yzhu303@ucr.edu

Performance

1. Intro
   - For a single program,

     $$Performance = \frac{1}{ET}, ET = IC \times CPI \times CT, CT = \frac{1}{frequency}$$

   - $$TFLOPS = \frac{\#flops \times 10^{-12}}{ET} = \frac{\%flops}{CPI \times CT}$$

     IC is ignored, but with more iterations we might have larger datasets which potentially changes IC. It is better to consider all 3 factors.
   - Goodput(Inference/Frames per second) vs Throughput(Bandwidth/IOPS/FLOPS) vs Latency(time)
   - Benchmark suite: same program with same result and validate result from different machines
   - $$Energy = Power \times ET$$

     Generally speaking, higher the power consumption, lower the time, leading to less energy(more efficient), even if with higher cooling cost. CO_2e is roughly proportional to $Energy$.
2. Factors
   - PLs: Java runs code in JVM, which translates virtual class code into physical machine code. So it is the most costly.
   - Programmers: Can set cycle time; can reduce CPI; but usually increase IC.
   - Compilers: Usually reduce IC, but CPI grows higher. No guarantee for performance.
   - Complexity: Not every machine instruction takes the same amount of time; not every abstraction operation takes the same amount of time.
3. Speedup
   - Definition:

     $$n = \frac{ET_x}{ET_y}, Speedup = \frac{ET_x}{ET_y}$$

   - Amdahl's Law:

     $$Speedup(f_1, \ldots, f_n, s_1, \ldots, s_n) = \frac{1}{(1 - \sum_i^n f_i) + \sum_i^n \frac{f_i}{s_i}}$$

- Corollary:
    1. Each optimization has an upper bound $\frac{1}{1-f}$.
    2. Make the common case(most costly part) fast.
    3. Optimization has a moving target(f_i and s_i, count separately or together).
    4. Parallelism is the key, but single-core performance still matters $\frac{1}{1-f_{parallelizable}} = \frac{1}{f_{unparallelizable}}$.
    5. Don't hurt the non-common case too much($\frac{1}{\frac{(1-f)}{perf(r)}+\frac{f}{s}}$).

Memory Basic

1. In von Neumann Architecture machine, instruction are also from memory. So all instructions need to fetch from memory at least once, some memory loads/stores fetch again.

   $$CPI_{avg} = 1 + (100\% + \%Memory\ Instruction) \times (CPI_{hit} + Miss\ Rate \times CPI_{miss\ penalty})$$

2. Locality
    - Spatial: Consecutive
        - Code: Sequential execution
        - Data: Sequential access
    - Temporal: Frequently
        - Code: Loops, frequently invoked functions.
        - Data: Reused data
3. Cache Design
    1. Valid Bit & Dirty Bit
        - Hit
            - Read: Return data
            - Write: Update in L1 & set DIRTY
        - Miss: Load data from lower-level memory &
            - If still empty blocks(not VALID): place there
            - Else: Select a victim block(LRU)
                - If victim block is DIRTY & VALID: write back
                    - If write-back or fetching causes any miss, repeat?
            - If Write: Update in L1 & set DIRTY (write-allocate)
    2. Block size == Line size,
    3. Tag Array: Hash table, reduce searching cost(approximately $O(1)$, but in fact $O(\log(n))$).
    4. Way-associativity: Tolerate some collision cases.
    5. | tag | set index | block offset | Why this design?
        1. A line should be consecutive -> offset left
        2. Adjacent line should correspond to different sets -> index middle

6. Formula:

$$Capacity = Associativity \times Blocksize \times \#Sets$$

$$\#bits(block\ offset) = \log(B)$$

$$\#bits(set\ index) = \log(S)$$

$$\#bits(tag) = address\_length - \log(S) - \log(B)$$

$$set\ index = (address/blocksize)\%S$$

Memory Optimizations

1. 3Cs
   - Compulsory: Cold start miss & First-time access
   - Capacity: Working set size(size of visited cache block before the next reuse of the current block) bigger than cache size(capacity). It also shows conflict but in a many-to-many manner.
   - Conflict: Collision in hash, mapping to the same set(associativity), even if there are enough space for working set.
2. 3Cs & A, B, C
   - Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower(need to compare more tags). Also need more power.
   - Without changing A & C, increasing B can reduce compulsory miss but potentially lead to more conflict misses(meanwhile reducing S). It will also make each cache miss slower(with constant bandwidth, more memory needs to be fetched).
   - Without changing A & B, increasing C can reduce capacity misses but make each cache hit slower(more bits). Also more costly and need more power.
3. Prefetch
   - Fetch next block of data before using the current one.
   - Hardware: The processor can keep track the distance between misses. It will overlap calculating and fetching(instead of exclusively waiting for data) by triggering the cache miss earlier to reduce miss penalty.
   - Software
     - x86 instruction: _mm_prefetch
     - gcc flag: -fprefetch-loop-arrays
   - But sometimes it won't bring benefit due to conflict miss or something else.
4. Stream buffer
   1. Captures the prefetched blocks
   2. Fully associative & Consult when miss & Retrieve if found
   3. Reduce compulsory misses & Avoid conflict misses triggered by prefetching
5. Miss cache
   1. Captures the missing blocks

    2. Fully associative & Consult when miss & Retrieve if found
    3. Reduce conflict misses
6. Victim cache
    1. Captures the evicted blocks
    2. Fully associative & Consult when miss & Swap if found
    3. Better than miss cache
7. Programmable alongside buffer
    1. Prefetching won't cause conflict misses
    2. Virtually add an associative set to frequently used data
    3. Need additional search time and power, but reduce power consumption overall(hit better than miss).
8. Multi-banked & non-blocking cache: Overlaps memory latency between different banks(but no benefit for query in same bank).
9. Early restart
    1. As soon as the request word of the block arrives, send it to CPU letting it continue execution, instead of waiting for the whole block finished.
    2. Useful with large blocks, but not a benefit if we want the next sequential word.
10. Critical word first & Warped fetch and requested word first
    - Request the exact word first from memory and send it to CPU immediately. Let CPU continue execution while filling the rest of the block.
    - Useful with large blocks
11. Write buffer
    - Write back always requires a period of time writing dirty data back to memory before fetching a new conflicted one.
    - Use a small piece of cache buffer to store the evicted data immediately and then it writes to memory asynchronously. Additionally, there are chances to merge adjacent write in write buffer.
12. Hardware optimizations
    1. Prefetch: Compulsory
    2. Write buffer: Miss penalty
    3. Bank/pipeline: Miss penalty
    4. Critical word first & Early restart: Miss penalty
13. Software optimizations
    1. Data layout: 3Cs
    2. Loop interchange: Conflict/Capacity miss
    3. Loop fission: Conflict miss(when cache has limited way associativity)
    4. Loop fusion: Capacity miss(when cache has enough way associativity)
    5. Blocking/Tiling: Capacity miss(smaller working set), conflict miss(fewer rows)
    6. Matrix transpose: Conflict miss

7. Use register whenever possible: Reduce memory access(but check commutativity and associativity carefully!)
8. Software prefetching
9. Sometimes with lower miss rate resulting higher CPI -> some vectorization issues
10. Lowest miss rate does not result fastest -> loop overhead causing high IC

VM

1. Purpose
    1. Abstraction for memory space(unified memory representation)
    2. Share physical memory with protection & isolation
    3. Allow applications work while memory is smaller than working set(different machines / multiple programs)
2. Mechanism: page
    1. Mapped by OS & hardware
    2. On demand
    3. Physical memory is a cache for virtual memory, storage is a lower level for physical memory(swap).
    4. Fully-associate, page size = block size, page table = tag array
3. Address translation
    1. Page table: Individual in each process, maintained by OS
    2. Multi-level page table
        - $\#PTE = \frac{page\ size}{sizeof(entry)}$
        - $\#levels = \lceil \log_{2^e ntries} \frac{memory\ space\ size}{page\ size} \rceil$, 6 for 64-bit 4KB machine and 4 for x86-64(48 meaningful bits).
    3. Data structure: B-tree (quick for both searching and inserting)
4. TLB
    1. Cache frequently used page table entries
    2. TLB + Virtual cache? Conflict issues(different programs use same virtual address; a program use aliasing virtual address pointing to the same physical address). L1 cache represents physical memory, ONLY physical address is suitable.
    3. Virtually indexed & physical tagged cache
        1. Offset unchanged, ignore it: |P_tag|P_index| & |V_tag|V_index|
        2. L1 cache uses physical address(both tag and index)
        3. Entries translate: virtual address -> physical address
        4. VM do not use V_index to diverse pages(fully associate, all in one set).
        5. Thus, VM can use index as PM's index. In other words, they can be consistent.
        6. Only |V_tag| -> |P_tag| is needed.
        7. Question: Will this harm fully-associate? This limits the design of cache.

5. Further caching
    1. TLB miss is expensive
    2. MMU have caches but not optimized for TLB(although TLB is almost the most important)
    3. Page Table Caches(AMD): Caches the address of Page Table Nodes
        1. Unified: Share same space
        2. Split: Each level get a private cache location
    4. Translation caching(Intel): Indexed by the prefix of virtual address (multi-level match)
        - Pros: Allowing each level lookup to perform independently, in parallel
        - Cons: Less space efficient

Pipeline

1. Lifetime of instruction
    1. IF
    2. ID
    3. EX
    4. MEM(read/write)
    5. WB(register)
    6. Update PC
2. Pipelining design
    1. ILP
    2. Clock synchronize among stages
    3. Pipeline register between stages
3. Throughput
    1. "single-cycle and sequential" -> "multi-stage but sequential" -> "multi-stage and pipelined"
    2. Shorter cycle time
    3. Theoretical CPI remains the same
4. Pipeline hazards
    1. Structural hazards: Resource conflicts(register file, memory)
    2. Control hazards: PC changed by instruction
    3. Data hazard: Instruction use data that has not yet generated/propagated
5. Solution
    1. Universal: Stall
    2. Structural hazards
        1. ID&WB: Write early, read late (in register file)
        2. MEM&IF: Split L1 cache
        3. WB&WB: Multibanks&non-blocking caches

Assignments

1. Latency(cycles) should be integer.
2. Miss rate lower than estimated: hardware prefetcher.
3. Miss penalty(cycles):

   $$\#\ of\ Cycles = \sum_{case=hit,miss,other} \#\ of\ Cycles_{case} = \#\ of\ Cycles_{other} + \sum_{case=hit,miss} CPI_{case} \times IC$$

   $$\begin{aligned}\sum_{case=hit,miss} CPI_{case} \times IC &= CPI_{hit} \times IC_{hit} + CPI_{miss} \times IC_{miss} \\ &= CPI_{hit} \times IC_{hit} + (CPI_{hit} + CPI_{miss\ penalty}) \times IC_{miss} \\ &= CPI_{hit} \times IC_{mem} + CPI_{miss\ penalty} \times IC_{miss} \\ &= CPI_{hit} \times IC_{mem} + CPI_{miss\ penalty} \times IC_{mem} \times Miss\ Rate \\ &= IC_{mem} \times (CPI_{hit} + CPI_{miss\ penalty} \times Miss\ Rate)\end{aligned}$$

   Note that $IC_{mem} = \#\ of\ Memory\ Accesses$, thus

   $$\#\ \textbf{of Cycles} = \#\ of\ Cycles_{other} + \#\ of\ Memory\ Accesses\ (CPI_{hit} + CPI_{miss\ penalty} \times Miss\ R$$

   For these cases above, $CPI_{hit}, \#\ of\ Cycles_{other}, \#\ of\ Memory\ Accesses$ are almost the same.

   $$CPI_{miss\ penalty} = \frac{\#\ of\ Cycles_2 - \#\ of\ Cycles_1}{\#\ of\ Memory\ Accesses_2 - \#\ of\ Memory\ Accesses_1}$$

   And the number of cycles can be devided by IC,

   $$CPI_{avg} = \frac{\#\ of\ Cycles_{other}}{IC_{total}} + \frac{\#\ of\ Memory\ Accesses}{IC_{total}} \times (CPI_{hit} + CPI_{miss\ penalty} \times Miss\ Ra$$

   Thus,

   $$\mathbf{\Delta CPI_{avg}} = \Delta CPI_{miss\ penalty} \times \frac{\#\ of\ Memory\ Accesses}{IC_{total}} \times Miss\ Rate$$

4. Conflict gap
   Number of sets:

   $$C = A \times BlockSize \times S \rightarrow S = \frac{C}{A \times BlockSize}$$

   The address gap between two closest conflicting cache blocks in an associative cache is

   $$Gap = BlockSize \times S$$

   Thus,

   $$B = A + Gap \times k, k \in \mathbb{Z}$$

More rigorously, any address within the corresponding block (any offset) is acceptable.

$$B = \lfloor \frac{A}{Linesize} \rfloor \times Linesize + Gap \times k + t, k \in \mathbb{Z}, t \in \mathbb{Z} \cap [0, Linesize)$$

5. L1&L2 TLB is in L1 cache, when L2 TLB miss, it happens to be a TLB miss.
6. Compiler will automatically align in structures
    1. Members are laid out in declaration order, with potential padding in between
    2. Each member is aligned to its alignment requirement (the largest factor from {1, 2, 4, 8})
    3. The total size of the structure is rounded up to a multiple of the largest alignment requirement among all members.
7. Static instructions: loop instructions in static code; dynamic instructions: loop instructions when running many times