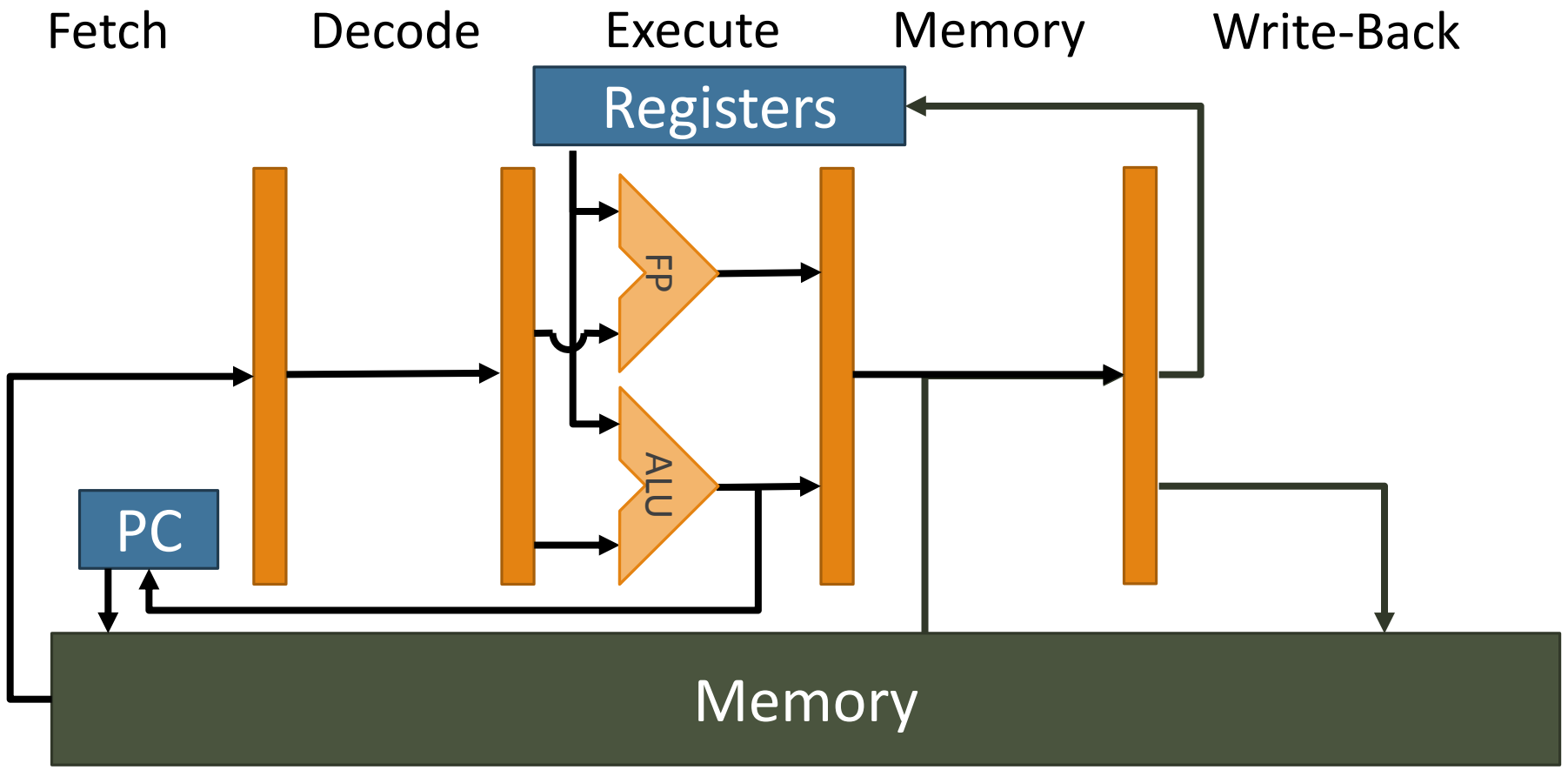
Processors, Performance, and Profiling

Architecture 101: 5-Stage Pipeline



Architecture 101

1. ***Fetch*** instruction from memory.
2. ***Decode*** instruction to get operation and registers.
3. ***Execute*** instruction.
4. ***Memory***: load and store data.
5. ***Write-back*** result to registers.

Architecture 201 (Out-of-Order, Superscalar)

1. ***Fetch*** several instructions from μop-$, I$, L2-$, L3-$, mem…
   - May cause page-fault if address not in memory.
   - Predict whether branch changes next fetch address.
2. ***Decode*** and place in out-of-order queues.
3. ***Execute*** any ready instructions and place in re-order queue.
   - Update register file(s) with result.
4. ***Memory***: load from caches or memory; write to write-queue.
5. ***Write-back*** result to architecture register file.

Architecture 301

Multi-processors and Multi-threading

- Memory coherence and consistency.
- May need compiler to insert barriers.

Very Large Instruction Word (VLIW; e.g. Itanium)

- Several logical instructions packed into one.

SIMD / SIMT (e.g. GPUs)

- Many (usually 32 or 64) threads operate in lock-step.
- Compiler inserts checks for divergence.

Architecture 201 (Out-of-Order, Superscalar)

1. ***Fetch*** several instructions from ***µop-$, I-$, L2-$, L3-$, mem***…
   - May cause ***page-fault*** if address not in memory.
   - ***Predict*** whether ***branch*** changes next fetch address.
2. ***Decode*** and place in ***out-of-order*** queues.
3. ***Execute*** any ready instructions and place in re-order queue.
   - Update register file(s) with result.
4. ***Memory***: load from caches or memory; write to write-queue.
5. ***Write-back*** result to architecture register file.

Cache Lines

Caches contain fixed-size ***lines*** of adjacent addresses.

- Must replace a complete line at a time.
- Fetches / loads across line boundaries may be slower.
  - Slightly higher chance of a cache miss with two lines.
- Cool lines are 1-word long.
- Recent x86 (Haswell, Piledriver) use 64-byte lines.

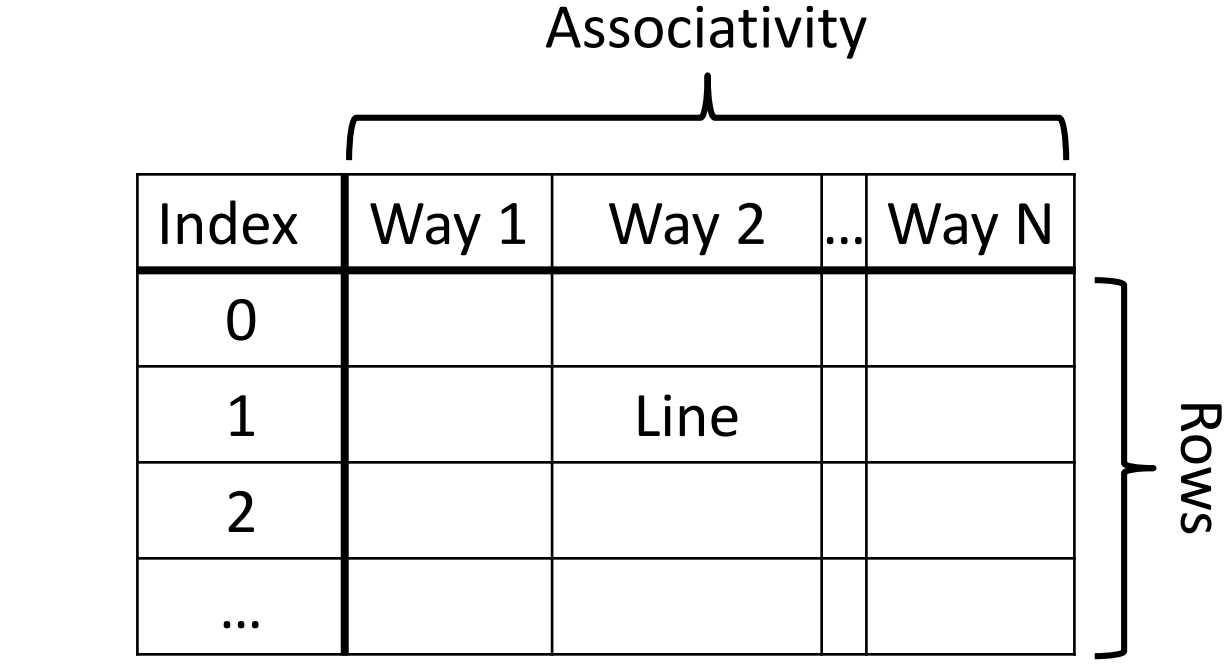
Cache Associativity

Addresses mapped to subset of lines of cache.

- ***N-Way Associativity***: number of lines that may hold an address.

1-way: “Direct mapped”

1 row: “Fully associative”



Size = Associativity * Rows * Line Width

Caches and Performance

COOL SIMULATOR

Fully-associative.

Joint I-$ and D-$.

Small (64 words).

No alignment concerns.

X86

Typically 4-way or 8-way.

L1: split I-$ and D-$.
L2: joint I-$ and D-$.

Size varies: 16kB → 8MB.

- L2 & up shared between cores

Multi-line instructions/data.

Caches and Performance

COOL SIMULATOR

Higher cost of inlining and unrolling.

Fully-associative.

Joint I-$ and D-$.

Small (64 words).

No alignment concerns.

ully 4-way or 8-way.

L1: split I-$ and D-$.
L2: joint I-$ and D-$.

Size varies: 16kB → 8MB.
- L2 & up shared between cores

Multi-line instructions/data.

Caches and Performance

COOL SIMULATOR	X86
Fully-associative.	Typically 4-way or 8-way.
Joint I-$ and D-$.	L1: split I-$ and D-$. L2: joint I-$ and D-$.
Small (64 words).	Size varies: 16kB → 8MB. ◦ L2 & up ***shared*** between cores
No alignment concerns.	Multi-line instructions/data.

Caches and Performance

COOL SIMULATOR	X86
Fully-as	8-way.
Jo	9.
Small (64 words).	Size varies: 1 B → 8MB.
	◦ L2 & up ***shared*** between cores
No alignment concerns.	Multi-line instructions/data.

1) **Static**: Optimize for exclusive access.
2) **"Fat binary"**: implement different time/space tradeoffs (e.g. less inlining, deliberate stalls).
3) **Dynamic**: reactive JIT.

Caches and Performance

COOL

Fully-associative.

Joint I-$ an

Small (6

No alignment concerns.

2-Byte align 1st instruction in function. Loops?

X86

Typically 4-way or 8-way.

L1: split I-$ and D-$.
L2: joint I-$ and D-$.

Size varies: 16kB → 8MB.

- L2 & up shared between cores

Multi-line instructions/data.

μop Cache

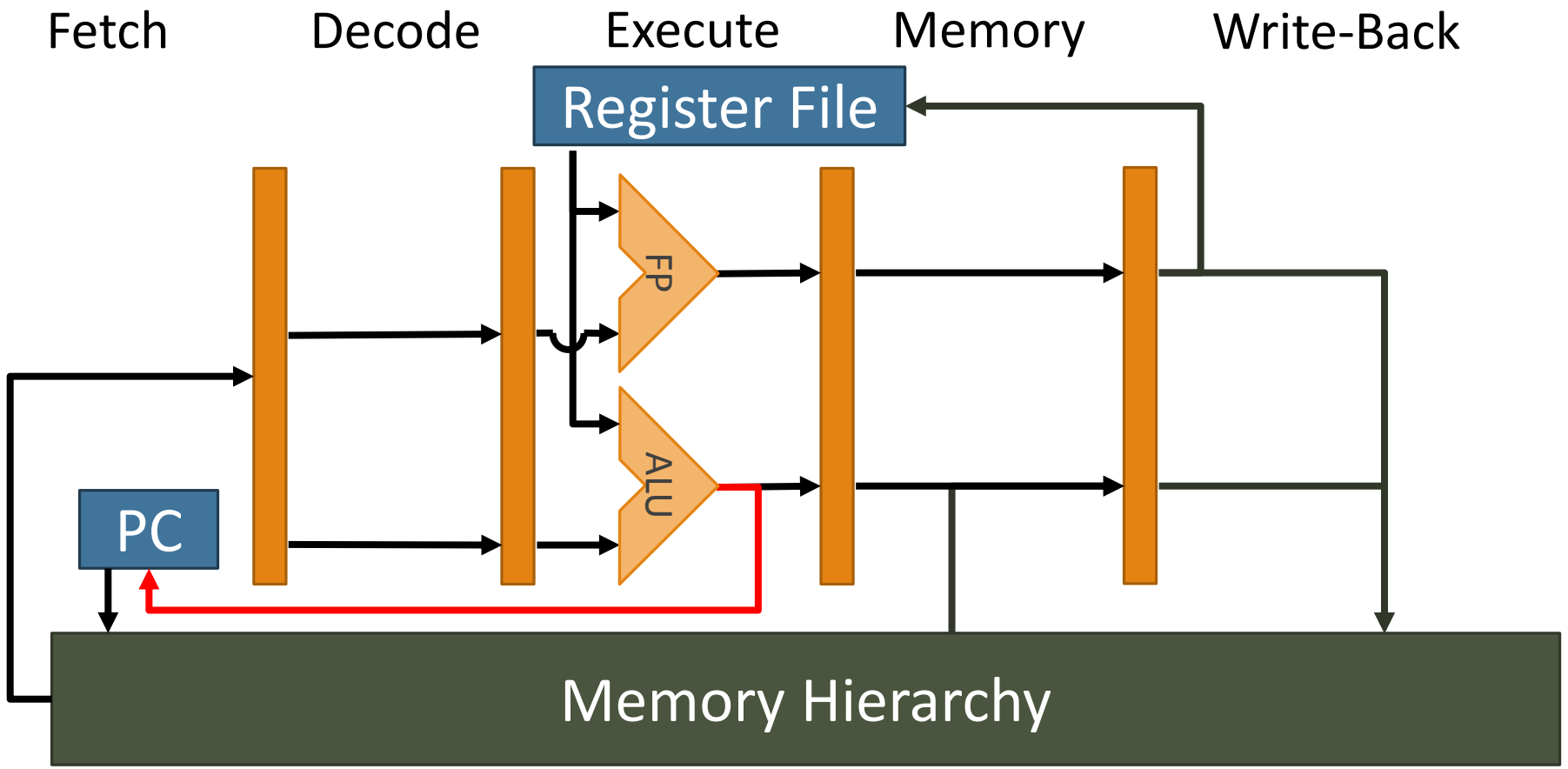
x86 embeds a RISC-like processor inside a CISC processor.

- L1 cache holds CISC instructions.
- μop cache holds RISC instructions.

μop details are typically ***not available***.

- Implementation detail: may change between revisions.
- Expect tight loops to reside in μop.

“5”-Stage Pipeline



Static Branch Prediction

Prediction depends only on current instruction.

- E.g., “always not-taken”.

Cool simulator:

- “Backward taken, forward not-taken” for conditional jumps.
- Prefer calling via labels over calling via registers.
  - Receiver class analysis, but without the inlining.

Dynamic Branch Prediction

Prediction depends on history of previous branches.

- All branches? Or just ones with this (hashed) address?
- How much history?
  - 1 bit: predict what happened last time.
  - 2+ bits: saturating counter or pattern detection.

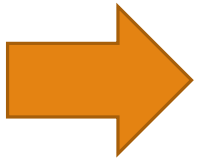
Dynamic Branch Prediction

Prediction depends on history of previous branches.

- All branches? Or just ones with this (hashed) address?
- How much history?

In some cases: avoid branches with predicated moves.

```
cmpq $0, %r13
jz Label
addq %r15, %r14
Label:
```



```
movq %r14, t8
addq %r15, t8
cmpq $0, %r13
cmovz t8, %r14
```

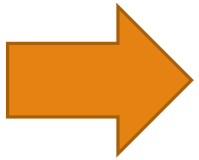
Dynamic Branch Prediction

Prediction depends on history of previous branches.

- All branches? Or just ones with this (hashed) address?
- How much history?

In ***some*** cases: avoid branches with predicated moves.

```
cmpq $0, %r13
jz Label
idivq %r15, %r14
Label:
```



```
movq %r14, t8
idivq %r15, t8
cmpq $0, %r13
cmovz t8, %r14
```

Dynamic Branch Prediction

Prediction depends on history of previous branches.

- All branches? Or just ones with this (hashed) address?
- How much history?

In ***some*** cases: avoid branches with predicated moves.

- May cause false errors (null dereference, divide-by-zero).
- May increase register pressure: spilling.
- May increase instruction count: no benefit if prediction is correct.

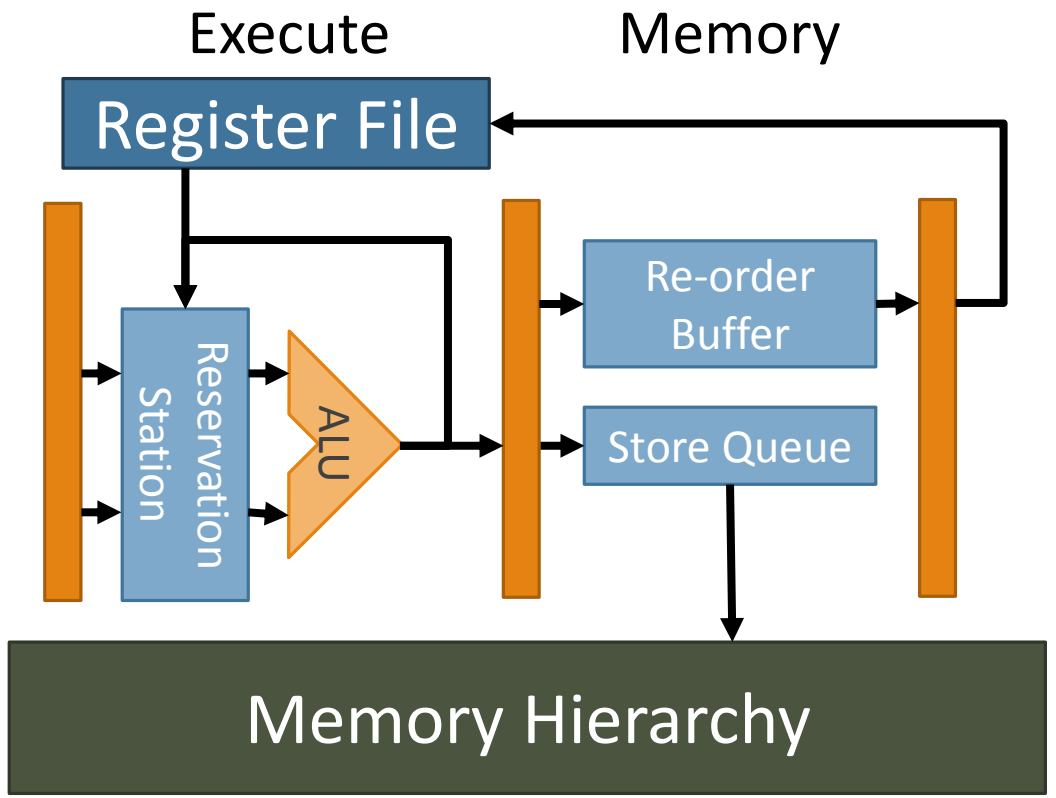
Out-of-Order Execution

Goal: Run instructions as soon as operands and resources are ready.

Pool pending instructions.

Interrupt safety:

- Re-order loads, stores, etc. before committing.



Out-of-Order Performance

Resolves many data dependencies automatically.

- E.g., can overlap independent slow operations.

However,

- Reservation stations have limited size.
- After mis-predicted jump, pipeline is partially flushed.

Taking Advantage of OOO Processors

Group data-independent instructions.

- OOO processor will issue these in parallel.

Issue slow operations (memory loads, floating points) early.

- Gives them more time to complete before needed.
- Single-threaded stores are handled asynchronously (i.e. fast).

More important at tops of functions, rarely used branches.

- E.g., exception handling code.

Too Many Questions

“Inlining ***may*** cause cache misses.”

“Instructions across cache lines ***may*** cause extra misses.”

“Predicated moves ***may*** reduce branch mis-predictions.”

“Predicated moves ***may*** increase register pressure.”

“Independent instructions ***may*** improve throughput.”

“Linear scan allocation ***may*** increase spilling.”

Profiling for Fun and Profit

The rules of Optimization Club

1. Get the right answer.
2. Make the common case fast (Amdahl's Law).
   - E.g., arrange for more cache hits by not unrolling.
3. Make the common case less common (Johnstone's Law).
   - E.g., arrange for fewer cache accesses by removing instructions.

We need to know what's slow before we can fix it.

Cool Simulator Profiler

```
> ./cool --asm hello-world.cl
> ./cool --profile hello-world.cl-asm
PROFILE:           instructions =          83 @    1 =>          83
PROFILE:         pushes and pops =          23 @    1 =>          23
PROFILE:              cache hits =          17 @    0 =>           0
PROFILE:            cache misses =         560 @  100 =>       56000
PROFILE:     branch predictions =           0 @    0 =>           0
PROFILE:  branch mispredictions =           8 @   20 =>         160
PROFILE:         multiplications =          0 @   10 =>           0
PROFILE:                 divisions =        0 @   40 =>           0
PROFILE:           system calls =           2 @ 1000 =>        2000
CYCLES: 58266
```

Cool Simulator Profiler

```
> ./cool --asm hello-world.cl
> ./cool --profile hello-world.cl-asm
PROFILE:           instructions =          83 @    1 =>          83
PROFILE:         pushes and pops =          23 @    1 =>          23
PROFILE:              cache hits =          17 @    0 =>           0
PROFILE:            cache misses =         560 @  100 =>       56000
PROFILE:      branch predictions =           0 @    0 =>           0
PROFILE:   branch mispredictions =           8 @   20 =>         160
PROFILE:         multiplications =           0 @   10 =>           0
PROFILE:                divisions =           0 @   40 =>           0
PROFILE:            system calls =           2 @ 1000 =>        2000
CYCLES: 58266
```

Cool Simulator Profiler

Tells you ***everything*** that costs cycles.

Find dominant cost and reduce it.

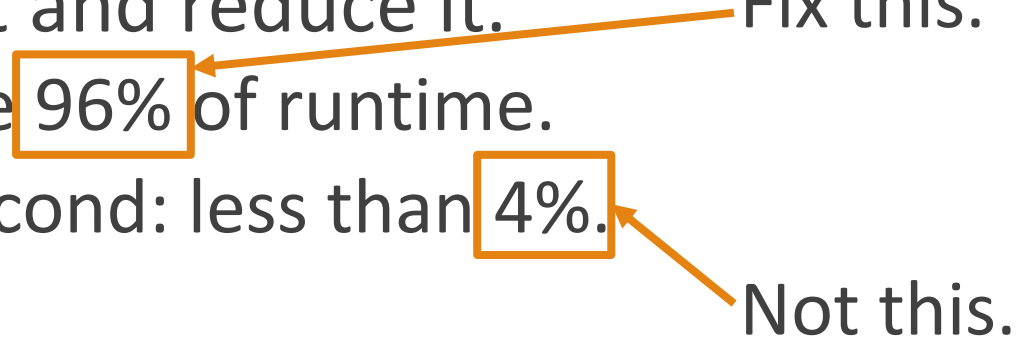
- Cache misses were 96% of runtime.
- Push/pop were second: less than 4%.

Cool Simulator Profiler

Tells you ***everything*** that costs cycles.

Find dominant cost and reduce it.

- Cache misses were 96% of runtime.
- Push/pop were second: less than 4%.



Cool Simulator Profiler

Tells you ***everything*** that costs cycles.

Find dominant cost and reduce it.

- Cache misses were 96% of runtime.
- Push/pop were second: less than 4%.

But first, profile many benchmarks.

- Want to improve ***compiler***, not a specific ***benchmark***.
- Ideally, benchmarks should represent real-world programs.

Function Profiling in Cool

The debug instruction reports current cycle.

- Insert debug sp (or similar) around loops, functions, etc.

```
DEBUG: 357: at time 25, fp = 1999999999 (last set at time 4 by instr at line 130)
Hello, world.
DEBUG: 391: at time 83, fp = 1999999997 (last set at time 78 by instr at line 385)
```

- Note: debug uses a cycle and a cache line!

Linux perf

```
> ./cool --x86 hello-world.cl
> gcc hello-world.s
> perf stat -r 100 -- ./a.out
 Performance counter stats for './a.out' (100 runs):

          0.419577 task-clock                #    0.702 CPUs utilized           ( +-  2.31% )
                 2 context-switches          #    0.005 M/sec                   ( +-  1.52% )
                 0 CPU-migrations            #    0.000 M/sec                   ( +-100.00% )
               135 page-faults               #    0.322 M/sec                   ( +-  0.02% )
         1,003,260 cycles                    #    2.391 GHz                     ( +-  2.69% )
           738,159 stalled-cycles-frontend   #   73.58% frontend cycles idle    ( +-  2.78% )
           610,462 stalled-cycles-backend    #   60.85% backend  cycles idle    ( +-  3.20% )
           586,118 instructions              #    0.58  insns per cycle
                                             #    1.26  stalled cycles per insn  ( +-  0.11% )
           113,068 branches                  #  269.480 M/sec                   ( +-  0.10% )
     <not counted> branch-misses

       0.000597343 seconds time elapsed                                          ( +-  2.10% )
```

Linux perf

```
> ./cool --x86 hello-world.cl
> gcc hello-world.s
> perf stat -r 100 -- ./a.out
 Performance counter stats for './a.out' (100 runs):

          0.419577 task-clock                #    0.702 CPUs utilized            ( +-  2.31% )
                 2 context-switches          #    0.005 M/sec                    ( +-  1.52% )
                 0 CPU-migrations            #    0.000 M/sec                    ( +-100.00% )
               135 page-faults               #    0.322 M/sec                    ( +-  0.02% )
         1,003,260 cycles                    #    2.391 GHz                      ( +-  2.69% )
           738,159 stalled-cycles-frontend   #   73.58% frontend cycles idle     ( +-  2.78% )
           610,462 stalled-cycles-backend    #   60.85% backend  cycles idle     ( +-  3.20% )
           586,118 instructions              #    0.58  insns per cycle
                                             #    1.26  stalled cycles per insn  ( +-  0.11% )
           113,068 branches                  #  269.480 M/sec                    ( +-  0.10% )
       <not counted> branch-misses

        0.000597343 seconds time elapsed                                        ( +-  2.10% )
```

Linux perf

```
> ./cool --x86 hello-world.cl
> gcc hello-world.s
> perf stat -r 100 -- ./a.out
 Performance counter stats for './a.out' (100 runs):

          0.419577 task-clock                #    0.702 CPUs utilized            ( +-  2.31% )
                 2 context-switches          #    0.005 M/sec                    ( +-  1.52% )
                 0 CPU-migrations            #    0.000 M/sec                    ( +-100.00% )
               135 page-faults               #    0.322 M/sec                    ( +-  0.02% )
         1,003,260 cycles                    #    2.391 GHz                      ( +-  2.69% )
           738,159 stalled-cycles-frontend   #   73.58% frontend cycles idle     ( +-  2.78% )
           610,462 stalled-cycles-backend    #   60.85% backend  cycles idle     ( +-  3.20% )
           586,118 instructions              #    0.58  insns per cycle
                                             #    1.26  stalled cycles per insn  ( +-  0.11% )
           113,068 branches                  #  269.480 M/sec                    ( +-  0.10% )
       <not counted> branch-misses

         0.000597343 seconds time elapsed                                        ( +-  2.10% )
```

?

Linux perf

```
> ./cool --x86 hello-world.cl
> gcc hello-world.s
> perf stat -r 100 -- ./a.out
 Performance counter stats for './a.out' (100 runs):

          0.419577 task-clock                #    0.702 CPUs utilized            ( +-  2.31% )
                 2 context-switches          #    0.005 M/sec                    ( +-  1.52% )
                 0 CPU-migrations            #    0.000 M/sec                    ( +-100.00% )
               135 page-faults               #    0.322 M/sec                    ( +-  0.02% )
         1,003,260 cycles                    #    2.391 GHz                      ( +-  2.69% )
                   lled-cycles-frontend      #   73.58% frontend cycles idle     ( +-  2.78% )
                   d-cycles-backend          #   60.85% backend  cycles idle     ( +-  3.20% )
                   tructions                 #    0.58  insns per cycle
                                             #    1.26  stalled cycles per insn  ( +-  0.11% )
           113,068 branches                  #  269.480 M/sec                    ( +-  0.10% )
   <not counted> branch-misses

       0.000597343 seconds time elapsed                                          ( +-  2.10% )
```

This would have been nice…

Linux perf

```
> perf list
> perf stat -r 100 -e cycles,instructions,branches,branch-misses -- ./a.out
 Performance counter stats for './a.out' (100 runs):

          964,559 cycles                    #    0.000 GHz                      ( +-  1.57% )
          564,539 instructions              #    0.59  insns per cycle          ( +-  0.12% )
          109,185 branches                                                      ( +-  0.11% )
            5,357 branch-misses             #    4.91% of all branches          ( +-  1.13% )

      0.000500748 seconds time elapsed                                          ( +-  3.18% )
> perf stat -r 100 -e cycles,instructions,cache-references,cache-misses -- ./a.out
 Performance counter stats for './a.out' (100 runs):

          993,489 cycles                    #    0.000 GHz                      ( +-  1.40% )
          560,510 instructions              #    0.56  insns per cycle          ( +-  0.12% )
           11,281 cache-references                                              ( +-  0.43% )
            2,858 cache-misses              #   25.337 % of all cache refs      ( +-  4.62% )

      0.000567105 seconds time elapsed                                          ( +-  2.82% )
```