15-740/18-740
Computer Architecture
Lecture 11: OoO Wrap-Up and Advanced Caching

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Chuck Thacker (Microsoft Research) Seminar
  - RARE: Rethinking Architectural Research and Education
  - October 7, 4:30-5:30pm, GHC Rashid Auditorium

- Ben Zorn (Microsoft Research) Seminar
  - Performance is Dead, Long Live Performance!
  - October 8, 11am-noon, GHC 6115

- Guest lecture Friday
  - Dr. Ben Zorn, Microsoft Research
  - Fault Tolerant, Efficient, and Secure Runtimes

Announcements

- Homework 2 due
  - October 10

- Midterm I
  - October 11
  - Sample exams online
  - You can bring one letter-sized cheat sheet

Last Time …

- Full Window Stalls
- Runahead Execution
- Memory Level Parallelism
- Memory Latency Tolerance Techniques
  - Caching
  - Prefetching
  - Multithreading
  - Out-of-order execution
- Improving Runahead Execution
  - Efficiency
  - Dependent Cache Misses: Address-Value Delta Prediction

OoO/Runahead Readings

- Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," HPCA 2003.
- Mutlu et al., "Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance," IEEE Micro Top Picks 2006.

- Zhou, Dual-Core Execution: "Building a Highly Scalable Single-Thread Instruction Window," PACT 2005.

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

Efficient Scaling of Instruction Window Size

- One of the major research issues in out of order execution

- How to achieve the benefits of a large window with a small one (or in a simpler way)?

  - Runahead execution?
    - Upon L2 miss, checkpoint architectural state, speculatively execute only for prefetching, re-execute when data ready
  - Continual flow pipelines?
    - Upon L2 miss, deallocate everything belonging to an L2 miss dependent, reallocate/re-rename and re-execute upon data ready
  - Dual-core execution?
    - One core runs ahead and does not stall on L2 misses, feeds another core that commits instructions

Runahead Execution (III)

- Advantages:
  + Very accurate prefetches for data/instructions (all cache levels)
    + Follows the program path
  + Uses the same thread context as main thread, no waste of context
  + Simple to implement, most of the hardware is already built in

- Disadvantages/Limitations:
  -- Extra executed instructions
  -- Limited by branch prediction accuracy
  -- Cannot prefetch dependent cache misses. Solution?
  -- Effectiveness limited by available "memory-level parallelism" (MLP)
  -- Prefetch distance limited by memory latency

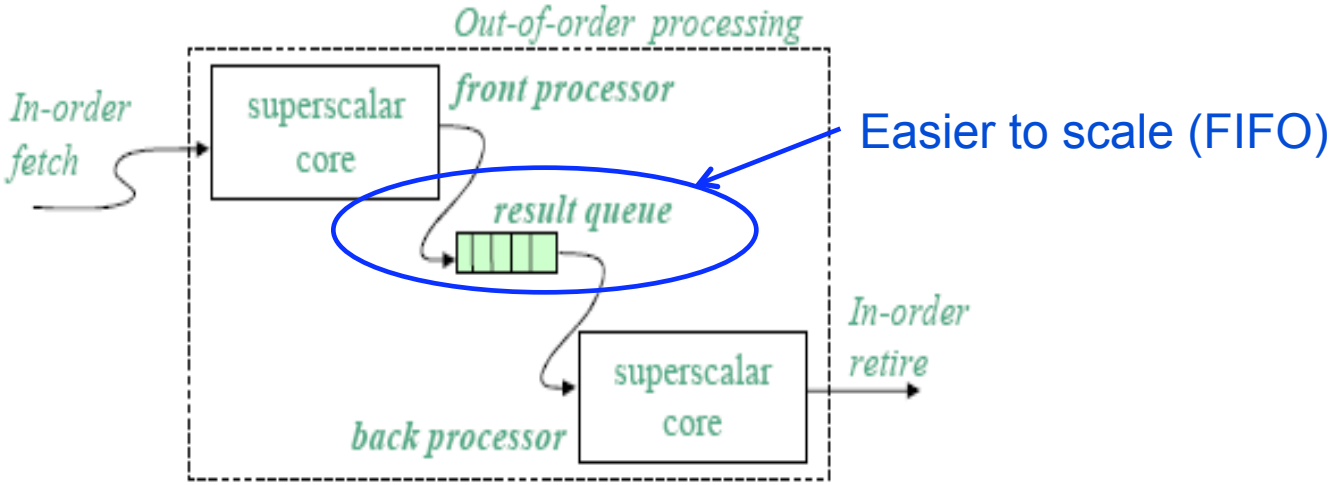
- Implemented in IBM POWER6, Sun "Rock"

Memory Latency Tolerance Techniques

- Caching [initially by Wilkes, 1965]
  - Widely used, simple, effective, but inefficient, passive
  - Not all applications/phases exhibit temporal or spatial locality
- Prefetching [initially in IBM 360/91, 1967]
  - Works well for regular memory access patterns
  - Prefetching irregular access patterns is difficult, inaccurate, and hardware-intensive
- Multithreading [initially in CDC 6600, 1964]
  - Works well if there are multiple threads
  - Improving single thread performance using multithreading hardware is an ongoing research effort
- Out-of-order execution [initially by Tomasulo, 1967]
  - Tolerates cache misses that cannot be prefetched
  - Requires extensive hardware resources for tolerating long latencies

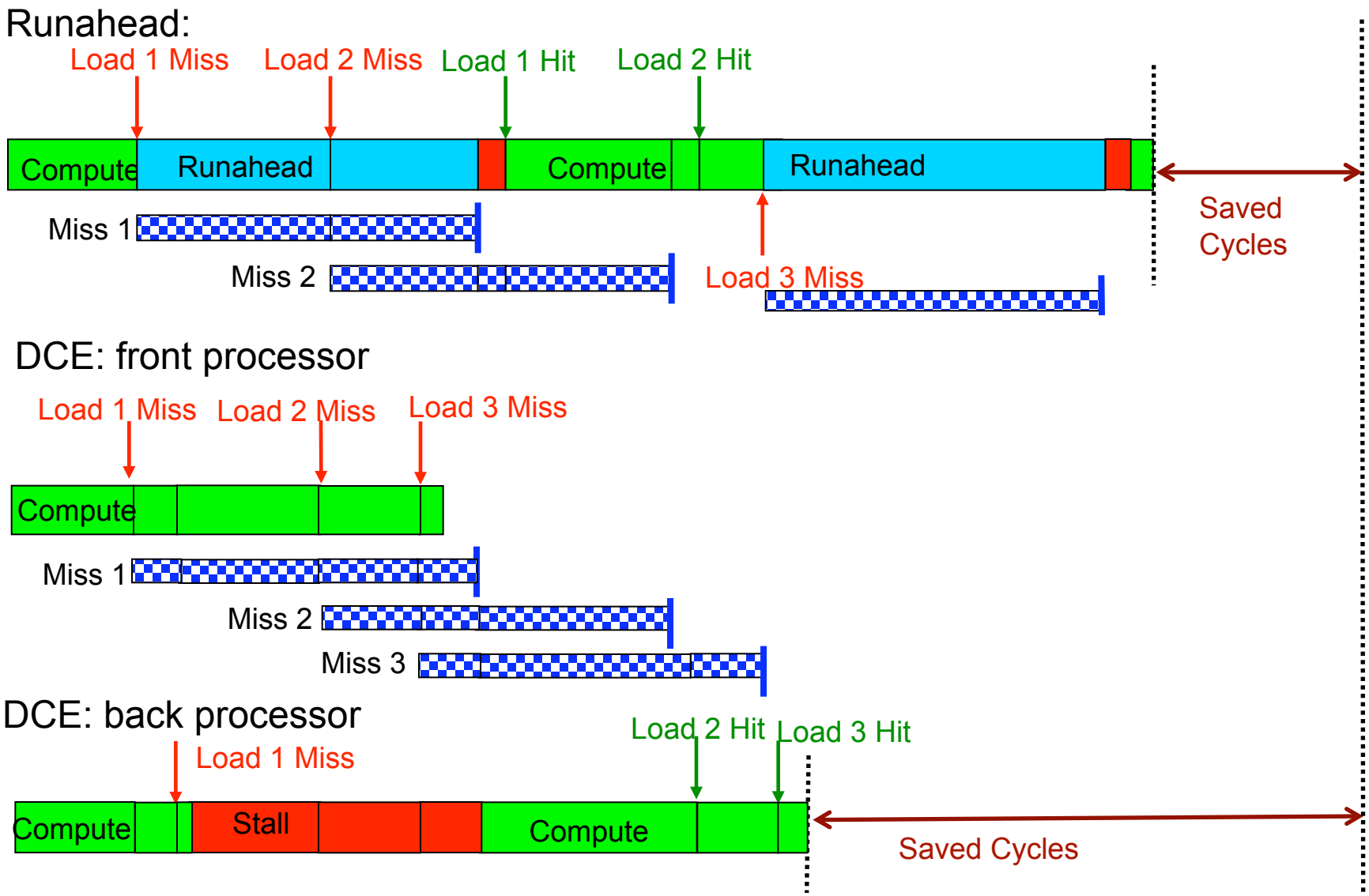
Runahead and Dual Core Execution

- Runahead execution:
  - + Approximates the MLP benefits of a large instruction window (no stalling on L2 misses)
  - -- Window size limited by L2 miss latency (runahead ends on miss return)
- Dual-core execution:
  - + Window size is not limited by L2 miss latency
  - -- Multiple cores used to execute the application



- Zhou, Dual-Core Execution: "Building a Highly Scalable Single-Thread Instruction Window," PACT 2005.

Runahead and Dual Core Execution



Handling of Store-Load Dependencies

- A load's dependence status is not known until all previous store addresses are available.

- How does the OOO engine detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
  - Option 1: Assume load independent of all previous stores
  - Option 2: Assume load dependent on all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store

Store Buffer Design (I)

- An age ordered list of pending stores
  - un-committed as well as committed but not yet propoagated into the memory hierarchy
- Two purposes:
  - Dependency detection
  - Data forwarding (to dependent loads)
- Each entry contains
  - Store address, store data, valid bits for address and data, store size
- A scheduled load checks whether or not its address overlaps with a previous store

Store Buffer Design (II)

- Why is it complex to design a store buffer?

- Content associative, age-ordered, range search on an address range
  - Check for overlap of [load EA, load EA + load size] and [store EA, store EA + store size]
    - EA: effective address

- A key limiter of instruction window scalability
  - Simplifying store buffer design or alternative designs an important topic of research

Memory Disambiguation (I)

- Option 1: Assume load independent of all previous stores

  + Simple and can be common case: no delay for independent loads

  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 2: Assume load dependent on all previous stores

  + No need for recovery

  -- Too conservative: delays independent loads unnecessarily

- Option 3: Predict the dependence of a load on an outstanding store

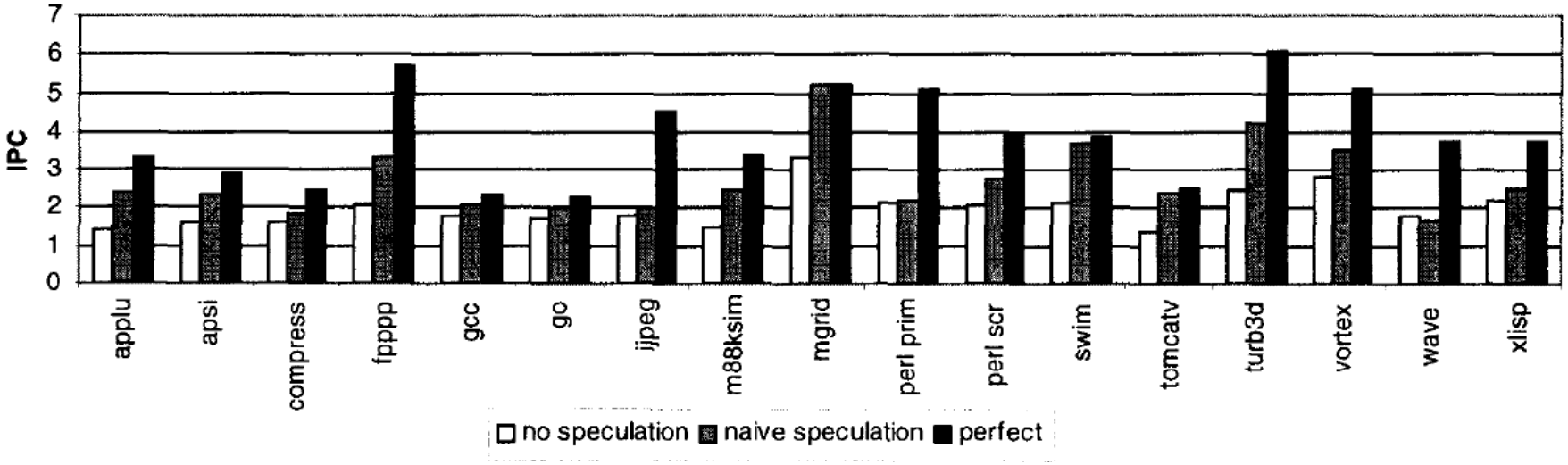
  + More accurate. Load store dependencies persist over time

  -- Still requires recovery/re-execution on misprediction

  - Alpha 21264 : Initially assume load independent, delay loads found to be dependent
  - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
  - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

Memory Disambiguation (II)

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.



- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Speculative Execution and Data Coherence

- Speculatively executed loads can load a stale value in a multiprocessor system
  - The same address can be written by another processor before the load is committed → load and its dependents can use the wrong value

- Solutions:

1. A store from another processor invalidates a load that loaded the same address

   -- Stores of another processor check the load buffer

   -- How to handle dependent instructions? They are also invalidated.

2. All loads re-executed at the time of retirement

Open Research Issues in OOO Execution (I)

- Performance with simplicity and energy-efficiency
- How to build scalable and energy-efficient instruction windows
  - To tolerate very long memory latencies and to expose more memory level parallelism
  - Problems:
    - How to scale or avoid scaling register files, store buffers
    - How to supply useful instructions into a large window in the presence of branches
- How to approximate the benefits of a large window
  - MLP benefits vs. ILP benefits
  - Can the compiler pack more misses (MLP) into a smaller window?
- How to approximate the benefits of OOO with in-order + enhancements

Open Research Issues in OOO Execution (II)

- OOO in the presence of multi-core

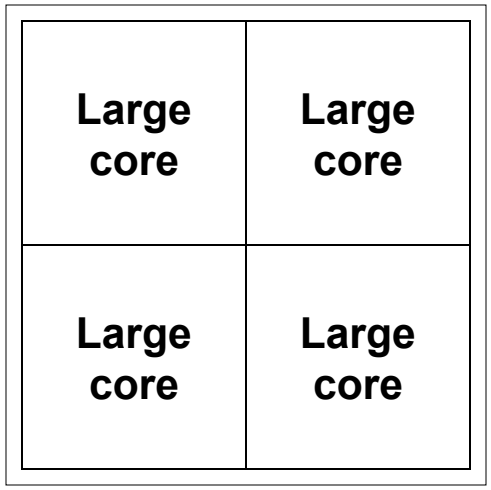
- More problems: Memory system contention becomes a lot more significant with multi-core
  - OOO execution can overcome extra latencies due to contention
  - How to preserve the benefits (e.g. MLP) of OOO in a multi-core system?

- More opportunity: Can we utilize multiple cores to perform more scalable OOO execution?
  - Improve single-thread performance using multiple cores

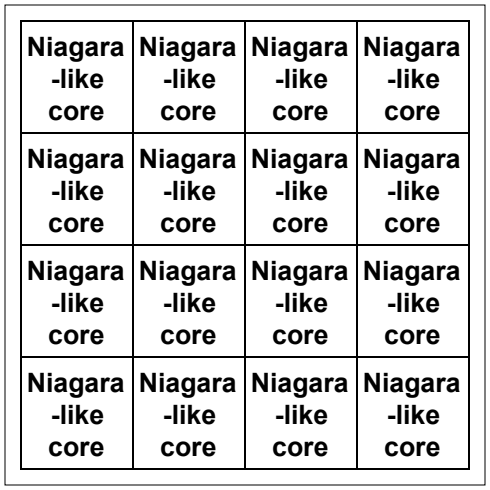
- Asymmetric multi-cores (ACMP): What should different cores look like in a multi-core system?
  - OOO essential to execute serial code portions

Open Research Issues in OOO Execution (III)

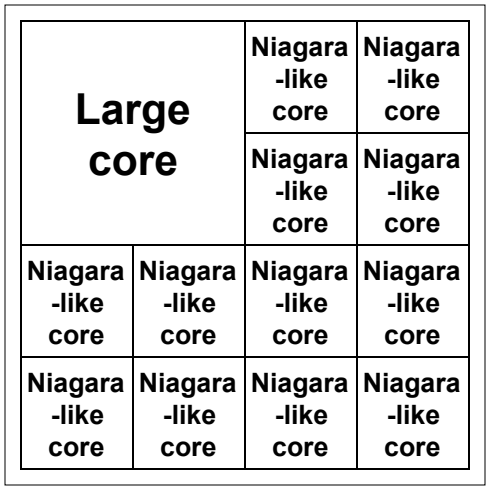
- Out-of-order execution in the presence of multi-core
- Powerful execution engines are needed to execute
  - Single-threaded applications
  - Serial sections of multithreaded applications (remember Amdahl's law)
  - Where single thread performance matters (e.g., transactions, game logic)
  - Accelerate multithreaded applications (e.g., critical sections)



"Tile-Large" Approach



"Niagara" Approach



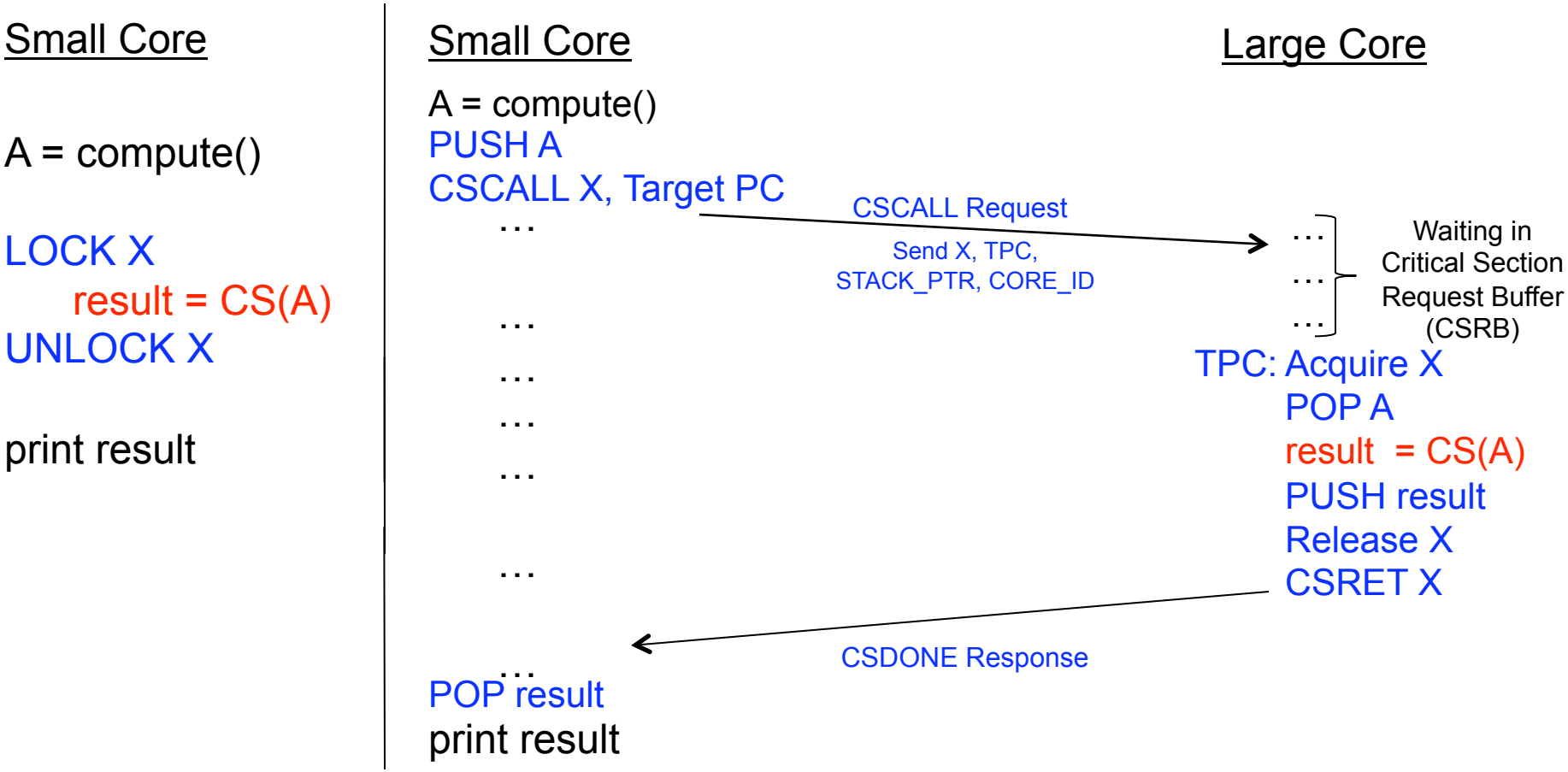
ACMP Approach

Asymmetric vs. Symmetric Cores

- Advantages of Asymmetric
  - + Can provide better performance when thread parallelism is limited
  - + Can be more energy efficient
    - + Schedule computation to the core type that can best execute it

- Disadvantages
  - - Need to design more than one type of core. Always?
  - - Scheduling becomes more complicated
    - - What computation should be scheduled on the large core?
    - - Who should decide? HW vs. SW?
  - - Managing locality and load balancing can become difficult if threads move between cores (transparently to software)
  - - Cores have different demands from shared resources

Accelerated Critical Sections (ACS)



- Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," APSLOS 2009.

Advanced Caching

Topics in (Advanced) Caching

- Inclusion vs. exclusion, revisited
- Handling writes
- Instruction vs. data
- Cache replacement policies
- Cache performance
- Enhancements to improve cache performance
- Enabling multiple concurrent accesses
- Enabling high bandwidth caches

Readings

- Required:
  - Hennessy and Patterson, Appendix C.1-C.3
  - Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," ISCA 1990.
  - Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

- Recommended:
  - Seznec, "A Case for Two-way Skewed Associative Caches," ISCA 1993.
  - Chilimbi et al., "Cache-conscious Structure Layout," PLDI 1999.
  - Chilimbi et al., "Cache-conscious Structure Definition," PLDI 1999.

Inclusion vs. Exclusion in Multi-Level Caches

- Inclusive caches
  - Every block existing in the first level also exists in the next level
  - When fetching a block, place it in all cache levels. Tradeoffs:

    -- Leads to duplication of data in the hierarchy: less efficient

    -- Maintaining inclusion takes effort (forced evictions)

    + But makes cache coherence in multiprocessors easier
      - Need to track other processors' accesses only in the highest-level cache
- Exclusive caches
  - The blocks contained in cache levels are mutually exclusive
  - When evicting a block, do you write it back to the next level?

  + More efficient utilization of cache space

  + (Potentially) More flexibility in replacement/placement

  -- More blocks/levels to keep track of to ensure cache coherence; takes effort
- Non-inclusive caches
  - No guarantees for inclusion or exclusion: simpler design
  - Most Intel processors

Maintaining Inclusion and Exclusion

- When does maintaining inclusion take effort?
  - L1 block size < L2 block size
  - L1 associativity > L2 associativity
  - Prefetching into L2
  - When a block is evicted from L2, need to evict all corresponding subblocks from L1 → keep 1 bit per subblock in L2
  - When a block is inserted, make sure all higher levels also have it

- When does maintaining exclusion take effort?
  - L1 block size != L2 block size
  - Prefetching into any cache level
  - When a block is inserted into any level, ensure it is not in any other

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Serial tag and data access

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter. Can you exploit this fact to improve hit rate in the second level cache?