

Compilers for architects

STOP DOING Optimization

- Code was never meant to be optimized
- YEARS OF OPTIMIZING yet NO REAL-WORLD USE FOUND for BETTER PERFORMANCE
- Wanted to get better performance anyways? We had a tool for that, it was called "Upgrading hardware"
- "Yes please give me a low memory footprint. Please give me 5% CPU utilization" - Statements dreamed up by the utterly Deranged

Look at what Low-level programmers have been demanding your Respect for all this time, with all the RAM & CPU cores we built for them

(This is REAL optimizations, done by REAL programeers):



?????



???????

```
x2 = number * 0.5f;  
y = number;  
i = * ( long * ) &y;  
i = 0x5F3759df - ( i >> 1 );  
y = * ( float * ) &i;  
y = y * ( threehalfs - ( x2 * y * y ) );
```

????????????????????

"I spent the entire week reducing the system latency by 2ms"

They have played us for absolute fools

Why care about compilers?

HW techniques seem to rule the field: branch prediction, OOO, speculation...

BUT

- Understanding HW/SW tradeoffs and interactions is a useful exercise
- Some ISA features are designed to help compiler optimization
- Not every computer* has an Apple silicon or Intel chip
- uarch can reduce CPI, compiler can reduce... I

Can we do better?

```
addi t0, x0, 0    # t0/i = 0
addi t1, x0, 100 # t1 = 100
j guard
loop: slli t2, t0, 2 # t2 = t0/i * 4
add t2, a0, t2     # t2 = A + t2
lw t3, 0(t2)      # t3 = A[i]
add t3, t3, a1    # t3 = A[i] + c
sw t3, 0(t2)      # A[i] = A[i] + c
addi t0, t0, 1    # t0/i++
guard: blt t0, t1, loop
```

```
for (int i = 0; i < 100; i++) {
    A[i] = A[i] + c;
}
```

Reduce # of computations in loop

```
addi t0, x0, 0    # t0/i = 0
addi t1, x0, 100 # t1 = 100
j guard
loop: slli t2, t0, 2 # t2 = t0/i * 4
add t2, a0, t2    # t2 = A + t2
lw t3, 0(t2)     # t3 = A[i]
add t3, t3, a1   # t3 = A[i] + c
sw t3, 0(t2)     # A[i] = A[i] + c
addi t0, t0, 1   # t0/i++
guard: blt t0, t1, loop
```

instrs: 4 + 100 * 7

```
addi t0, x0, 0    # t0/i = 0
addi t1, x0, 100 # t1 = 100
addi t2, a0, 0    # t2 = A
j guard
loop: lw t3, 0(t2)    # t3 = A[i]
add t3, t3, a1     # t3 = A[i] + c
sw t3, 0(t2)      # A[i] = A[i] + c
addi t2, t2, 4    # advance pointer
addi t0, t0, 1    # t0/i++
guard: blt t0, t1, loop
```

instrs: 5 + 100 * 6

Get rid of i

```
addi t0, x0, 0    # t0/i = 0
addi t1, x0, 100 # t1 = 100
addi t2, a0, 0    # t2 = A
j guard
loop: lw t3, 0(t2)    # t3 = A[i]
add t3, t3, a1    # t3 = A[i] + c
sw t3, 0(t2)      # A[i] = A[i] + c
addi t2, t2, 4    # advance pointer
addi t0, t0, 1    # t0/i++
guard: blt t0, t1, loop
```

instrs: 5 + 100 * 6

```
addi t1, a0, 400 # stop before A[100]
addi t2, a0, 0    # t2 = A
j guard
loop: lw t3, 0(t2)    # t3 = A[i]
add t3, t3, a1    # t3 = A[i] + c
sw t3, 0(t2)      # A[i] = A[i] + c
addi t2, t2, 4    # advance pointer
guard: blt t2, t1, loop
```

instrs: 4 + 100 * 5



How else to reduce total instructions?

```
addi t1, a0, 400 # stop before A[100]
addi t2, a0, 0   # t2 = A
j guard
loop: lw t3, 0(t2) # t3 = A[i]
add t3, t3, a1   # t3 = A[i] + c
sw t3, 0(t2)     # A[i] = A[i] + c
addi t2, t2, 4   # advance pointer
guard: blt t2, t1, loop
```

```
addi t1, a0, 400 # stop before A[100]
addi t2, a0, 0   # t2 = A
j guard
loop: lw t3, 0(t2)    # t3 = A[i]
add t3, t3, a1    # t3 = A[i] + c
sw t3, 0(t2)     # A[i] = A[i] + c
addi t2, t2, 4   # advance pointer
guard: blt t2, t1, loop
```



Downsides to loop unrolling?

Review: conditional instructions

Some ISAs set condition codes as side effect of instruction

overflow, zero, not zero, negative, etc

Often paired with branch instructions that don't have source registers

ex: `cmp eax, 0; jne B` instead of `bne t1, x0`

Can sometimes be used in conjunction with non-branch instructions

Conditional move (CMOVcc) in x86 will move value from memory or register to register based on condition (turn into a nop otherwise)

Arm conditional execution

source

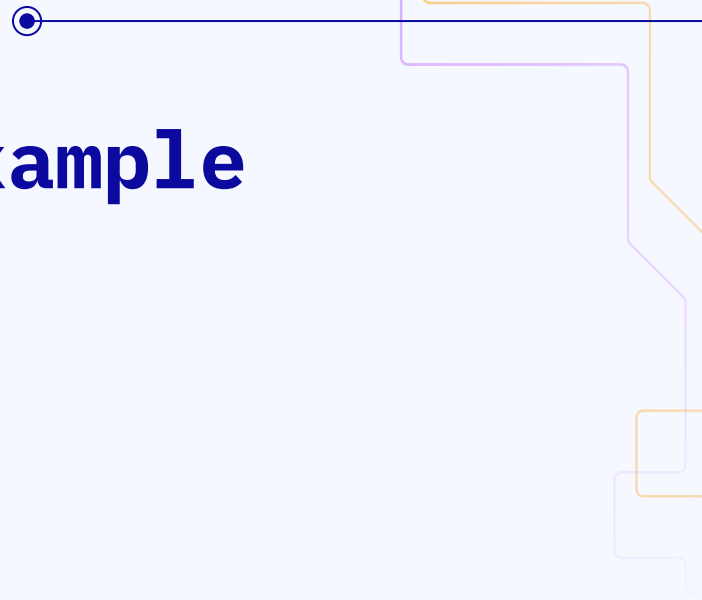
“Almost all ARM instructions can include an optional condition code. This is shown in syntax descriptions as {cond}. An instruction with a condition code is only executed if the condition code flags in the CPSR* meet the specified condition.”

“Almost all ARM data processing instructions can optionally update the condition code flags according to the result. To make an instruction update the flags, include the S suffix as shown in the syntax description for the instruction.”

*Current Program Status Register (holds condition flags)

Arm conditional GCD example

Source





Do conditional instructions introduce hazards
in a traditional pipelined processor?

What about RISC-V?

(from the spec)

The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC and Xtensa ISA), rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a

We considered but did not include conditional moves or predicated instructions, which effectively replace unpredictable short forward branches. Conditional moves are the simpler of the two, but are difficult to use with conditional code that might cause exceptions (memory accesses and floating-point operations). Predication adds additional flag state to a system, additional instructions to set and clear flags, and additional encoding overhead on every instruction.

Both conditional move and predicated instructions add complexity to out-of-order microarchitectures, adding an implicit third source operand due to the need to copy the original value of the destination architectural register into the renamed destination physical register if the prediction is false. Also, static compile-time decisions to use predication instead of branches can result in lower performance on inputs not included in the compiler training set, especially given that unpredictable branches are rare, and becoming rarer as branch prediction techniques improve.

ISA vs uArch

We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict [13, 17, 16] and have been implemented in commercial processors [27].

If the effect of a conditional branch is only to conditionally skip over a subsequent FX or LS instruction and the branch is highly unpredictable, POWER7 can often detect such a branch, remove it from the instruction pipeline, and conditionally execute the FX or LS instruction. The conditional branch is converted to an internal “resolve” operation, and the subsequent FX or LS instruction is made dependent on the resolve operation. When the condition is resolved, depending on the taken or not-taken determination of the condition, the FX or LS instruction is either executed or ignored. This may cause a delayed issue of the FX or LS instruction, but it prevents a potential pipeline flush due to a mispredicted branch.

IBM Power-7 paper

What if the compiler could speculate?

Would want to move speculated instrs before condition evaluation

Why? Might help VLIW scheduling or reducing pipeline hazards

Compiler needs to be able to find such instrs and move them without affecting correctness

We also need to:

Ignore exceptions in speculated execution

Be able to exchange stores and loads/stores



Cannot do this with *just* a compiler; need HW support!

Example of compiler speculation

```
if X == 0; X = Y; else X += 4;
```

```
lw t1, 0(t0)
bne t1, x0, rare
```

```
lw t1, 0(s0)
```

```
j store
```

```
rare: addi t1, t1, 4
```

```
store: sw t1, 0(t0)
```

Assume branch is *almost never* taken (X=Y much more likely than X+=4)

What could go wrong if x != 0?

- Unnecessary page fault
- Y's address could be invalid (memory protection exception)

```
lw t1, 0(t0)
```

```
lw t3, 0(s0) # speculative load
```

```
beq t1, x0, store
```

```
addi t3, t1, 4
```

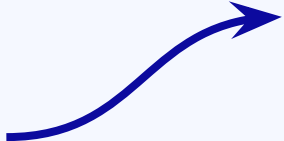
```
store: sw r3, 0(t0)
```


Four approaches to exceptions:

- 1) OS returns undefined value instead of ending execution
Works fine for correct programs, yields incorrect results for programs that will have real exceptions
- 2) ISA has speculative instructions (do not result in exceptions) + exception check instructions used after speculation is resolved
- 3) Track exceptions using “poison bits” on registers that only activate exception when value is used
Need to mark which instructions are speculative
- 4) Hardware buffers speculative instructions (like a ROB-lite)

What about reordering loads/stores?

```
lw t3, 0(t2)
...
sw t3, 0(t2)
lw t4, 4(t2)
...
sw t4, 4(t2)
```



```
lw t3, 0(t2)
lw t4, 4(t2)
...
...
sw t3, 0(t2)
sw t4, 4(t2)
```

Compiler can probably compute that this is fine to do

But what about this:

```
sw t3, 8(t2)
lw t4, 4(t1)
lw t4, 4(t1) # speculative
sw t3, 8(t2) # compare addr
guardian instr # fix if needed
```