



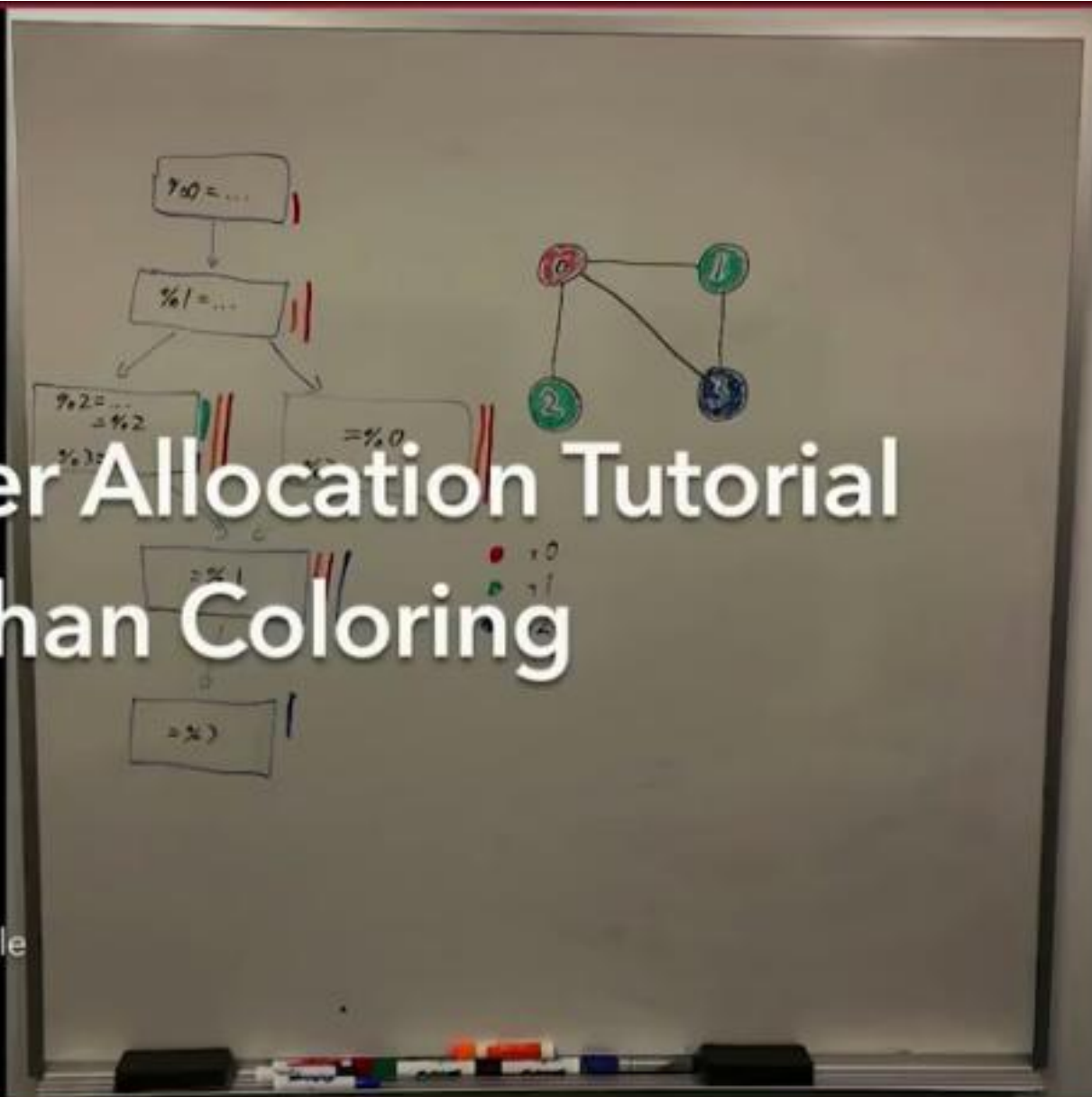
Register Allocation Tutorial More than Coloring

Presenter: Matthias Braun

Register Allocation Tutorial

More than Coloring

Matthias Braun, Apple



**DON'T
PANIC**

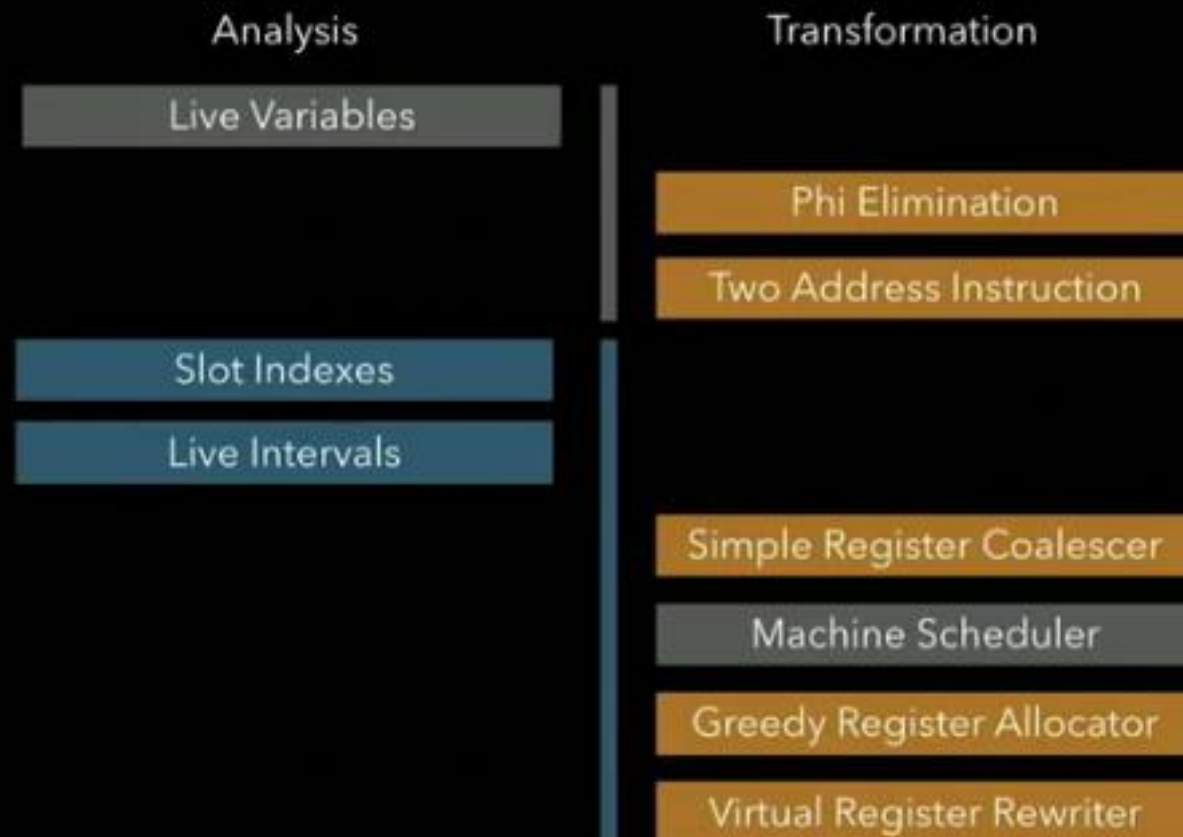
Welcome!

- This is a tutorial about the LLVM register allocation infrastructure and passes.
- Goals:
 - Present "Greedy" allocator
 - Teach concepts to enable debugging
 - High level understanding to enable further exploration
 - Introduction for developers wanting to improve the allocators
- Not enough time for: All the details, alternative allocators

Register Allocation

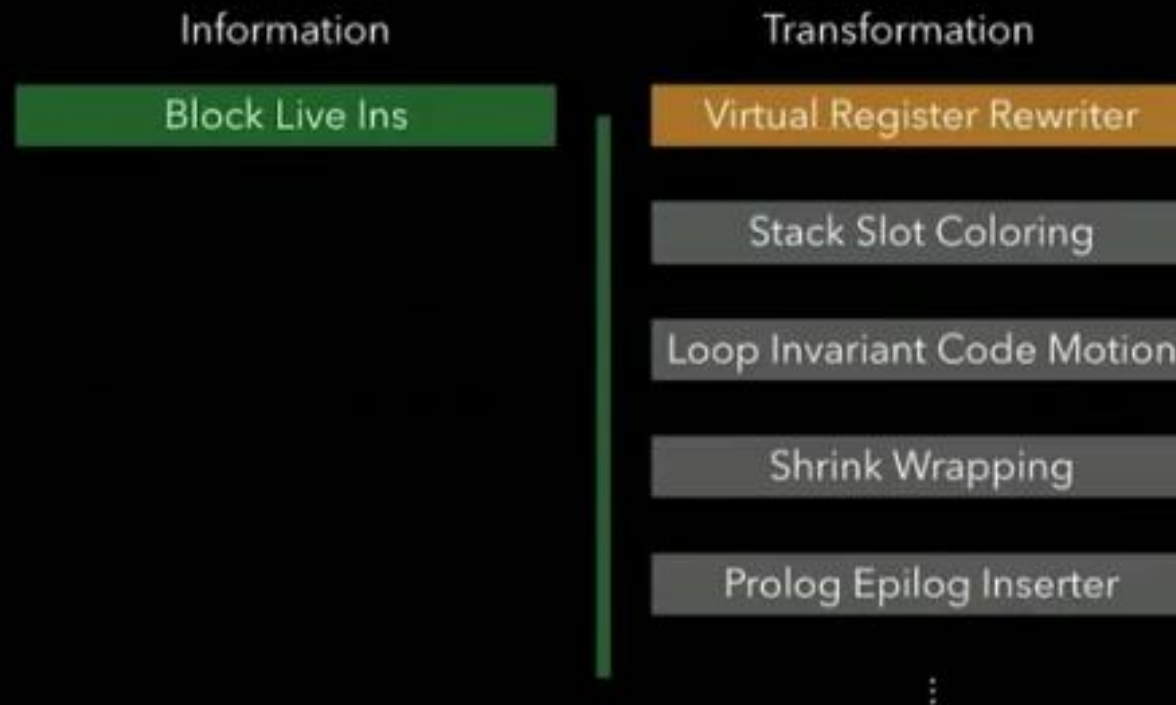
- Register Allocation maps virtual registers to target machine registers.
- Constraints:
 - limited number of registers: stash values to memory
 - respect instruction set and system conventions
- Optimization Targets:
 - minimize spill and reload code, put it in less frequently executed places
 - use as few registers as possible
 - eliminate copy instructions

Allocation Pass Pipeline



Not Shown: Detect Dead Lanes, Process Implicit Defs, Rename Independent Subregs

Related Passes After Register Allocation



LLVM Machine Representation

Virtual Reg. Def Reg. Use

`%1 : gpr32 = COPY %0`

Reg. Class

Physical Reg. Def Reg. Use

`$w0 = Add $w3, 42`

```
%0 = Load ...
%1 = Add %0, 13
Store %1, ...
%2 = Sub %0, 7
Store %2, ...
```



```
$x5 = Load
$x3 = Add $x5, 13
Store $x3
$x3 = Sub $x5, 7
Store $x3
```

Register Allocation

Part 2: Early Passes

Getting Out of SSA: Φ Elimination

- Get out of SSA form
- Replace **PHI** with **COPY** instructions in predecessor block and **PHI** block



Φ Elimination: Why the Extra Copy?


Φ Interferes with own Argument (swap):

```
bb.0:  
...  
bb.2:  
  %10 = PHI %0, %bb.0, %11, %bb.1  
  %11 = PHI %1, %bb.0, %10, %bb.1  
  ...
```



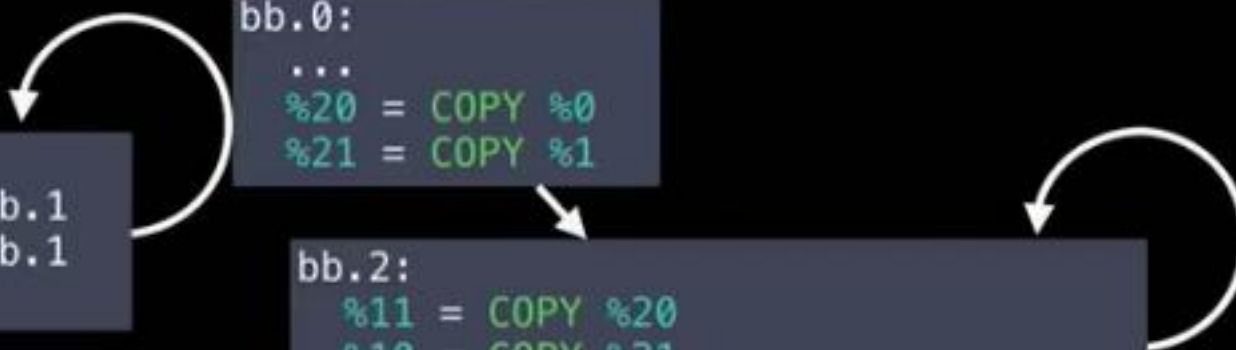
✗ Incorrect Lowering:

```
bb.0:  
...  
  %10 = COPY %0  
  %11 = COPY %1  
bb.2:  
  ...  
  %10 = COPY %11  
  %11 = COPY %10
```



✓ Correct Lowering with 3 COPYs per Φ :

```
bb.0:  
...  
  %20 = COPY %0  
  %21 = COPY %1  
bb.2:  
  %11 = COPY %20  
  %10 = COPY %21  
  ...  
  %20 = COPY %11  
  %21 = COPY %10
```



Example: Φ Elimination

```
bb.0:
  %0:gr32 = MOV32ri 0
bb.1:
  %1:gr32 = PHI %0, %bb.0, %2, %bb.1
  CALL64pcrel32 @func, csr_none
  %2:gr32 = INC32r %1(tied 0)
  CMP32ri8 %2, 100
  JNE_1 %bb.1
bb.2:
  $eax = COPY %2
  RETQ
```



```
bb.0:
  %0:gr32 = MOV32ri 0
  %3:gr32 = COPY %0
bb.1:
  %1:gr32 = COPY %3
  CALL64pcrel32 @func, csr_none
  %2:gr32 = INC32r %1(tied 0)
  CMP32ri8 %2, 100
  %3 = COPY %2
  JNE_1 %bb.1
bb.2:
  $eax = COPY %2
  RETQ
```

Getting Out of SSA: Two Address Instructions

- Two Address Instruction: Writes to same register it reads from
- Cannot enforce this early as SSA enforces new register for every definition
- Implement via **COPY**, can usually coalesce them later
- Expands **REG_SEQUENCE**, **INSERT_SUBREG** to **COPY** sequences

```
%0 = ...  
%1 = Sub %0(tied 0), ...  
... possibly using %0 ...
```



```
%0 = ...  
%1 = COPY %0  
%1 = Sub %1(tied 0), %1  
... possibly using %0 ...
```

Two Address Instruction: Commutative Instructions

```
%0 = COPY $r0  
%1 = COPY $r1  
%2 = Add %0(tied 0), %1  
$r1 = COPY %2
```

Without commuting: Cannot eliminate all COPYs

```
$r0 = COPY $r0  
$r1 = COPY $r1  
$r0 = ADD32rr $r0(tied 0), $r1  
$r1 = COPY $r0 # cannot coalesce
```

After Two Address Instruction Commutes:

```
%0 = COPY $r0  
%1 = COPY $r1  
%2 = Add %1(tied 0), %0  
$r1 = COPY %2
```

Can eliminate all COPYs:

```
$r0 = COPY $r0  
$r1 = COPY $r1  
$r1 = ADD32rr $r1(tied 0), $r0  
$r1 = COPY $r1
```

Example: Two Address Instructions

```
bb.0:
  %0:gr32 = MOV32ri 0
  %3:gr32 = COPY %0

bb.1:
  %1:gr32 = COPY %3
  CALL64pcrel32 @func, csr_none
  %2:gr32 = INC32r %1(tied 0)
  CMP32ri8 %2, 100
  %3 = COPY %2
  JNE_1 %bb.1

bb.2:
  $eax = COPY %2
  RETQ
```



```
bb.0:
  %0:gr32 = MOV32ri 0
  %3:gr32 = COPY %0

bb.1:
  %1:gr32 = COPY %3
  CALL64pcrel32 @func, csr_none
  %2:gr32 = COPY %1
  %2 = INC32r %2(tied 0)
  CMP32ri8 %2, 100
  %3 = COPY %2
  JNE_1 %bb.1

bb.2:
  $eax = COPY %2
  RETQ
```


Simple(?) Register Coalescing

- Eliminate as many **COPY**s as possible
- Live range splitting in later passes will insert new **COPY**s where beneficial

```
%0 = ...  
%1 = COPY %0  
... no redefinition of %0, %1  
    = Use %0  
    = Use %1
```



```
%0 = ...  
...  
    = Use %0  
    = Use %0
```

Example: Register Coalescing

```
bb.0:
  %0:gr32 = MOV32ri 0
  %3:gr32 = COPY %0

bb.1:
  %1:gr32 = COPY %3
  CALL64pcrel32 @func, csr_none
  %2:gr32 = COPY %1
  %2 = INC32r %2(tied 0)
  CMP32ri8 %2, 100
  %3 = COPY %2
  JNE_1 %bb.1

bb.2:
  $eax = COPY %2
  RETQ
```



```
bb.0:
  %3:gr32 = MOV32ri 0

bb.1:
  CALL64pcrel32 @func, csr_none

  %3 = INC32r %3(tied 0)
  CMP32ri8 %3, 100

  JNE_1 %bb.1

bb.2:
  $eax = COPY %3
  RETQ
```


Liveness Representation

Debug Output

```
$ clang -mllvm -debug-only=regalloc test.c # or:
$ llc -debug-only=regalloc test.ll
...
***** INTERVALS *****
%0 [192r,208r:0) 0@192r weight:0.000000e+00
%1 [288r,336B:0)[416B,432r:0) 0@288r weight:0.000000e+00
...
***** MACHINEINSTRS *****
...
0B      bb.0.entry:
        successors: %bb.7(0x30000000), %bb.1(0x50000000); %bb.7(37.50%), %bb.
1(62.50%)
        liveins: $w0, $w1
16B      %7:gpr32 = COPY $w1
32B      %6:gpr32 = COPY $w0
48B      CBNZW %6:gpr32, %bb.1
...
```

i Needs assertions enabled!

Live Intervals: Slot Indexes

```
0B    bb.0:  
16B    %0 = ...  
32B    %1 = ...  
48B    CondJump %bb.2  
  
64B    bb.1:  
  
80B    bb.2:  
96B    Use %1
```

Live Intervals: Slot Indexes

```
0B    bb.0:  
16B    %0 = ...  
32B    %1 = ...  
48B    CondJump %bb.2  
  
64B    bb.1:  
72B    %1 = COPY %0  
  
80B    bb.2:  
96B    Use %1
```

Live Intervals: Segments

```
0B   bb.0:  
16B   %0 = ...  
32B   %1 = ...  
48B   CondJump %bb.2  
  
64B   bb.1:  
72B   %1 = COPY %0  
  
80B   bb.2:  
96B   Use %1
```



LiveIntervals

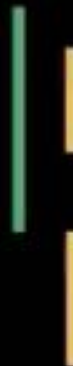
%0: [16;72)

%1: [32;64), [72;96)

`LiveInterval` class contains multiple Segments

Live Intervals: Slots

```
0B   bb.0:  
16B   %0 = ...  
32B   %1 = ...  
48B   CondJump %bb.2  
  
64B   bb.1:  
72B   %1 = COPY %0  
  
80B   bb.2:  
96B   Use %1
```



LiveIntervals

%0: [16r;72r)

%1: [32r;64B), [72r;96r)

Slots

B Base/Block

e EarlyClobber

r Register (Def/Use)

d Dead

Slots denote positions within an instruction

Live Intervals: Value Numbering

```
0B   bb.0:
16B   %0 = ...
32B   %1 = ...
48B   CondJump %bb.2

64B   bb.1:
72B   %1 = COPY %0

80B   bb.2:
96B   Use %1
```



LiveIntervals

%0: [16r;72r:0) 0@16r

%1: [32r;64B:0), [72r;80B:1),
[80B;96r:2)
0@32r, 1@72r, 2@80B

Slots

B Base/Block

e EarlyClobber

r Register (Def/Use)

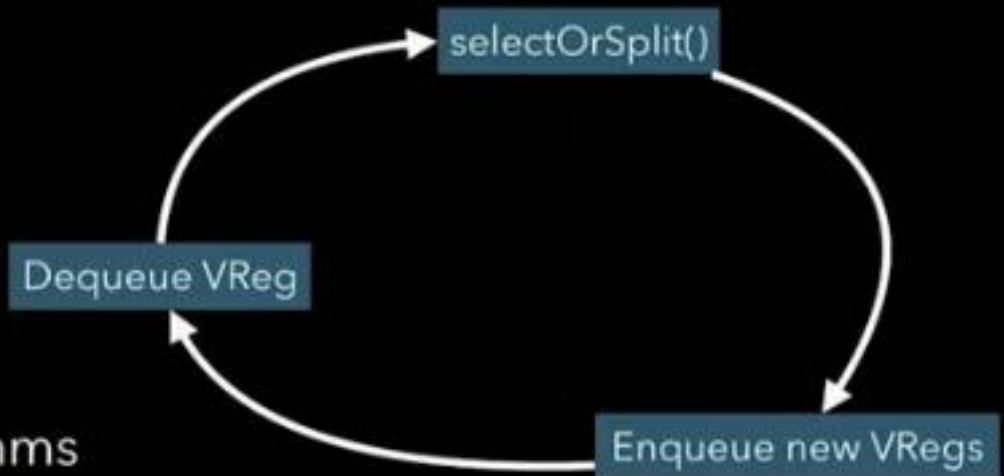
d Dead

Each Segment has value number
similar to SSA

Core Allocator

RegAllocBase

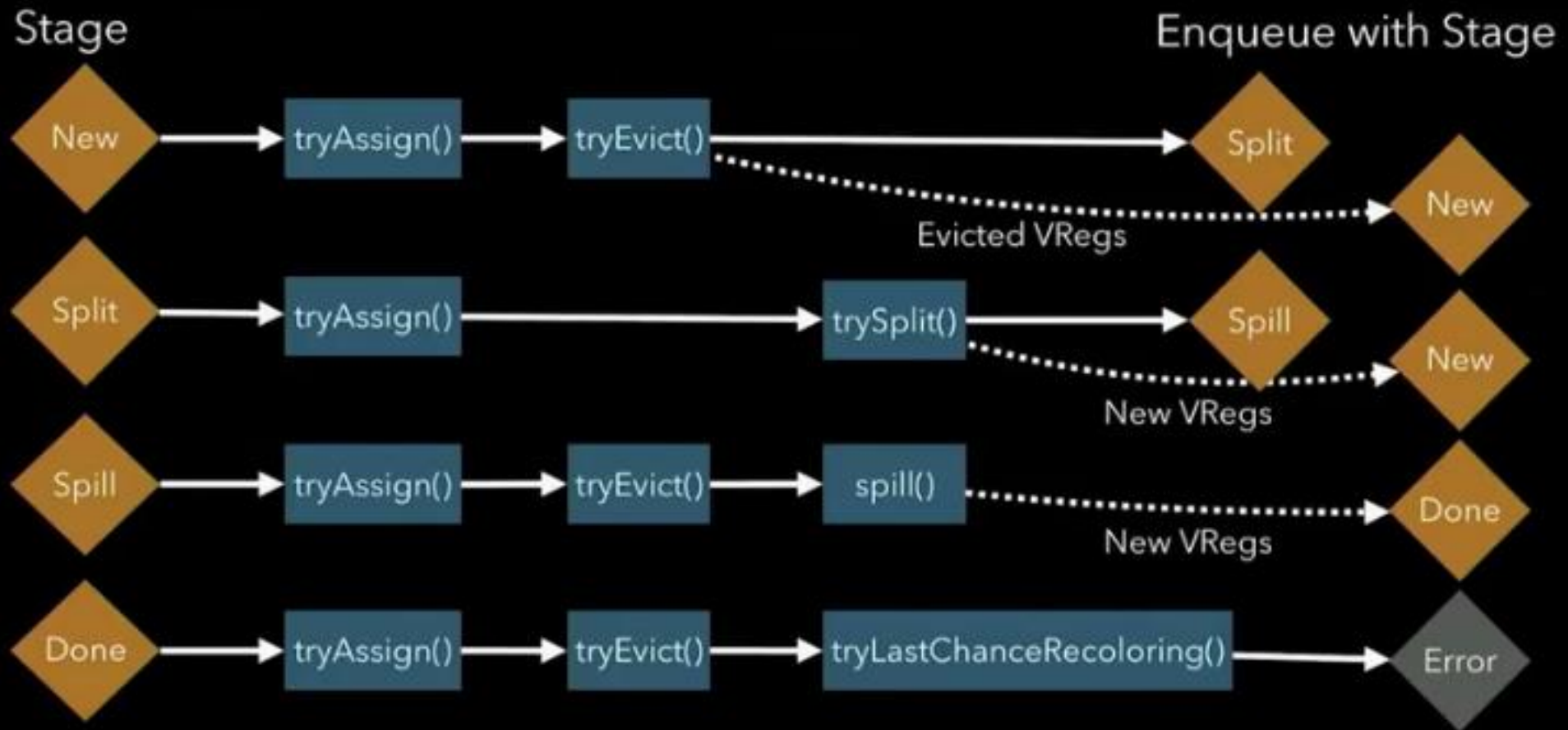
- Provides infrastructure:
 - Priority Queue of Virtual Registers
 - LiveRegMatrix for interference checks (highly tuned B+ tree)
- Designed for iterative/dynamic algorithms
- No upfront interference graph, can add/modify/delete VRegs any time!
- Default Allocator Implementation: RegAllocGreedy



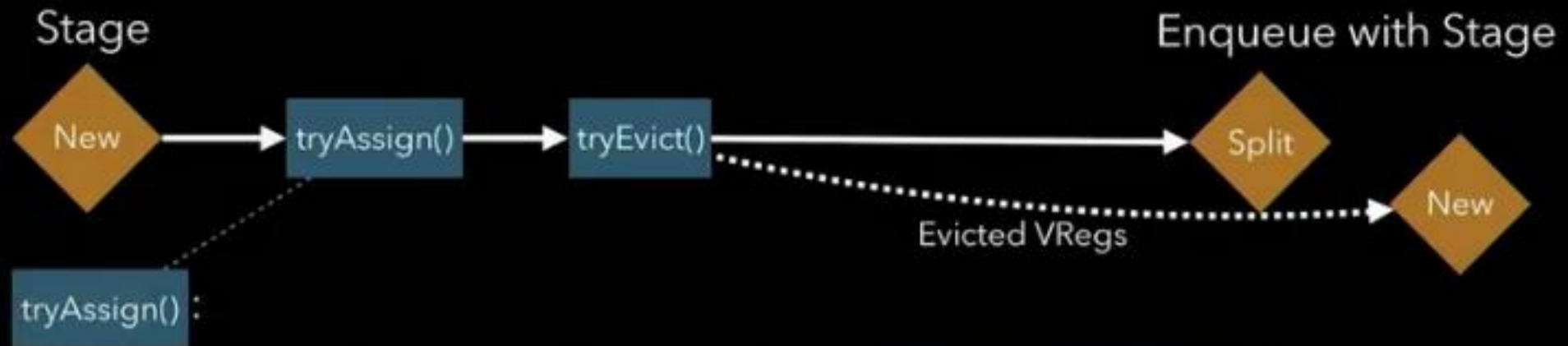
RegAllocGreedy: enqueue()

- Prioritize in this order:
 - Longer global live ranges
 - Live ranges with physical register hints
 - Sort block-local ranges in instruction order
 - Low Priority for virtual registers in late stages (see next slides)

RegAllocGreedy: selectOrSplit()

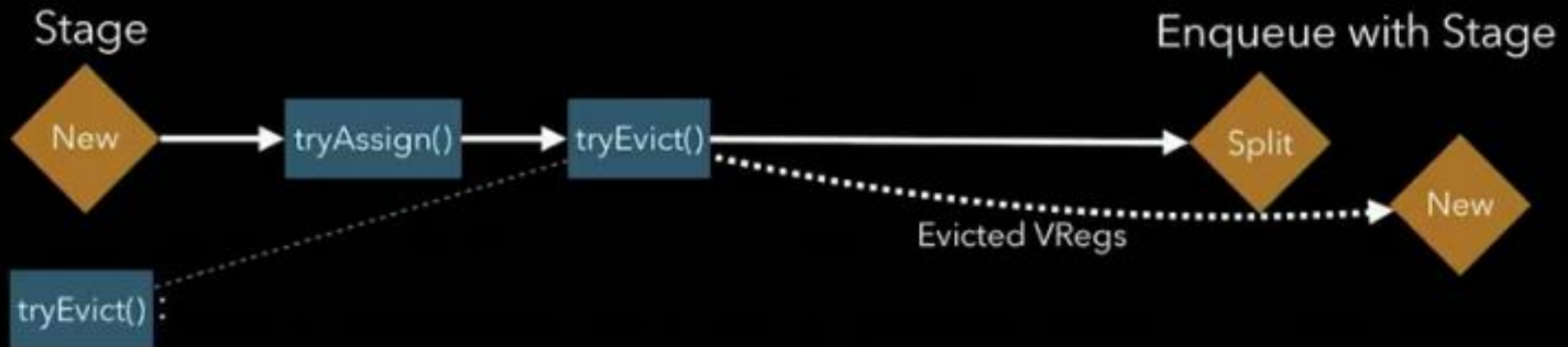


RegAllocGreedy: tryAssign()



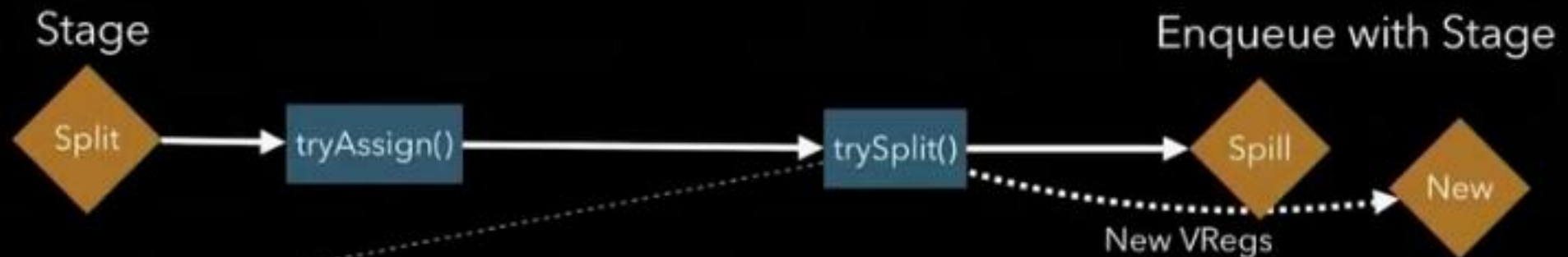
- Attempt to assign to physical registers in AllocationOrder
- If successful we are done

RegAllocGreedy: tryEvict()



- Revert existing assignment if deemed beneficial for spill costs
- (Re-)Enqueue evicted registers with incremented eviction counter
- Ensure progress: Only evict registers with lower counter

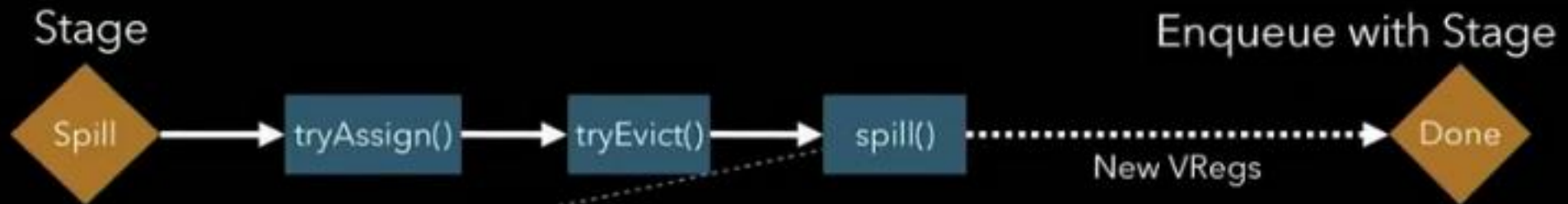
RegAllocGreedy: trySplit()



`trySplit()`:

- Perform live range splitting:
 - `tryLocalSplit()` split around register mask operands (calls)
 - `tryInstructionSplit()` split around constrained instructions
 - `tryRegionSplit()`, `tryBlockSplit()` split at region boundaries (using `SplitAnalysis`, `rematerialize`)

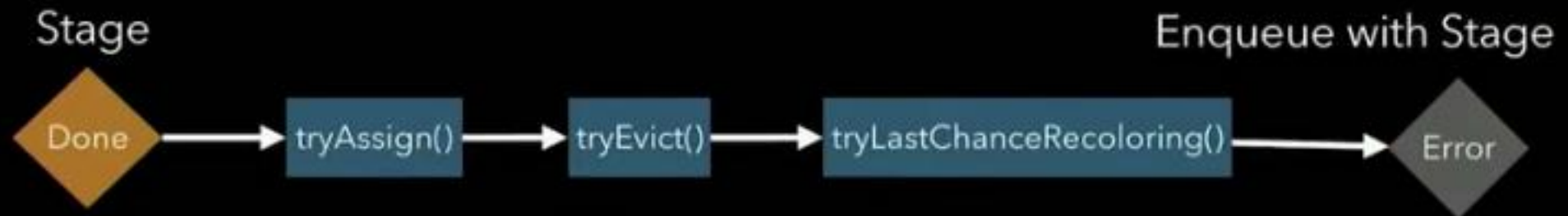
RegAllocGreedy: spill()



spill()

- Create spill and reload instructions
- Use `SpillPlacer` to determine positions

RegAllocGreedy: Error Condition



- Spilling should leave minimal sized live ranges that we can assign
- Error state can be reached with (unrealistically) complicated inline assembly constraints

Example: Allocation: tryAssign()

```
bb.0:
  %3:gr32 = MOV32ri 0

bb.1:

  CALL64pcrel32 @func, csr_none

  %3 = INC32r %3(tied 0)
  CMP32ri8 %3, 100

  JNE_1 %bb.1

bb.2:
  $eax = COPY %3
  RETQ
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%3 [16r,48B:0)[48B,
112r:2)[112r,192r:1)]...
AllocationOrder(GR32) = [ $eax ...
hints: $eax
RS_Assign Cascade 0
wait for second round
```

Example: Allocation: trySplit()

```
bb.0:
    %3:gr32 = MOV32ri 0

bb.1:

    CALL64pcrel32 @func, csr_none

    %3 = INC32r %3(tied 0)
    CMP32ri8 %3, 100

    JNE_1 %bb.1

bb.2:
    $eax = COPY %3
    RETQ
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%3 [16r,48B:0)[48B,
112r:2)[112r,192r:1)...
hints: $eax
RS_Split Cascade 0
...
Split for $eax in 1 bundles, intv 1.
splitAroundRegion with 2 globals.
...
queuing new interval: %4 [56r,104r:
0)...
queuing new interval: %5 [16r,48B:0)
[48B,56r:1)[104r,112r:2)[112r,192r:
3)...
```

Example: Allocation: trySplit()

```
bb.0:
    %5:gr32 = MOV32ri 0

bb.1:
    %4:gr32 = COPY %5
    CALL64pcrel32 @func, csr_none
    %5 = COPY %4
    %5 = INC32r %5(tied 0)
    CMP32ri8 %5, 100

    JNE_1 %bb.1

bb.2:
    $eax = COPY %5
    RETQ
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%3 [16r,48B:0)[48B,
112r:2)[112r,192r:1)]...
hints: $eax
RS_Split Cascade 0
...
Split for $eax in 1 bundles, intv 1.
splitAroundRegion with 2 globals.
...
queuing new interval: %4 [56r,104r:
0)...)
queuing new interval: %5 [16r,48B:0)
[48B,56r:1)[104r,112r:2)[112r,192r:
3)...)

```

Example: Allocation: tryAssign()

```
bb.0:
    %5:gr32 = MOV32ri 0

bb.1:
    %4:gr32 = COPY %5
    CALL64pcrel32 @func, csr_none
    %5 = COPY %4
    %5 = INC32r %5(tied 0)
    CMP32ri8 %5, 100

    JNE_1 %bb.1

bb.2:
    $eax = COPY %5
    RETQ

; VirtRegMap: %5->$eax
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%5 [16r,48B:0)[48B,
56r:1)[104r,112r:2)[112r,192r:3)...
hints: $eax
assigning %5 to $eax...
```

Example: Allocation: trySpill()

```
bb.0:
  %5:gr32 = MOV32ri 0

bb.1:
  %4:gr32 = COPY %5
  CALL64pcrel32 @func, csr_none
  %5 = COPY %4
  %5 = INC32r %5(tied 0)
  CMP32ri8 %5, 100

  JNE_1 %bb.1

bb.2:
  $eax = COPY %5
  RETQ

; VirtRegMap: %5->$eax
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%4 [56r,104r:0)...
hints: $eax
RS_Spill Cascade 0
Inline spilling GR32:%4 [56r,104r:
0)...
spillAroundUses %4
...
unassigning %5 from $eax...
```

Example: Allocation: trySpill()

```
bb.0:
  %5:gr32 = MOV32ri 0

bb.1:
  MOV32mr %stack.0, %5
  CALL64pcrel32 @func, csr_none
  %5 = MOV32rm %stack.0
  %5 = INC32r %5(tied 0)
  CMP32ri8 %5, 100

  JNE_1 %bb.1

bb.2:
  $eax = COPY %5
  RETQ

; VirtRegMap:
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%4 [56r,104r:0)...
hints: $eax
RS_Spill Cascade 0
Inline spilling GR32:%4 [56r,104r:
0)...
spillAroundUses %4
...
unassigning %5 from $eax...
```


Example: Allocation: tryAssign()

```
bb.0:
  %5:gr32 = MOV32ri 0

bb.1:
  MOV32mr %stack.0, %5
  CALL64pcrel32 @func, csr_none
  %5 = MOV32rm %stack.0
  %5 = INC32r %5(tied 0)
  CMP32ri8 %5, 100

  JNE_1 %bb.1

bb.2:
  $eax = COPY %5
  RETQ

; VirtRegMap: %5->$eax
```

```
$ llc -debug-only=regalloc ...
...
selectOrSplit GR32:%5 [16r,48B:0)[48B,
52r:1)[104r,112r:2)[112r,192r:3)]...
hints: $eax
assigning %5 to $eax...
```

Virtual Register Rewriter

- Allocator stores results in `VirtRegMap`. `VirtRegRewriter` is a separate pass.
- Replaces virtual registers with physical registers.
- Initializes block live-in lists.
- Update global state, remove identity copies, ...

Example: VirtRegRewriter

```
bb.0:
    %5:gr32 = MOV32ri 0

bb.1:
    MOV32mr %stack.0, %5
    CALL64pcrel32 @func, csr_none
    %5 = MOV32rm %stack.0
    %5 = INC32r %5(tied 0)
    CMP32ri8 %5, 100

    JNE_1 %bb.1

bb.2:
    $eax = COPY %5
    RETQ

; VirtRegMap: %5->$eax
```



```
bb.0:
    $eax = MOV32ri 0

bb.1:
    liveins: $eax
    MOV32mr %stack.0, $eax
    CALL64pcrel32 @func, csr_none
    $eax = MOV32rm %stack.0
    $eax = INC32r $eax(tied 0)
    CMP32ri8 $eax, 100

    JNE_1 %bb.1

bb.2:
    liveins: $eax
    RETQ
```

Late and Ad Hoc Allocation

Callee Saved Registers

- Callee Saved Registers are saved/restored by `PrologEpilogInsertion` pass
- Shrink Wrapping pass decides placement

Register Scavenging

- Last step of prolog epilog insertion
- Simulate liveness in basic block, allocate virtual registers
- Insert extra spills and reloads if necessary
- Target must allocate space for spill in advance before frame setup!

```
RegScavenger::addScavengingFrameIndex(int FrameIndex)
```

⚠ Precondition: Per virtual register: Only one definition, all uses in same block

LiveRegUnits

- Global liveness information available as block live-in lists

```
bb.0:  
  liveins: $r0, $r3
```

- Use `LiveRegUnits` or `LivePhysRegs` to compute local liveness for instructions inside a block.

Thank You For Your Attention!



- blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-3.0.html
- llvm.org/devmtg/2011-11/#talk6 (Register Allocation in LLVM 3.0)
- llvm.org/devmtg/2017-10/#tutorial3 (The LLVM Machine Representation)