yzhu303@ucr.edu

Branch Prediction

1. Definition
    1. Control Hazards: conditional jump instructions (for, if-else, switch, while)
    2. Taken or Not Taken: T means using branch target address, NT means ignore the address and go PC+1.
    3. The T/F statements in high-level languages do not directly map to T/NT in assembly level. The compiler can adjust the order of the two branch directions.
2. Branch predictors
    1. Changes the state based on actual outcomes (learners that fitting actual values).
        1. If NT, state--; If T, state--.
    2. If guess right: no penalty(speculative execution)
    3. If wrong: flush IF & ID; reset PC
3. Local Predictor
    1. Each branch instruction has its own state as an entry (unlimited entries)
    2. Each entry describe its state using 2 bits: 00&01 for NT, 10&11 for T
4. Global Predictor
    1. All branch instructions using the same global state table
    2. Store N-bit history in 2^N entries, each entry has a K-bit state
    3. Fits for N-th order Markov process (larger input dimensions and stronger pattern-matching capabilities)
5. Hybrid Predictors
    1. Tournament Predictor: use a higher-level predictor to decide whether to use the global or local predictor.
    2. gshare: Entry Index = XOR(Branch PC, Global History)
    3. TAGE
        1. Multi-level partially cached state tables
        2. The longest matching entry; if mismatch, use the base predictor
    4. Perceptron
        1. Simple neural networks: vector dot product, can be computed within 1 cycle(just like conventional ones)
        2. Input: global history or local histroy or both. -1 for NT, +1 for T
        3. Update: For all decision bits which is the same as the output bit, weigh++; For different, weight--
        4. Output: NT for negative, T for Non-negative
        5. Ideally, each static branch is allocated its own perception

6. Conclusion: AMD Zen 2 use perceptron for 1st level predictor and TAGE for 2nd level.
   1. Perceptron is faster for training ("bit paralleled") and inference (vector dot product)
   2. TAGE is more accurate.
   3. TAGE is more storage efficient, while the best histroy length of perceptron is also long.

Date Hazards & OoO Scheduling

1. Stall: avoid data hazards
2. Data forwarding: save 1 cycle for dataflow from WB(first half cycle) to ID(second half cycle)
3. Compiler optimizations: can help but is limited
   1. Generalizability: needs to be applicable to different microarchitectures
   2. Static Nature: cannot optimize dynamic instructions (lack of infomation about branch prediction and cache)
4. False dependency: WAR/WAW (overwrites the same register, structual hazard)
5. Register renaming
   1. Done by processor (compiler have no infomation about physical registers)
   2. Map architectural registers to physical registers
      1. Mapping table
      2. Reference Counters
6. OoO
   1. Sequential consistency: a later instruction cannot write back before all prior ones has finished
   2. Speculative execution
      1. ReOrder Buffer: store results for speculative instructions. If an ealier instruction failed to commit(exception or mis-prediction), the physical registers and all ROB entries after the failed instruction are flushed.
      2. Commit/Retire: release ROB entry when all prior instructions are non-speculative
7. Superscalar
   1. Fetch Width, Decode Width
   2. Renaming Width, Commit Width
   3. Issue Width: the number of instructions that can be issued to several "entry points" (AG, ALU, BR, MUL) per clock cycle
   4. CPI

$$CPI_{theoretical} = \frac{1}{min(Width_{issue}, Width_{fetch}, Width_{decode})}$$

8. Example: popcount

1. Loop unrolling: fewer dynamic instructions, fewer branch instructions but higher mis-prediction
2. Lookup table: fewer instructions but lower CPI
3. Static loop: unrolled by compiler

9. Programming Tips
    1. Minimize critical path
        1. Cache & Memory: loopup table
        2. Branch: sorting and unrolling
    2. Exploit ILP
        1. Overlap to hide latency
        2. Use more functional units
    3. Builtin instructions
    4. Compiler
10. Modern Processors' Functional Unit
    1. Branch
    2. INT
    3. VEC
    4. MEM
        1. STD(cached)
        2. STA(actual)
        3. Load(actual)
11. Diagram
    1. Pipeline
        1. Stages
            1. IF
            2. ID
            3. EX(ALU/BR/AG)
            4. MEM 4 stages(M1, M2, M3, M4)
            5. WB
        2. Time-Stage Diagram with Instruction ID
        3. Instruction ID-Time Diagram with Stage
    2. OoO: Time-Stage Diagram with Instruction ID
        1. Stages
            1. IQ
                1. IF
                2. ID
                3. REN
            2. ALU, MUL
            3. MEM
                1. AG

        2. M1, M2, M3, M4
    4. BR
    5. ROB
2. Assumption:
    1. Infinite Renaming Width (Physical registers)
    2. Infinite Commit Width
    3. Fetch Width = Decode Width = Issue Width

Parallel Architecture

1. Simultaneous MultiThreading
    1. Problem
        1. Poor ILP within a single program wastes lots of stages when functional units are idle
        2. Another program to further ultilize the pipeline
    2. Design
        1. Duplicate PC, duplicate physical resigters
        2. Not necessary for duplicated cache, functional untis, reorder buffer
    3. Pros
        1. More tolerable to branch mis-prediction: can execute the other thread instead of speculative execution
        2. More tolerable to cache miss: can execute the other thread instead of waiting
    4. Cons
        1. Hurt single-threaded performance: sharing resource
        2. Higher cache miss rate
2. Chip MultiProcessors
    1. Design: cores have own registers and L1&L2 cache, but share L3 cache(Last-level cache)
    2. Coherency & Consistency
        1. Coherency: share the same value between processors
            1. Snooping protocol: each processor broadcasts/listens to cache miss
            2. State
                1. Invalid
                2. Shared: can read by this processor, may also exist on other processors
                3. Exclusive: can read/write by this processor, other processors cannot access
            3. 4Cs of cache miss: compulsory, conflict, capacity, coherency
            4. True sharing & False sharing
        2. Consistency: see the changes in the same order between processors

        1. Non-determinstic: OS scheduling, processor's OoO scheduling...
3. Comparison
    1. If we don't parallel the program, no speedup on SMT/CMP
    2. It's not clear which one will have better performance in branch prediction or cache hit (while running multiple programs, same or different)
    3. They are designed for latency-sensitive, parallelism-limited tasks.
4. GPU
    1. Vector processing and simple operations
        1. Simple ALUs
        2. Almost branches
    2. Deadline driven and throughput-oriented (rather than latency-oriented)
        1. High-bandwidth memory cause higher latency
        2. ALUs can be more but slower(dark silicon)

Dark Silicon & Golden Age

1. Power Consumption
    1. Power and Energy

$$Energy = Power \times ET$$

        1. Lower power does not necessary means lower energy if it slows down too much
    2. Dynamic/Active Power

$$P_{dynamic} \sim a \times C \times V^2 \times f \times N \sim V^3$$

        1. a : average switches per cycle(statistical, no dynamic power if transistor does not switch.)
        2. C: capacitance
        3. V: voltage
        4. f: frequency, highest frequency usually linear with V
        5. N: the number of transistors
    3. Static/Leakage Power

$$P_{leakage} \sim N \times V \times e^{-V_t} \sim V$$

        1. e^{-V_t}: threshold voltage where transistor conducts
    4. Power Density

$$P_{density} = \frac{P}{area}$$

    5. Conclusion

        1. Dennard Scaling discontinued: We cannot make voltage lower, if we want to charge/switch transistors quickly(in high frequency).
        2. Moore's Law shows transistors are becoming smaller and faster(higher frequency), leading Power Density become extremely high.
        3. Lowering the frequency(without changing voltage) helps reducing the heat generation(per second), but will consume more energy.
2. Dark silicon problem
    1. Assumption
        1. Fixed power budget
        2. Fixed computation workload
        3. Same power efficiency
    2. Solution
        1. Same finish time
        2. Different schedule & different frequency
        3. Different latency per job
    3. LIKWID: profiling tool providing power/energy information
3. Golden Age
    1. big.LITTLE system
        1. With limited thread-level parallelism, big.LITTLE would deliver at least the same level of performance as LITTLE only (big cores perform better in single-threaded program)
        2. With rich thread-level parallelism, big.LITTLE would deliver at least the same level of performance as big only (we can have more cores within the same area of chips and LITTLE cores are more power efficient)
    2. FPGAs & DSAs
        1. FPGAs
            1. Do not outperform GPU all the time
            2. Lookup Tables and reconfigurable wires(or interconnects)
            3. ASIC-like but programmable
                1. flexible for functions(e.g. changing algorithm or key)
                2. simulator for verification
            4. Higher latency but significant CPU savings(computation offloading)
        2. DSAs
            1. Offer the best performance-per-watt but the worst programmability
            2. Optimizations
                1. Datapath specialization & Fixed function: no instruction units
                2. Local and optimized memory: datapath rearrangement
                3. Reduced overhead: no OoO scheduling and not branch control units
                4. Parallelsim: multiple circuit replica

3. TPU
    1. Integrated with high-level frameworks like TensorFlow, exposing only minimal details to the user
    2. 16-bit(half-precision) dense GEMM
    3. CISC-like deign that can reduce IC, but almost no opportunity for ILP(pipelining)
    4. Implements Data-Level Parallelism for memory-bounded applications
    5. 8x faster and 25% power consumption
3. Socs
    1. Activate different specialized processors for different tasks
    2. CPU-centric, other processors incur data movement overhead

Assignments

1. When it is difficult to determine the proportion of the linear part in Amdahl's Law, revert to the original definition of speedup.
2. do-while execute at least once, first execute loop iteration and then check the condition.