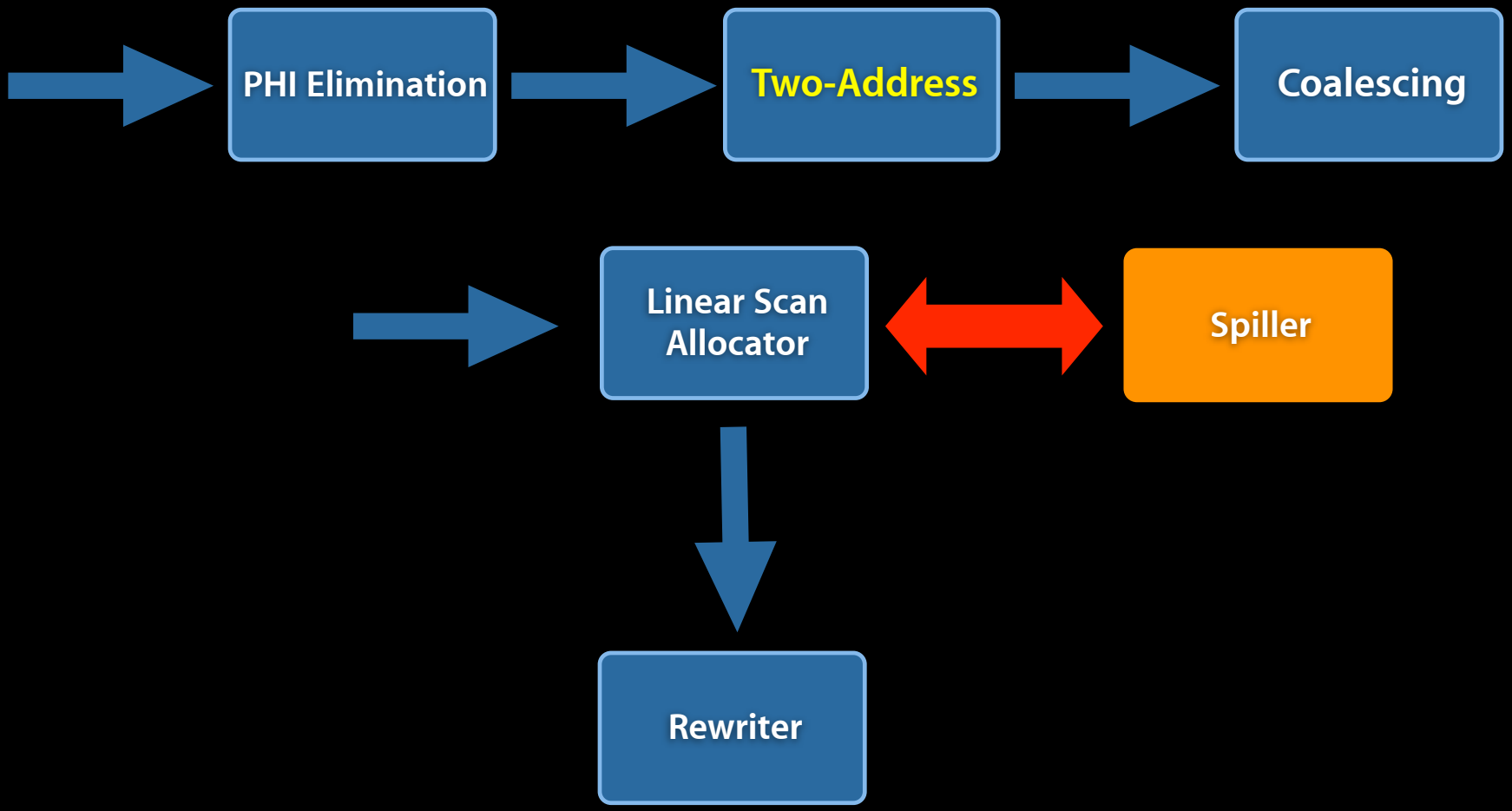
PHI Elimination

Move code out of SSA form and eliminate PHI instructions



Problem: Introduce lots of copies for the coalescer

Design of the Register Allocator

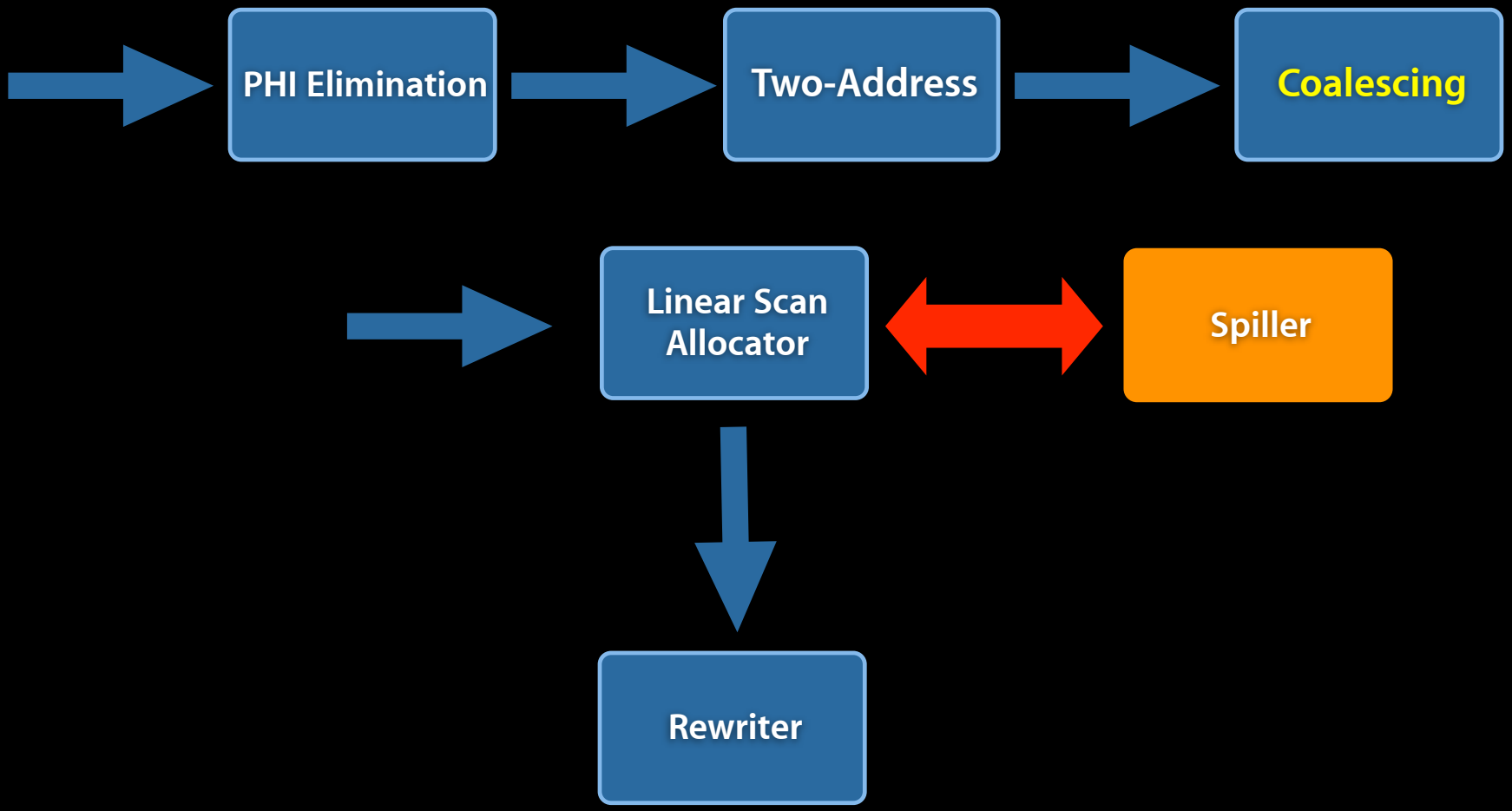


Two-Address Pass

Convert SSA 3-address instructions into instructions with read-modify-write operands

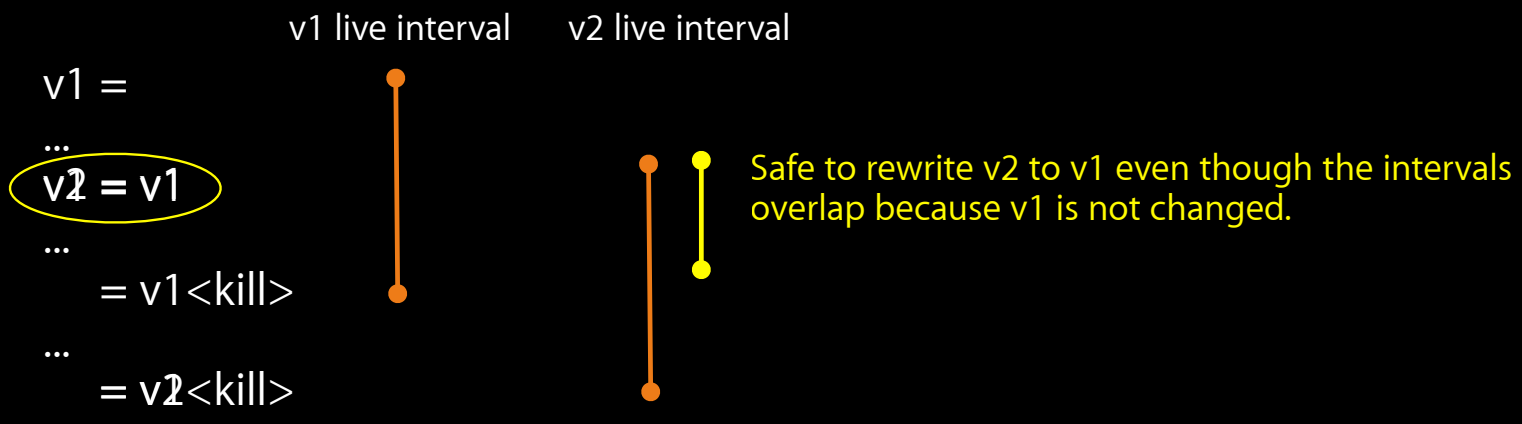
```
v1 = v2
v1 = add  v1, v3
```

Design of the Register Allocator



Register Coalescing

Eliminate copies by registers renaming

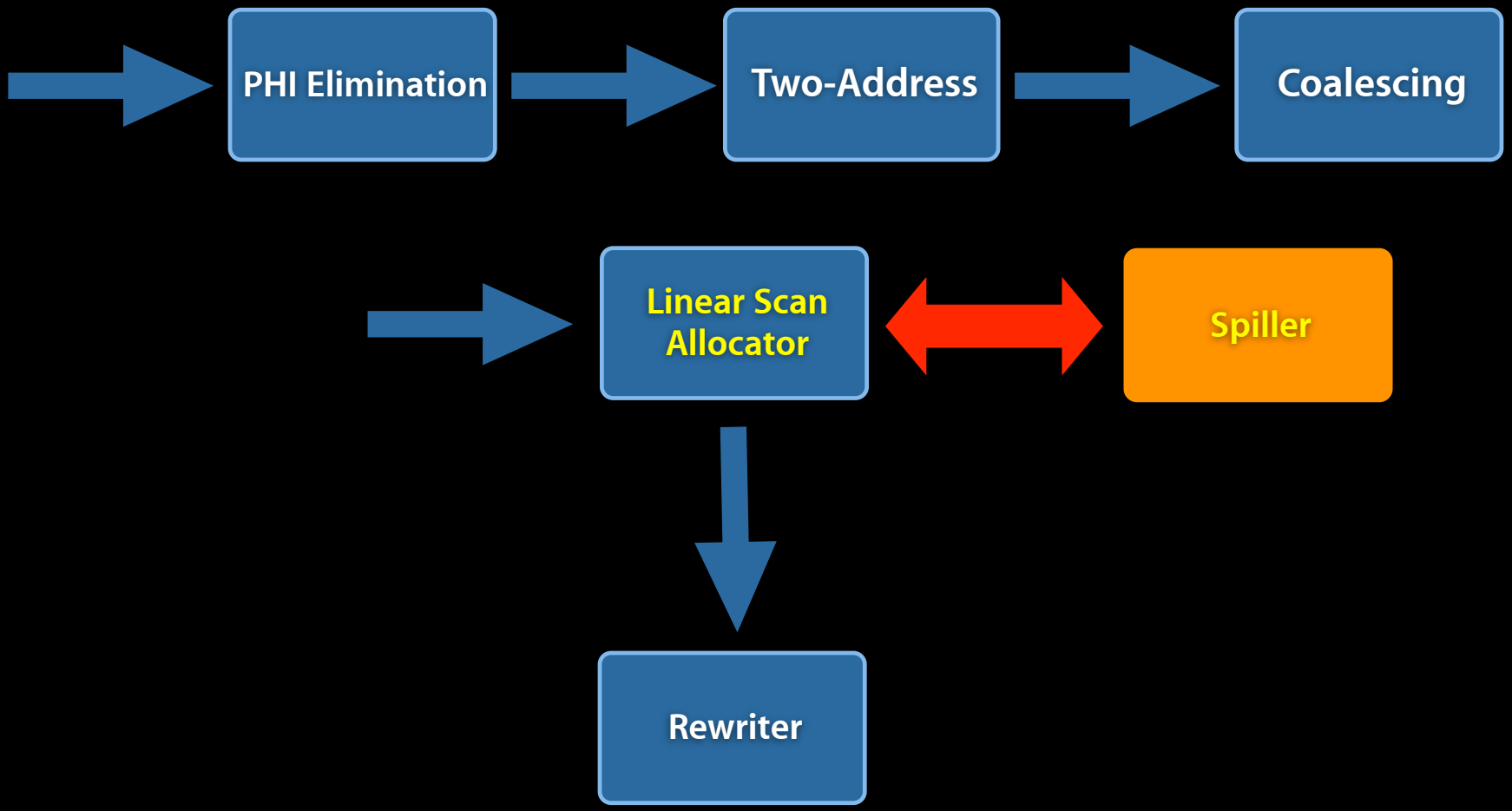


- Implementation is very aggressive:
  - Does value numbering to coalesce live ranges that "conflict"

Why coalesce aggressively?

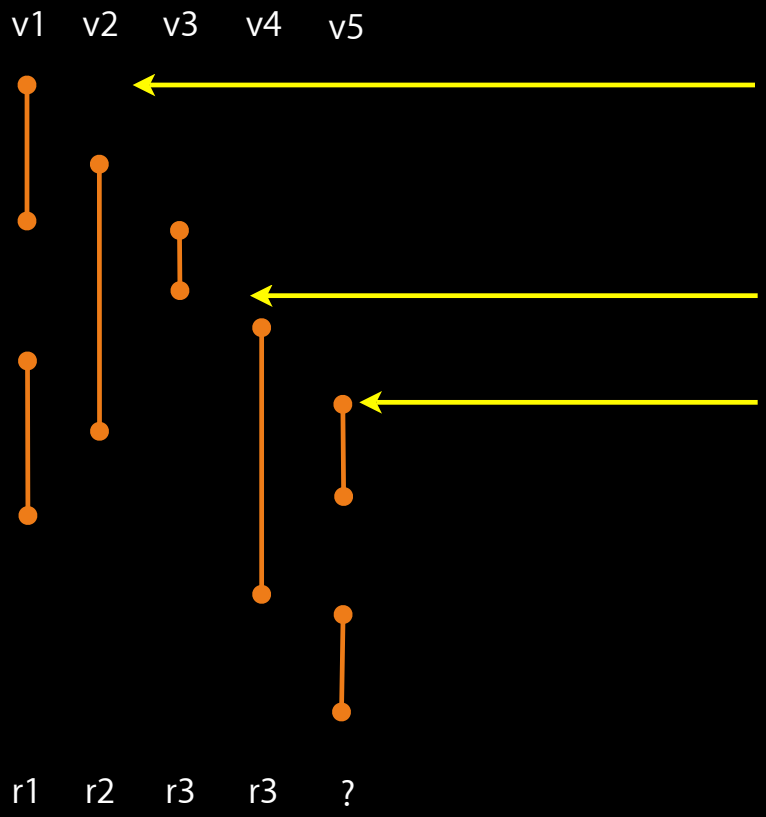
- Coalescing expects allocator to split later if needed
- Don't trust random decisions from input:
  - Copies coming in are from PHI elimination
- Can be useful places to split to reduce register pressure, but:
  - cannot be trusted, miss many important cases
  - coalescing happens before we know true register pressure

Design of the Register Allocator



Linear Scan Register Allocator

Single pass over list of variable live intervals ordered by starting points



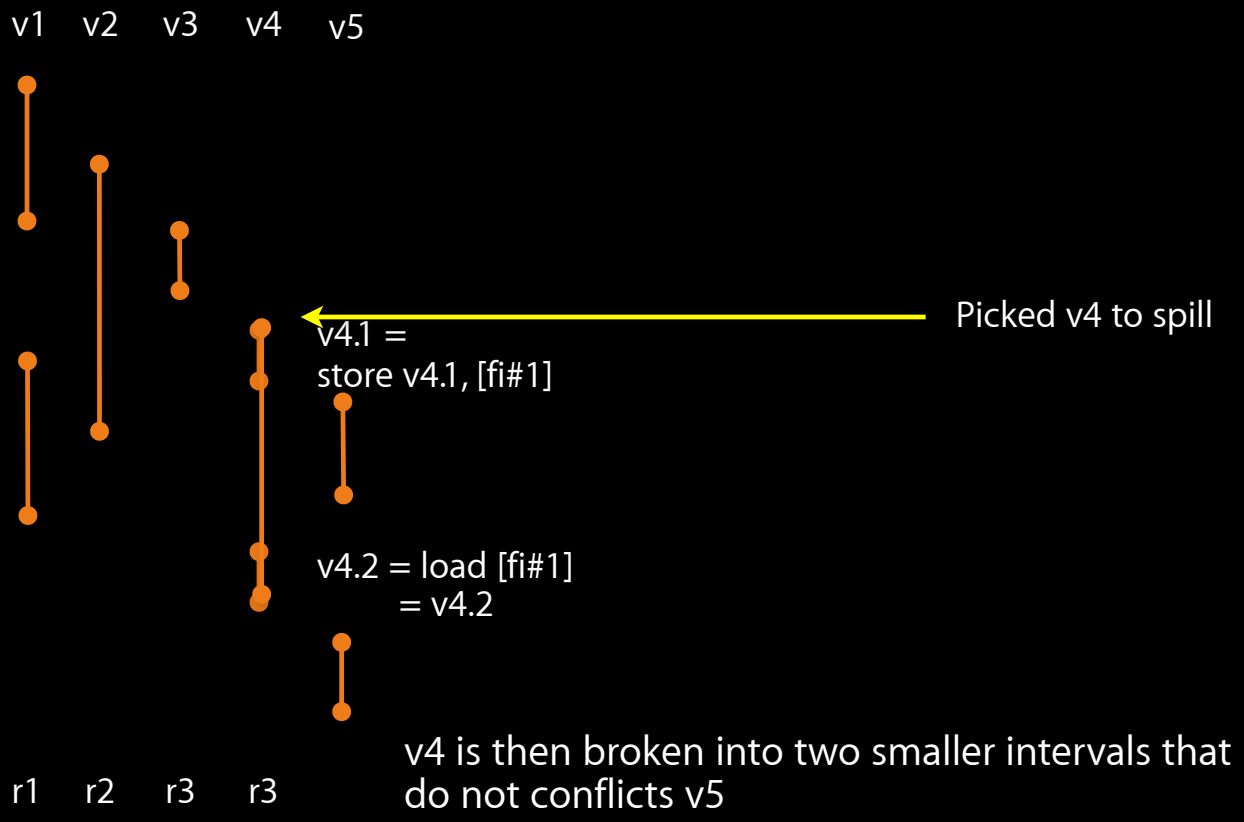
Starting from v1, followed by v2, v3, v4, and then v5

v3 live interval ends here, r3 is now available

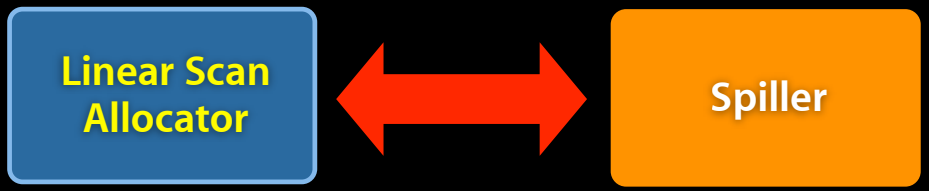
v5 conflicts with v1, v2, and v4, spilling is required

Linear Scan Register Allocator cont.

- Picking spill candidate based on def / use "density"
- Backtrack to the starting point of the spilled live interval

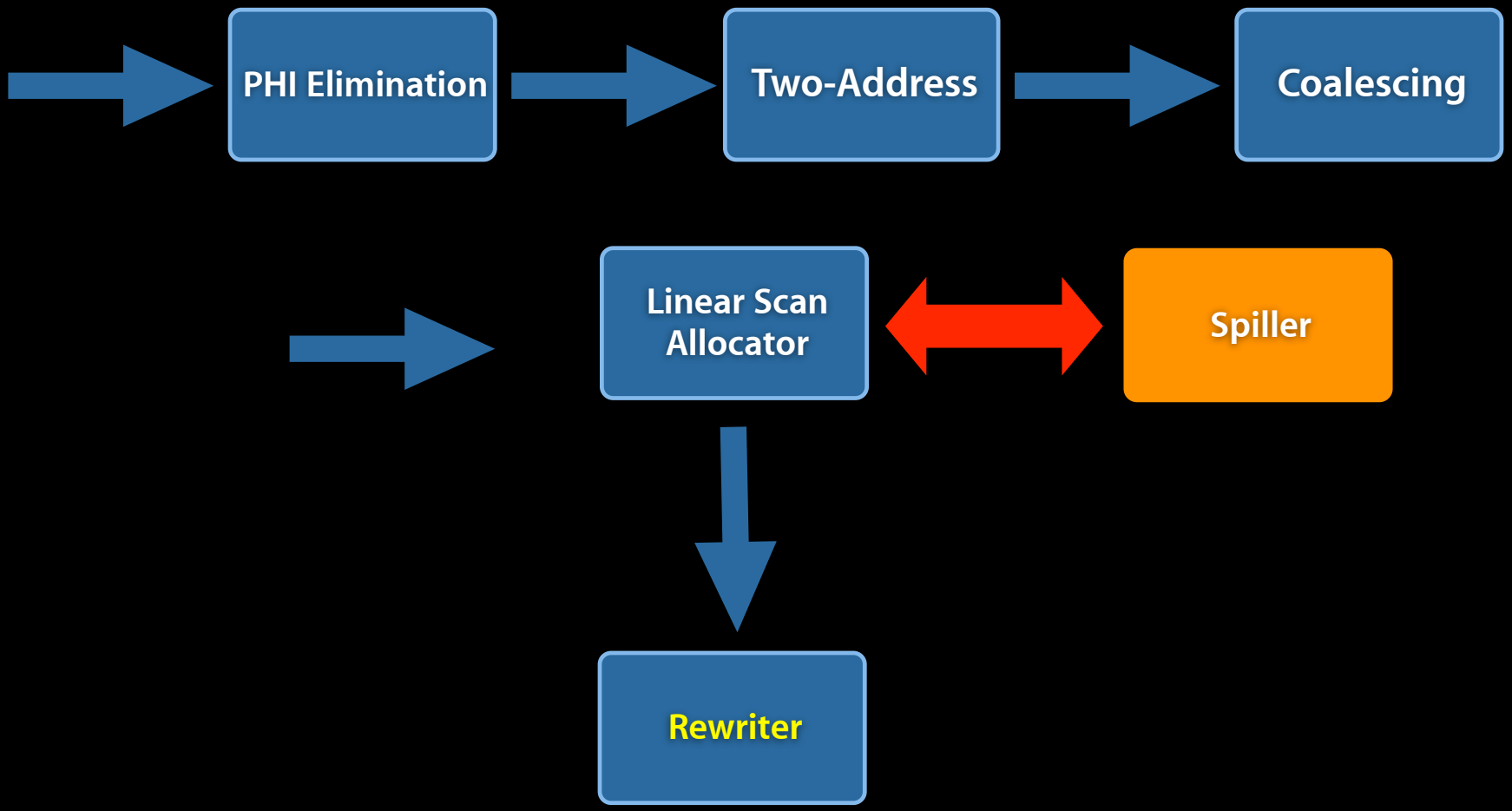


Linear Scan Register Allocator cont.



- Spiller and allocator share responsibilities:
  - Linear scan decides: which live interval to spill
  - Spiller decides: how the interval is spilled
- Major problem:
  - Spill code insertion is deferred until all of allocation is done
  - Major bookkeeping nightmare

Design of the Register Allocator

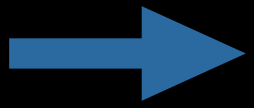


Rewriter

- Rewrite virtual registers to allocated physical registers
- Insert spill and reload code
- Also perform some micro-optimizations

```
Allocations:
v1 -> EBX
v2 -> EAX
v3 -> EAX
v4 -> FI#4, ECX
...

v2 = addri v1, 17
v3 = mulrr v4, vr2
...
```



```
EAX = addri EBX, 17
ECX = load [FI#4]
EAX = mulrr ECX, EAX
...
```

LLVM Register Allocation

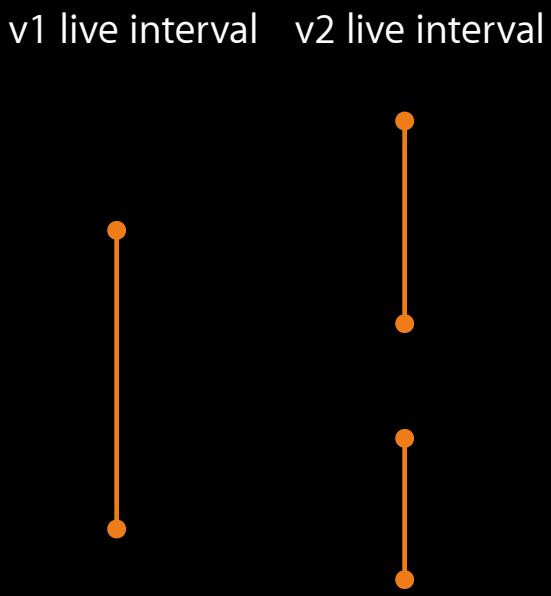
- Motivation
- Overview
- Optimizations
- Future Work

Coalescing: Instruction Commuting

```
bb:
// v2 is livein
v1 = op
...
v2 = add v2, v2<kill>
...
v2 = v2
...
   = v2<kill>
```

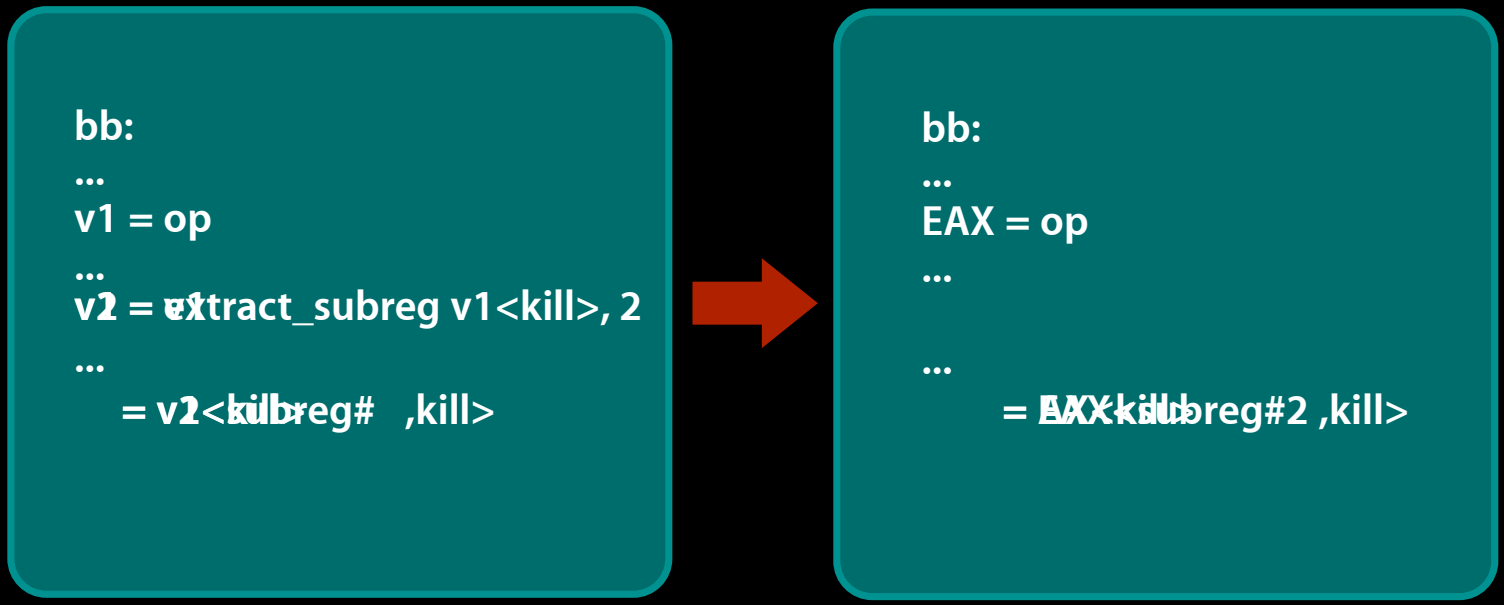
"add" is commutable

Forward substitute

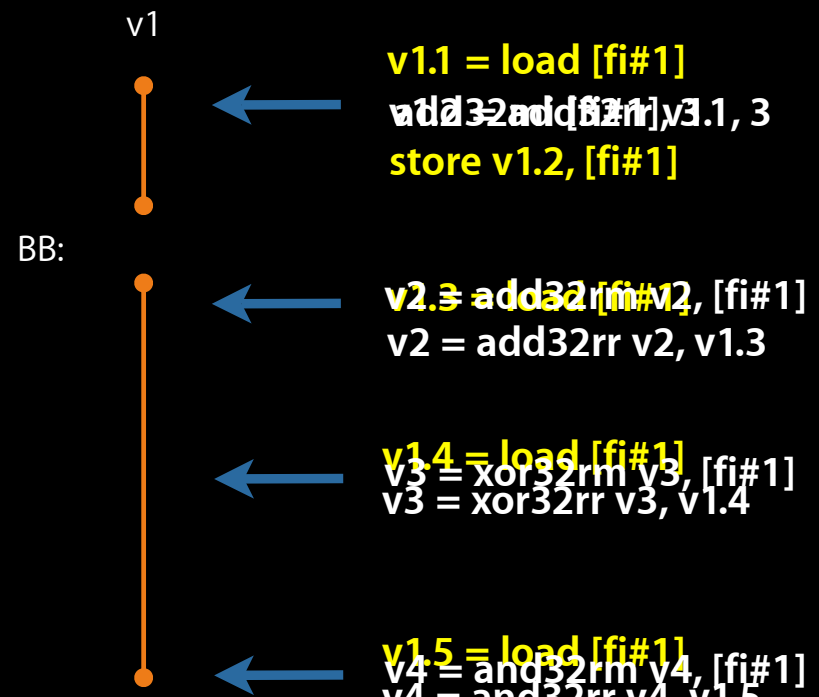


Coalescing: Sub-registers

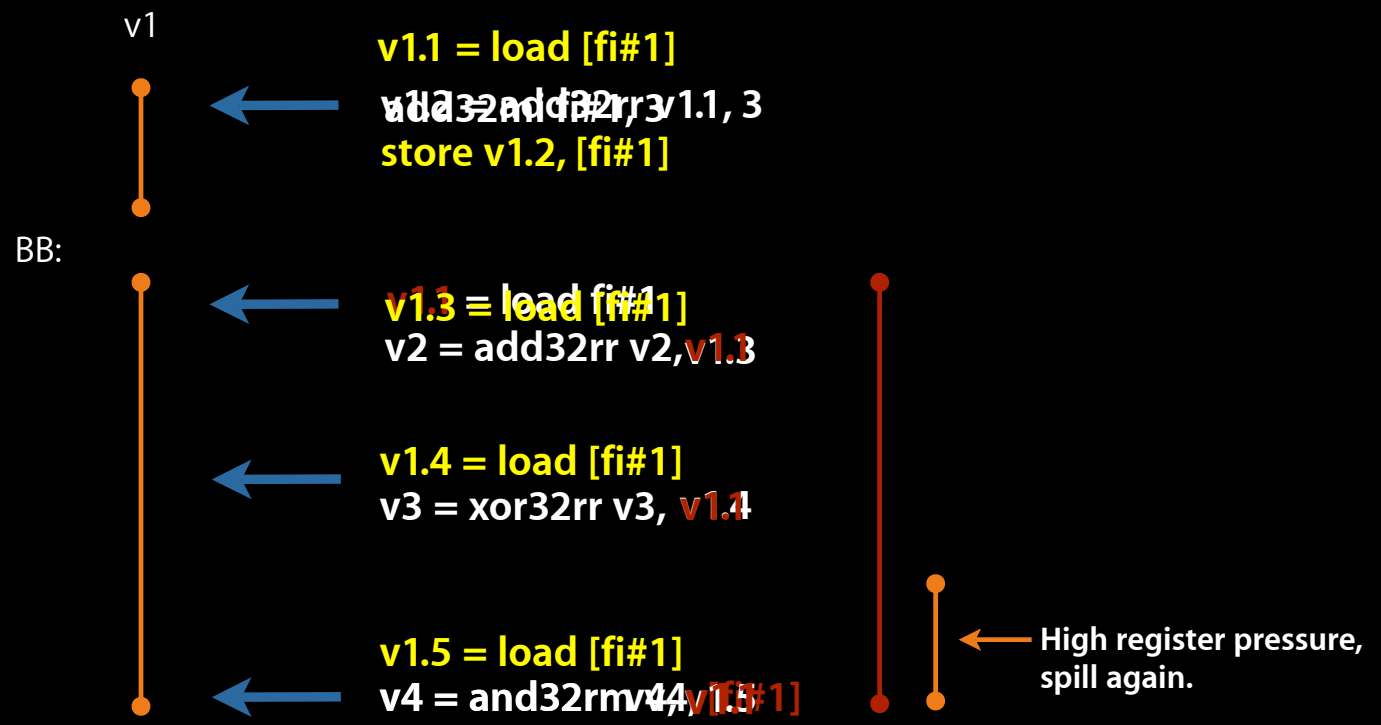
- Eliminate pseudo instructions to "extract" part of a register
- Critical for targets such as X86 which has registers that are part of larger registers
  - e.g. AL, AH are sub-registers of AX; AX is a sub-register of EAX



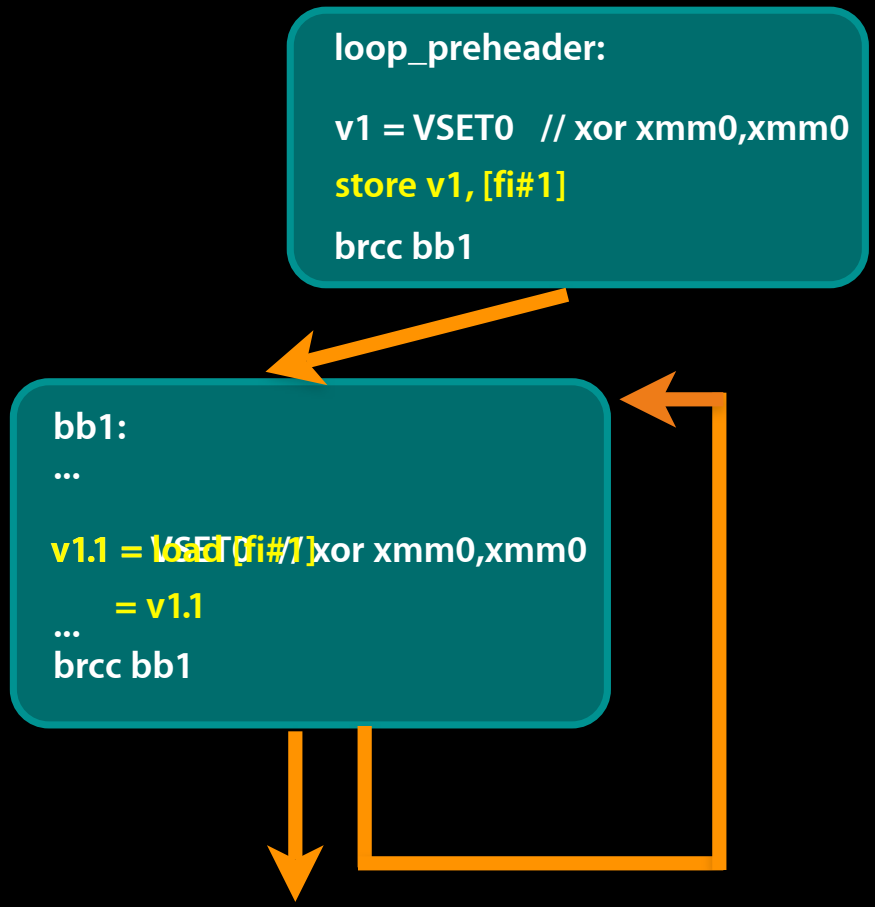
Spilling: Fold Spills and Reloads



Spilling: Splitting at BB Boundaries

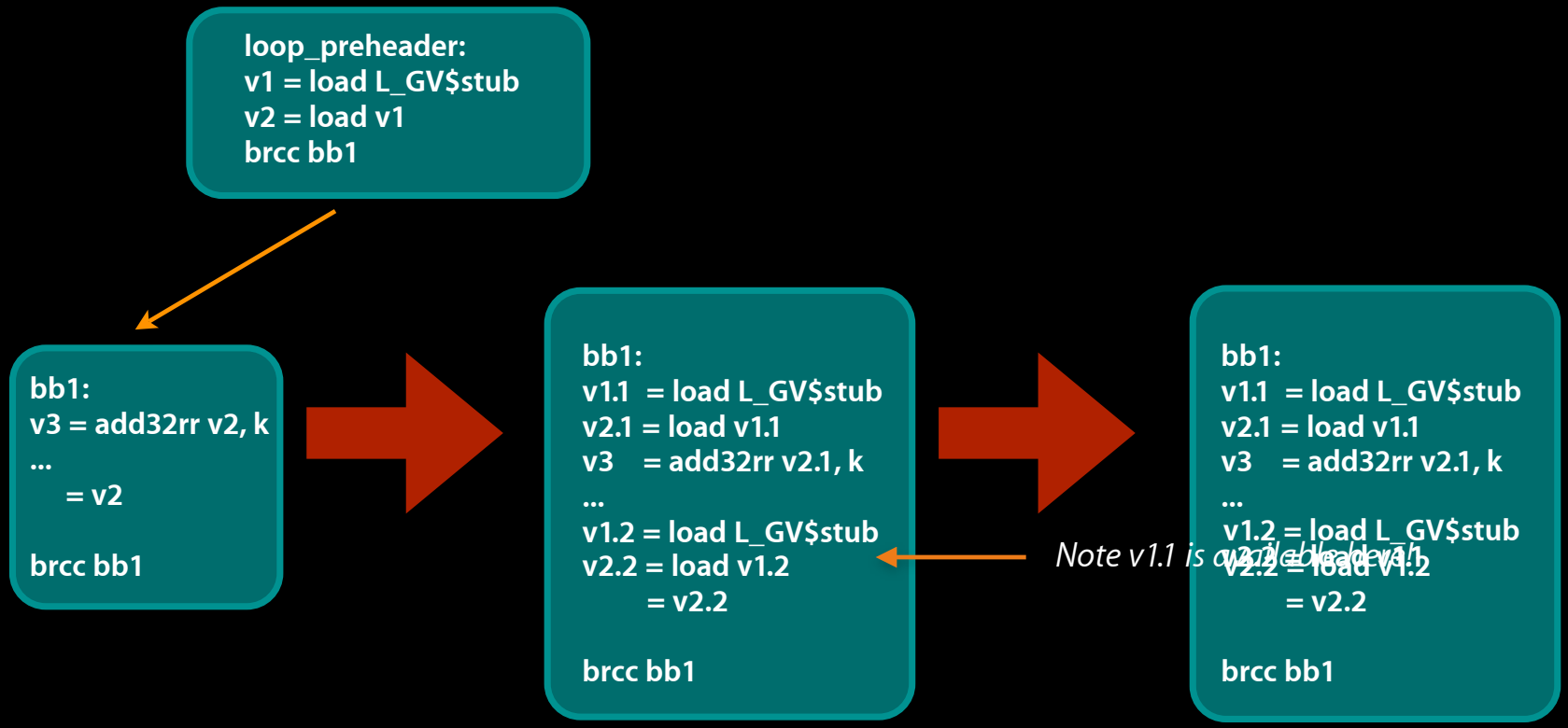


Machine LICM example visited: Simple Re-materialization



- Currently only re-materialize instructions with no register operands
- Hacked to allow PIC base register operands

Generalized Re-materialization



- Need alias analysis information for load motion
- Must track available values / register

LLVM Register Allocation

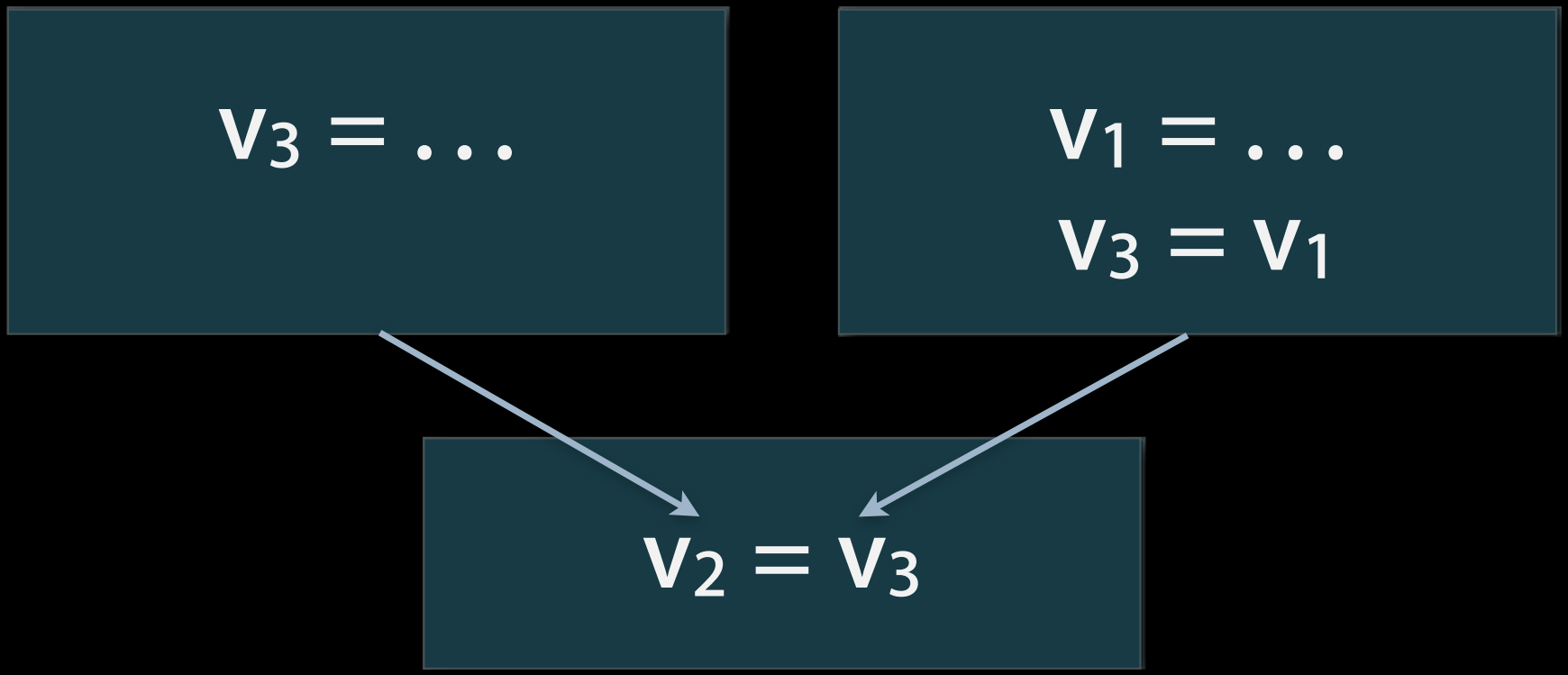
- Motivation
- Overview
- Optimizations
- Future Work

Goals

- Faster compile times
- Generate faster code
- More maintainable and flexible code generator

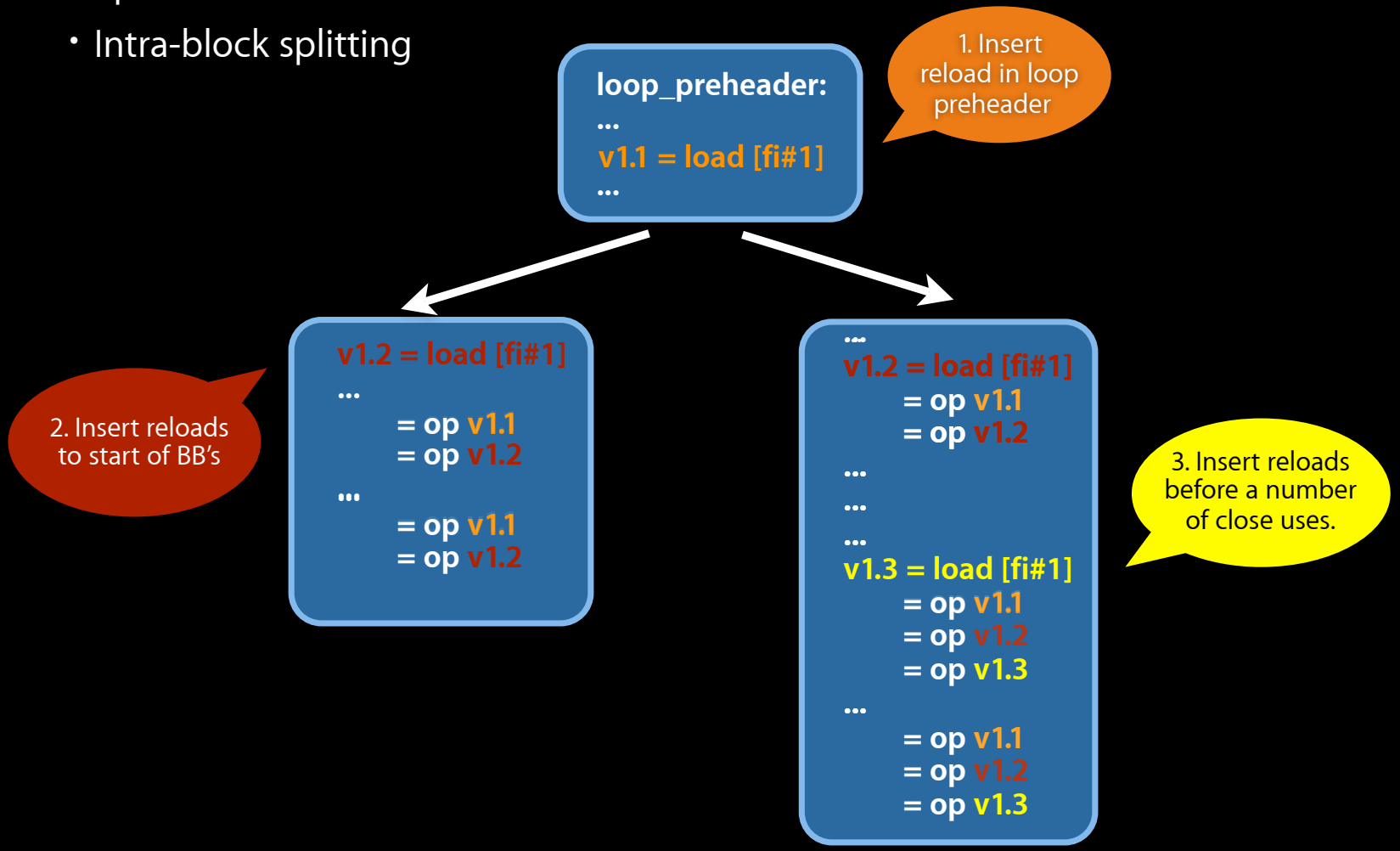
Strong PHI Elimination

- Perform PHI elimination less naively
- Less work for the coalescer, compile-time benefit

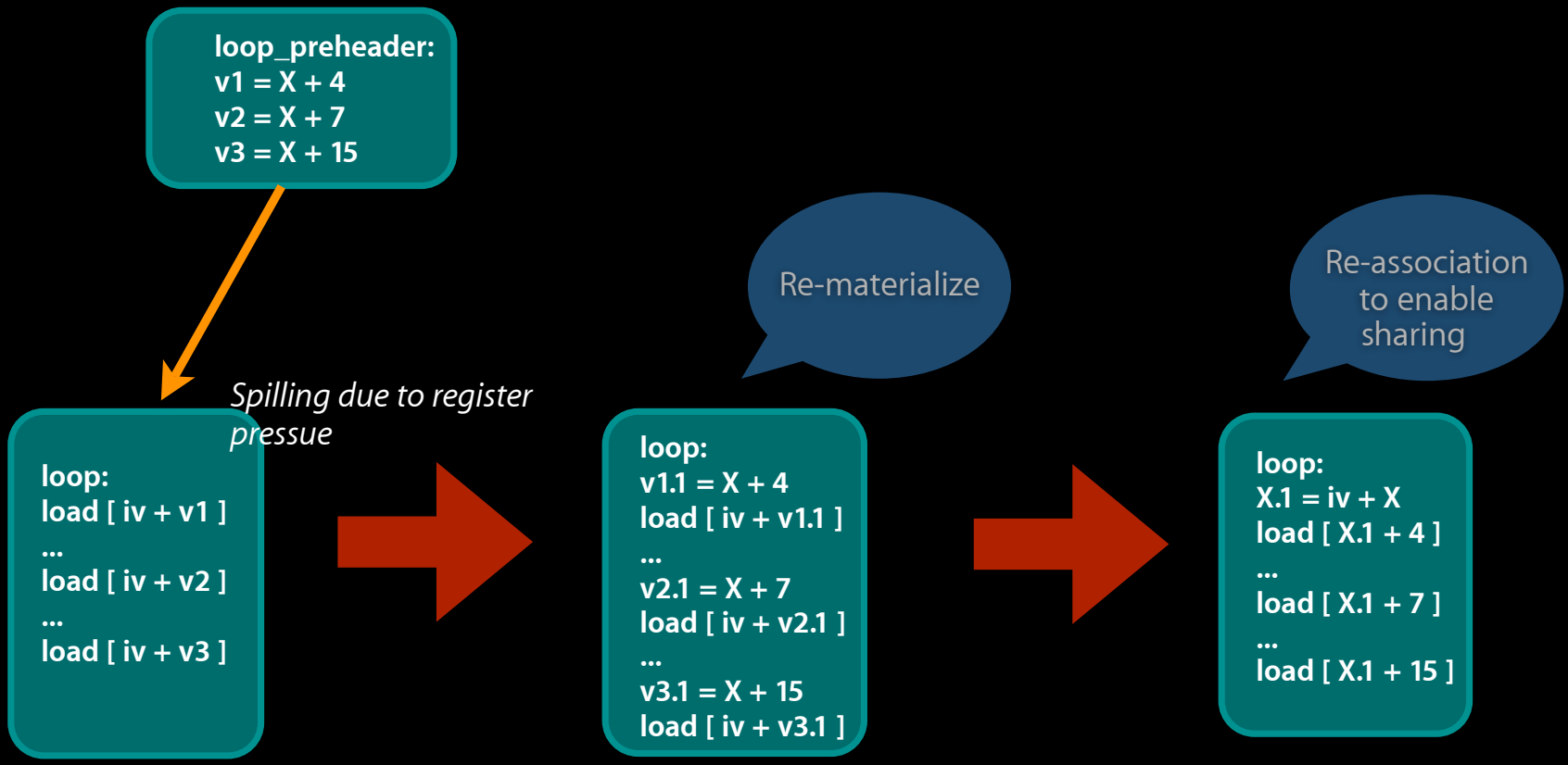


Next Step: Iterative Splitting

- Split on loop boundaries
- Split on basic block boundaries
- Intra-block splitting

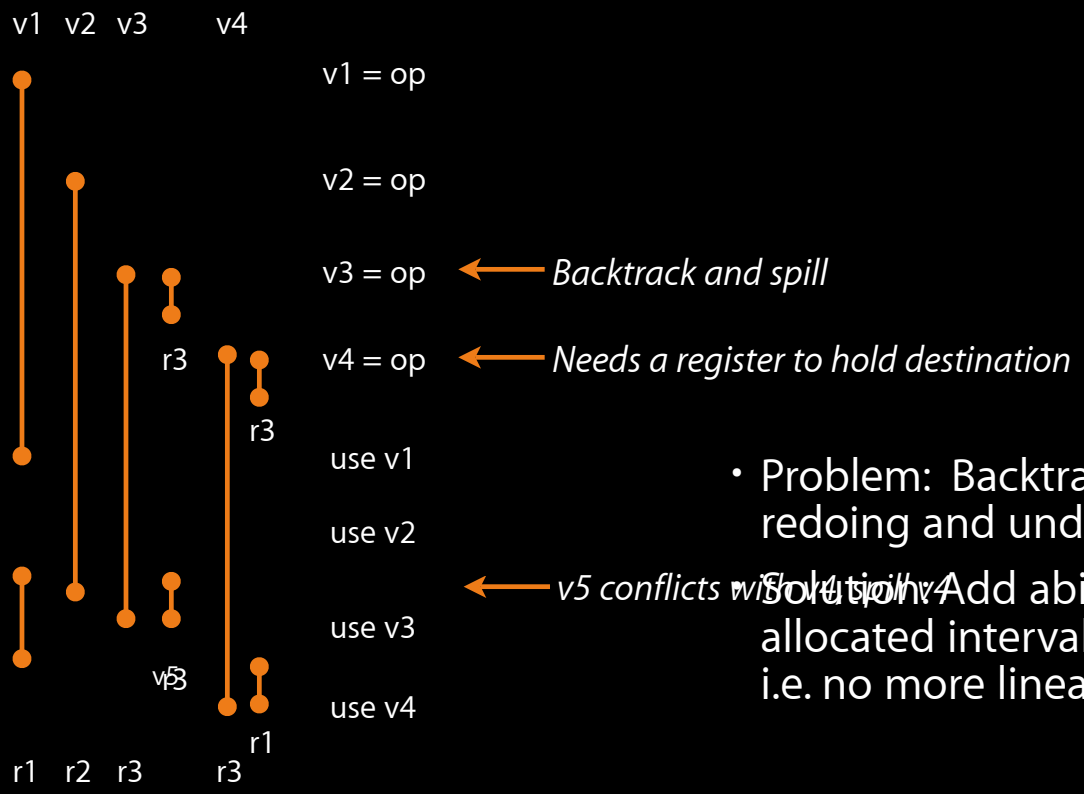


Aggressive Re-materalization



Backtracking in Linear Scan Allocator

- Two conflicting problems:
  - Assign registers aggressively to maximally use them, spilling when they run out
  - Spilling a use requires a register to reload into; a def must also target a register before it is spilled



- Problem: Backtracking is slow, requires redoing and undoing regalloc
- Solution: Add ability to spill previously allocated interval without backtracking, i.e. no more linear scan!

Summary: Much to be done!

- High Level Plans:
  - Smarter PHI elimination for faster compiles
  - Kill backtracking: Use iterative approach instead of linear scan
  - Maintainability: Insert spill code during spilling instead of after regalloc

- Improved Spilling:
  - Split live intervals at arbitrary places
  - Aggressive re-materialization in spiller
    - Use availability info to remat instructions with reg uses
    - Use alias Info to remat loads
  - Reschedule to reduce register pressure?

Questions?