Carnegie Mellon University
Electrical and Computer Engineering Department

18-742 Spring 2025
Midterm 1 - Practice Questions

**Note: The actual midterm will have 9 questions.
This example has only 4 questions.**

(Also, these questions overlap with one another more than the actual midterm will...)

INSTRUCTIONS:

- Closed book. No notes. 80 minutes.
- If you find a question ambiguous, be sure to write down any assumptions you make.
- Be clear and concise.
- **Answer 7 of the 9 questions.** We will grade the **first 7 questions answered.** If you start to answer a question but decide later that you don't want it graded, draw a big **X** across the page.
- For your 7 graded questions, it is better to partially answer a question than to not attempt it at all.
- All questions have the same weight.
- You may find some questions easier than others.
- If you need more space for your answer, there are two blank pages at the end. Be sure to clearly label any answer on these blank pages with BOTH the question number (1-9) and the part number (a,b,c).

QUESTIONS:

Parallel Computer Architectures ("Amdahl's Law, Cache Coherence, Memory Consistency")

1. Multicore processors are prevalent in today's processor market. Within a multicore processor, various cores are capable of executing distinct threads of a program simultaneously. This architectural design mandates the implementation of effective cache coherence models. These models are crucial for ensuring the correct execution of programs.

   (a) Discuss two conditions under which Amdahl's Law may *overestimate* the potential performance improvements in parallel computing systems.

   Solution: Amdahl's Law provides a theoretical maximum speedup for parallelizable workloads but has several limitations in predicting real-world performance:

   1. It does not account for overheads introduced by parallelization, such as synchronization, communication, and increased memory access times, which can significantly affect performance.
   2. It assumes infinite scalability of the parallel portion, which is unrealistic due to hardware limitations, such as memory bandwidth limitations.

   (b) Discuss a condition under which Amdahl's Law may *underestimate* the potential performance improvements in parallel computing systems.

   Solution: Amdahl's Law may underestimate parallel processing performance because it assumes a fixed problem size. In reality, as more processors are added, the problem size can also increase—a concept known as weak scaling. This approach, unlike Amdahl's fixed perspective, allows for processing larger datasets more efficiently with additional processors. Thus, Amdahl's Law doesn't fully capture the potential performance gains in real-world applications where scaling the problem size with the number of processors is beneficial.

   (c) Consider the following code segments intended to run concurrently on two different processors. Processor 1 executes Code Segment 1, and Processor 2 executes Code Segment 2. Initially, shared variables `A` and `B` are both set to 0. The system does not inherently guarantee the order of memory operations across processors without explicit synchronization instructions. Can an efficient cache coherence protocol ensure correct execution of this program? Please explain your answer.

   Code Segment 1 (Processor 1):

   ```
   A = 1;
   B = 1;
   ```

   Code Segment 2 (Processor 2):

   ```
   while (B == 0);
   int readA = A;
   ```

   Solution: Cache coherence primarily ensures the integrity of data across multiple caches, guaranteeing that all processors access the most recent version of shared data. It does not directly manage the *ordering* of memory requests, which is crucial for the correct execution of this program.

Further Explanation (Not Required for Full Credit):

The correct execution of this program can be ensured through the adoption of a hardware memory consistency model, such as Sequential Consistency, or by implementing FENCE instructions in the software, as below:

Code Segment 1 (Processor 1):

```
        A = 1;
        FENCE;
        B = 1;
```

Code Segment 2 (Processor 2):

```
        while (B == 0);
        FENCE;
        int readA = A;
```

The FENCE instruction after writing to A in Code Segment 1 ensures that the write to A is visible to all processors before the write to B occurs. This ordering guarantees that if Processor 2 sees the updated value of B (i.e., B == 1), the write to A (i.e., A = 1) must have been committed and thus will be visible when read.

Similarly, the FENCE instruction in Code Segment 2 ensures that the loop waiting for B to change to 1 completes and all the effects of previous memory operations (specifically, the read of B) are visible before proceeding to read A. This synchronization prevents the possibility of reading a stale value of A.

Cache Coherence **("Parallel Programming, Coherence Protocol, Directory Scalability")**

2. Parallel processing requires the implementation of efficient coherence mechanisms to guarantee the accurate execution of parallel programs.

   (a) Consider a scenario where two processors, P1 and P2, are working on a shared data structure that is cached in both of their local caches. Describe how a lack of cache coherence would affect the execution of a program that requires both processors to update the shared data structure. How would cache coherence mechanisms mitigate these effects?

   Solution:
   Without Cache Coherence: If P1 modifies the shared data structure, the change might not be immediately visible to P2 if P2 has a cached copy of the data. As a result, P2 might read stale data and perform incorrect computations or updates. Similarly, if both P1 and P2 attempt to update the shared data structure simultaneously, their updates might overwrite each other without proper synchronization, leading to data loss or incoherence.

   With Cache Coherence Mechanisms: Cache coherence mechanisms would ensure that once P1 updates the shared data structure, P2's cache is either invalidated or updated with the new value before P2 can read the data again. This ensures that P2 always works with the most recent data. If both P1 and P2 attempt to update the shared data simultaneously, cache coherence protocols would serialize these updates in some order, ensuring that both updates are applied consistently without data loss.

   (b) In the paper "Why On-Chip Cache Coherence is Here to Stay," which method is recommended to scale the *directory area* for cache coherence? Explain how.
   (A) Data sharing
   (B) Hierarchical design
   (C) Inexact sharer tracking
   (D) Metadata compression

   Solution: Hierarchical cache design promotes area scalability by structuring caches in multi-level hierarchies, thus minimizing the overall footprint. This approach reduces the need for large, flat directories by utilizing shared cluster caches, effectively optimizing the use of silicon area.

   (c) According to the same paper, how does "inclusion" facilitate scaling on-chip cache coherence?

   Solution: Inclusion in cache coherence protocols means that the shared cache contains a superset of the data in all private caches. This simplifies coherence management by centralizing the tracking of which cores have cached copies of memory blocks. Inclusion facilitates scaling by making it easier to determine which caches might contain copies of a particular memory block, thus reducing the overhead associated with cache coherence maintenance as core counts increase.

Data Prefetching **("Memory Access Prediction, Data Access Patterns, Memory Latency")**

3. Data prefetching is the process of predicting and prefetching future memory references before they are explicitly requested by applications, with the goal of hiding the memory access latency.

   (a) Discuss two potential drawbacks of data prefetching.

   Solution: Two drawbacks from the list below for full credit.

   1. Cache Pollution: Prefetching incorrect data may lead to the eviction of useful data from the cache, reducing cache efficiency.
   2. Memory Bandwidth Pollution: Excessive prefetch requests can increase contention on the memory bus, potentially degrading system performance.
   3. Metadata Storage: Implementing prefetching mechanisms might require significant storage for history and metadata, leading to large area occupation on the chip.
   4. Energy Waste: Incorrect prefetch requests waste energy by unnecessarily activating the memory subsystem.

   (b) As discussed in class, "helper thread prefetching" constructs a *helper* thread which performs data prefetching for the *main* thread. Compare Bingo prefetching with helper thread prefetching, highlighting two advantages and two disadvantages of Bingo relative to helper thread prefetching.

   Solution: Two advantages and two disadvantages from the lists below for full credit.
   Advantages:

   1. Bingo does not need threads dedicated to prefetching
   2. Bingo does not need a pre-execution context dedicated to prefetching
   3. Bingo is capable of advancing significantly ahead in the execution sequence
   4. Bingo has the ability to predict and prefetch cache misses that are dependent on earlier misses (although this is not its primary function)
   5. Bingo maintains its predictive accuracy regardless of the performance of the branch predictor

   Disadvantages:

   1. Helper thread prefetching can predict memory accesses that are typically challenging to predict
   2. Helper thread prefetching offers flexibility by allowing the use of distinct prefetching threads tailored to specific application needs
   3. Unlike Bingo, helper thread prefetching leverages available idle contexts, thus avoiding significant additional area requirements

   (c) Why is the consolidation of metadata tables effective in reducing Bingo's storage overhead?

Solution: The rationale lies in the significant *redundancy* in footprints among the metadata tables—consolidation efficiently eliminates these overlaps, thus diminishing the storage demands of the prefetching method.

Runahead Execution (**“Memory-level Parallelism, Helper Thread Prefetching”**)

4. Runahead execution allows memory-level parallelism (MLP) to break past reorder buffer (ROB) limits, by speculatively ignoring dependencies and continuing execution of the thread upon a miss in order to issue prefetch requests.

   (a) As discussed in class and the runahead execution paper, another method for prefetching challenging memory references is “helper thread prefetching,” which constructs a *helper* thread to perform data prefetching for the *main* thread. Mention one benefit of runahead execution in comparison to helper thread prefetching.

   Solution: One of the following items for full credit.

   1. Uses the same thread context as main thread, no waste of context
   2. No need to construct a pre-execution thread

   (b) Runahead execution makes use of a “runahead cache.” What is its purpose and how does it function?

   Solution: Runahead cache enables data communication through memory in runahead mode. A dependent load reads its data from the runahead cache.

   (c) When compared to hardware prefetchers like Bingo, list two advantages and two disadvantages of runahead execution.

   Solution: Two advantages and two disadvantages from the lists below for full credit.
   Advantages:

   1. Runahead execution issues very accurate prefetch requests
   2. Runahead execution operates with low overhead, not requiring large metadata tables
   3. Runahead execution contributes to training the branch predictor

   Disadvantages:

   1. Runahead execution involves executing extra instructions, which may be power-inefficient
   2. Runahead execution is limited by branch prediction accuracy
   3. Runahead execution cannot prefetch *dependent* cache misses
   4. Runahead execution’s effectiveness is limited by available MLP
   5. Runahead execution’s prefetch distance (how far ahead to prefetch) limited by memory latency