ECE 571 – Advanced Microprocessor-Based Design Lecture 7

Vince Weaver

https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

18 September 2024

Announcements

- Homework #1 grades out soon
- Homework #2 reading due Friday

HW#1 Review – Aggregate counts

- bzip2 benchmark – what does it do?
- 19 billion instructions +/- 1000 or so
  (this is test input maybe?)
- 12 billion cycles +/- 100 million
  why would cycles vary?
  Better than last time
- Didn't ask, but cycles/s = 3.3GHz or so (actual=2.6)
- Didn't ask, but roughly what's the IPC? 1.6 or so
- Reversed: similar for instructions but different for cycles

– HW2 will show you why I asked that

HW#1 Review – Sampling

- Perf record: 6.4s (why slower?)

```
  39.19%  bzip2    bzip2                      [.] mainSort
  30.53%  bzip2    bzip2                      [.] mainGtU
  12.54%  bzip2    bzip2                      [.] handle_compress.isra.0
   8.10%  bzip2    bzip2                      [.] generateMTFValues
   7.72%  bzip2    bzip2                      [.] BZ2_compressBlock
```

- Valgrind, 1m28s == roughly 20 times slower?

```
8,150,804,034 (41.35%)  blocksort.c:mainSort
5,619,003,640 (28.51%)  blocksort.c:mainGtU
2,434,222,854 (12.35%)  bzlib.c:handle_compress.isra.0
1,601,050,270 ( 8.12%)  compress.c:generateMTFValues
1,511,856,286 ( 7.67%)  compress.c:BZ2_compressBlock
```

- Gprof, 4.6s

Different results, using function entry instead of exact instruction count for sampling?

```
time    seconds    seconds     calls    s/call   s/call  name
 51.13       1.82      1.82        53      0.03      0.05  mainSort
 20.65       2.56      0.74 89518573      0.00      0.00  mainGtU
 13.48       3.04      0.48        53      0.01      0.01  generateMTFVal
  7.02       3.29      0.25     12223      0.00      0.00  default_bzallo
  5.62       3.49      0.20        53      0.00      0.06  BZ2_compressBl
```

HW#1 Review – Skid

- Skid instructions – mov is more likely than sub?
- movzbl = move 8-bit byte from memory into 32-bit long register, zero extending

```
       |      n = ((Int32)block[ptr[unLo]+d]) - med;
  1.17 |        mov    (%r11),%esi
  0.78 |        lea    (%rbx,%rsi,1),%eax
  1.51 |        movzbl 0x0(%r13,%rax,1),%eax
  5.47 |        sub    %r9d,%eax
       |      if (n == 0) {
  1.77 |        test   %eax,%eax
```

instructions:uppp ppp = precise IP, how much skip 0=arbitrary, 1=constant, 2=request 0, 3=require 0

```
0.88 |        mov    (%r11),%esi
1.48 |        lea    (%rbx,%rsi,1),%eax
5.51 |        movzbl 0x0(%r13,%rax,1),%eax
1.67 |        sub    %r9d,%eax
```

Can check Agner Fog's page for how long instructions might take, remember though they are best case and cache misses on read/write can be much longer. (1 cycle cmp or sub, 3 cycles movz)

SMT Wrapup from last time

SMT Variations

- Fine-grained – rotate threads every cycle
- Coarse-grained – rotate threads only if long latency event happens (cache miss)
- Simultaneous – issue from any combination of threads, to maximize use of pipeline (have to be superscalar)

SMT Push with Sun Niagara

- Remembering this as snagged one Yifeng was throwing out
- They gave a bunch to universities back then (why?)
- 4, 6, or 8 cores each with 4 threads, a lot
- Only one FPU per core though (shared by 4 threads)
- Rock was the next gen even better, but cancelled at last minute during Oracle takeover (was there at conference where the guy was going to present)

SMT Downsides

- Can actually slow down code (especially if both threads trying to use same functional units, also if both using memory heavily as cache is often shared)
- Security? Information Leakage?
- How wide (Sun Niagara aside)

Out-of-Order Processors

- Most modern cores do this, even many embedded these days
- Finding a good writeup hard
- Many just tell you Tomasulo's Algorithm, which was for FPU out of order in 1960s IBM 360. Modern processors don't really do this.

Out-of-Order Processors

- Tries to exploit instruction-level parallelism
- Instead of being stuck waiting for a resource to become available for an instruction (cache, multiplier, etc) keep executing instructions beyond as long as there are no dependencies
- Need to ensure that instructions commit in order
  Need to make sure loads/stores happen in order.

OoO – Register Renaming

- Often parallelism restricted due to dependencies on small number of architectural registers
-
```
add  r0, r1, r2
st   r0, [r5]
add  r0, r3, r4
st   r0, [r6]
```
- Is there a dependency on r0? No, once written to you don't need to track the old value anymore
- Can have much larger physical register sets (80? 100?)

and assign these to instructions ready to execute

- Only map back to architecturally visible ones at retirement

OoO – Ensuring In-order Retirement

- Need to track the original order instructions would retire (graduate)
- Data structure? FIFO? Held there until all previous instructions done, then can write back register values and retire
- Some chips use re-order buffer (ROB)
- Intel chips used register renaming and register alias table RAT
- historical: scoreboards and Tomosulu algorithm

OoO – Load/Store Buffer

- You want loads and stores to commit in order too or do you?
- is there any harm in loads bypassing each other? what about stores happening before loads?
- However if you have the results already, the values of stores in the queue can be forwarded to loads reading from same address
- Have to be careful if speculative execution

OoO – Load/Store Correctness

- Can be problem on systems with mmap I/O if loads happen wrong order
- Stores bypassing can be bad, what if they can bypass a lock?
- Solution? Memory barriers, special instructions that make all loads/stores be resolved before continuing
- Varies a bit with memory model
  - strong/sequential (x86)
  - weak/relaxed (arm)

- doing things in software can be a pain, to make x86 emulation easier apple arm chips have a mode that makes it more x86-like

Speculative Execution

- What do you do about branches?
  They happen often
- Stall?
- Branch prediction, guess which way things go
- What happens when you're wrong? Need to flush all the pipelines and re-start

Precise Exceptions

- One example is skid from HW#1
- What happens on exception? (interrupt, branch mispredict, etc)
- Many instructions “in-flight”, which one caused exception?
- Need to find out which one did, then back things out so can restart
- A lot of trouble

Benefits to OoO

- Can get a nice performance boost

Downside to OoO

- Security issues: spectre, meltdown
- A lot more complex than in-order
- Can waste power, especially if you re-execute or throw out code due to speculative execution

Perf Counters related to Stalls

- Front-end stalls – fetch, decode, icache misses
- Back-end stalls – memory accesses

Real-World Pipelining Examples (from P&H)

- ARM Cortex-A53 (found in Pi3)
  - Eight-stage pipeline
  - Dynamic multi-issue, two instructions
  - Static in-order pipeline
  - First 3 stages fetch two insns at a time, filling a 13-entry instruction queue (branch predictors)
  - Pipelines: one for load, one for store, two for ALU, one multiply, one divide, one FP/SIMD (mul/div/sqrt)

one FP/SIMD for other

- What's the peak possible IPC?
- Patterson and Hennesey report SPEC CPU 2006 INT results. Best case is hmmer (search for gene sequence) with IPC 1.03 (CPI 0.97). Worst is mcf (public transit vehicle scheduling) IPC 0.12 (CPI 8.56). Mostly memory constrained.
- In-order so depends a lot on compiler to get good performance.
- 100mW (1 core at 1GHz)

• Intel Core i7 920 (Nehalem, 2008)

- Decodes CISC instructions to micro-ops
- Can issue up to 6 micro-ops per cycle
- 14 pipeline stages
- dynamic out-of-order with speculation
- register renaming, useful with speculation, as no need to store snapshot to undo speculation, just mark the speculated register results as invalid
- Instruction fetch, fetches 16 bytes. If wrong, 15 cycle penalty
- Predecode instruction buffer – transform 16 bytes (x86 insns 1-15 bytes) into x86 insns

- 18-instruction instruction queue.
- Micro-op decode – three decoders handle decode of instructions that map to 1 uop. One other handles microcode engine that produces longer sequences, up to 4uops a cycle.
- Can also do micro-op fusion (fuse two different insns into one uops, such as cmp/branch)
- Micro-ops go ins a 28-entry uop buffer
  Loop Stream Detector – if code is in tight loop (less than 28 insns) it can execute from this buffer and not need to fetch.

- Instruction issue. Reservation station. Up to six uops can be issued
- Finished instructions go back to reservation station and retirement unit, wait to update register state when determined it is no longer speculative.
- Once instruction hits the head of the reorder buffer, instruction commits and is removed from re-order buffer
- Even though 6 uops can issue, only 4 can be finished a turn? What's the peak IPC? (4)
- Again, SPECCPU. Best is libquantum IPC=2.2 (CPI

0.44). Worst, again, mcf IPC=0.37 (CPI=2.67)

◦ Where do the wasted cycles go? Stalls? But also mis-speculation where work is done and then thrown out.

◦ 130 Watts (2.66GHz)