Implementing Cache Coherence Protocols in Multicore Systems

15-418/15-618: Final Project Report

TEAM: Bharathi Sridhar (bsridha2), Tanvi Daga (tdaga)

URL: https://github.com/bharathi2203/418-Directory-Based-Cache-Coherence/tree/main

PROJECT SUMMARY

In this project, we are implementing and analyzing advanced cache coherence protocols for multicore systems. We focused on the directory-based scheme, its variants like the limited pointer and sparse directory schemes, and their effects on system performance. We measured critical performance metrics including read and write hits and misses, cache evictions, and interconnect traffic. These implementations aim to provide insights into achieving efficient data consistency and enhanced performance in multicore environments. We will perform a detailed analysis of the performance of each protocol across many traces with different cache access patterns and hardware specifications.

BACKGROUND

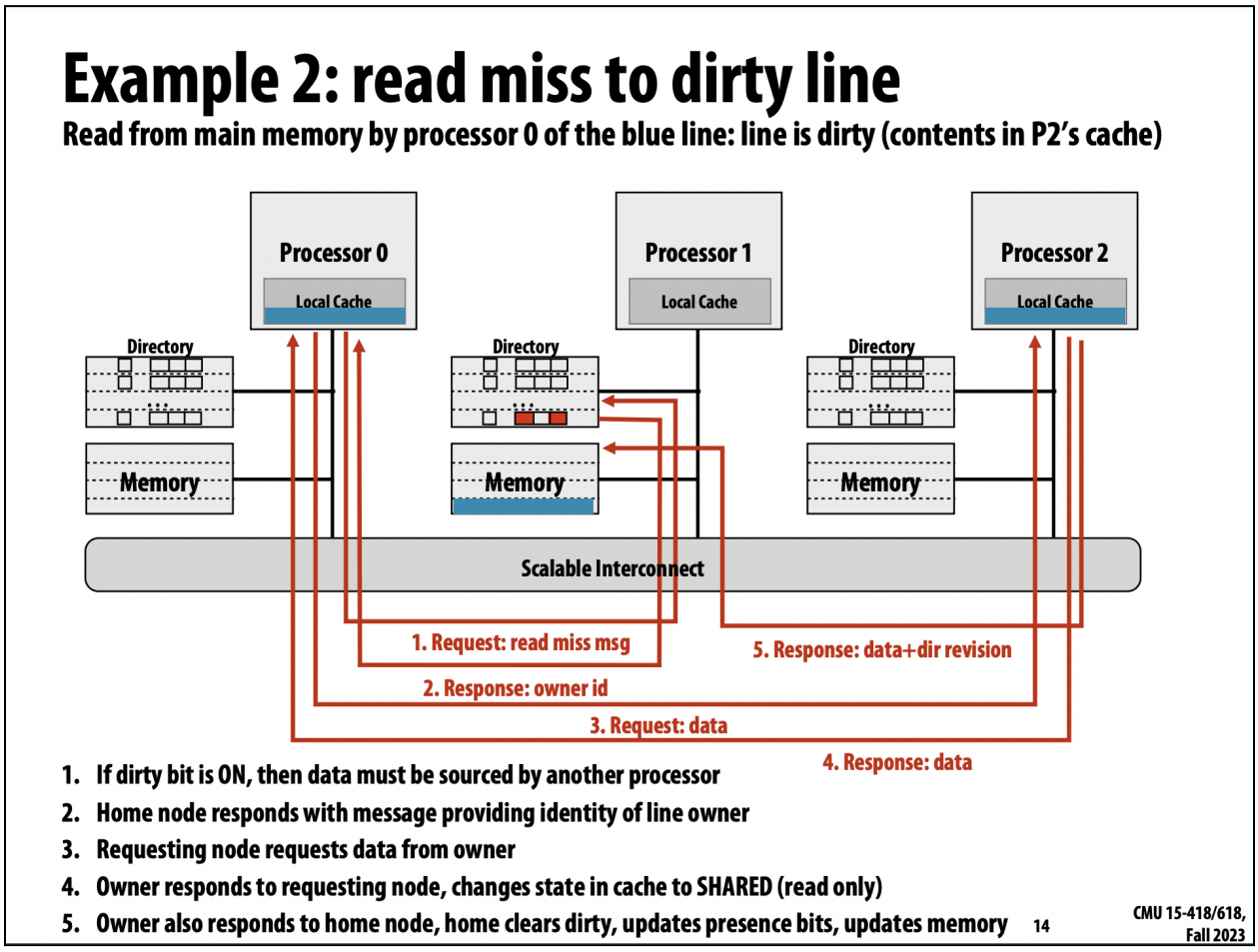
In multicore systems, dependencies are created when different cores often access and modify shared data. Managing these dependencies effectively to maintain data consistency and coherence without depreciating performance is a significant challenge. We need to ensure that when one core modifies data, other cores relying on this data are notified or updated efficiently. Additionally, the time taken to access memory can vary depending on the memory's location relative to a specific processor. This non-uniformity adds another layer of complexity to designing cache coherence protocols, as they must account for varying access times and their impact on overall system performance. A good cache coherence protocol must be robust and must be able to perform well against different applications that might have very different memory access patterns. Some may demonstrate high locality of reference, where repeated accesses to the same data occur, while others might have random access patterns.

One set of protocols discussed in lecture are the MSI, MESI, and MOESI snooping protocols which implement different strategies to maintain cache coherence by ensuring that multiple caches have consistent views of shared data. Overall, snooping based cache coherence protocols are not scalable as they rely on broadcasting which generates a lot of interconnect traffic. However in directory based cache coherence, each processor tracks the state of a cache line in a directory entry and uses point-to-point messages between requesting and home node to maintain cache coherence. This helps avoid broadcasting. For our project, we aim to implement a few other cache coherence protocols:

Directory based scheme

Directory-based cache coherence schemes provide a system for managing the state of cache lines which simplifies communication between processors and memory in a multiprocessor system. Directories usually maintain a record of which processors have cached or modified a particular memory line, which removes the need to broadcast commands and thus reduces bus traffic while still maintaining coherence. This model is particularly beneficial in machines where directories are distributed along with the corresponding memory segment as there is a reduction in unnecessary data movement.

When a processor requires data that it does not have in its cache (miss), it first checks the home node's directory to obtain the data. If the data is unmodified elsewhere, it is fetched directly from memory. However, if the data has been modified and exists in a different cache, the home node directs the requesting node to the owner for the latest data. Write operations must invalidate other processors' caches accordingly to maintain coherence.

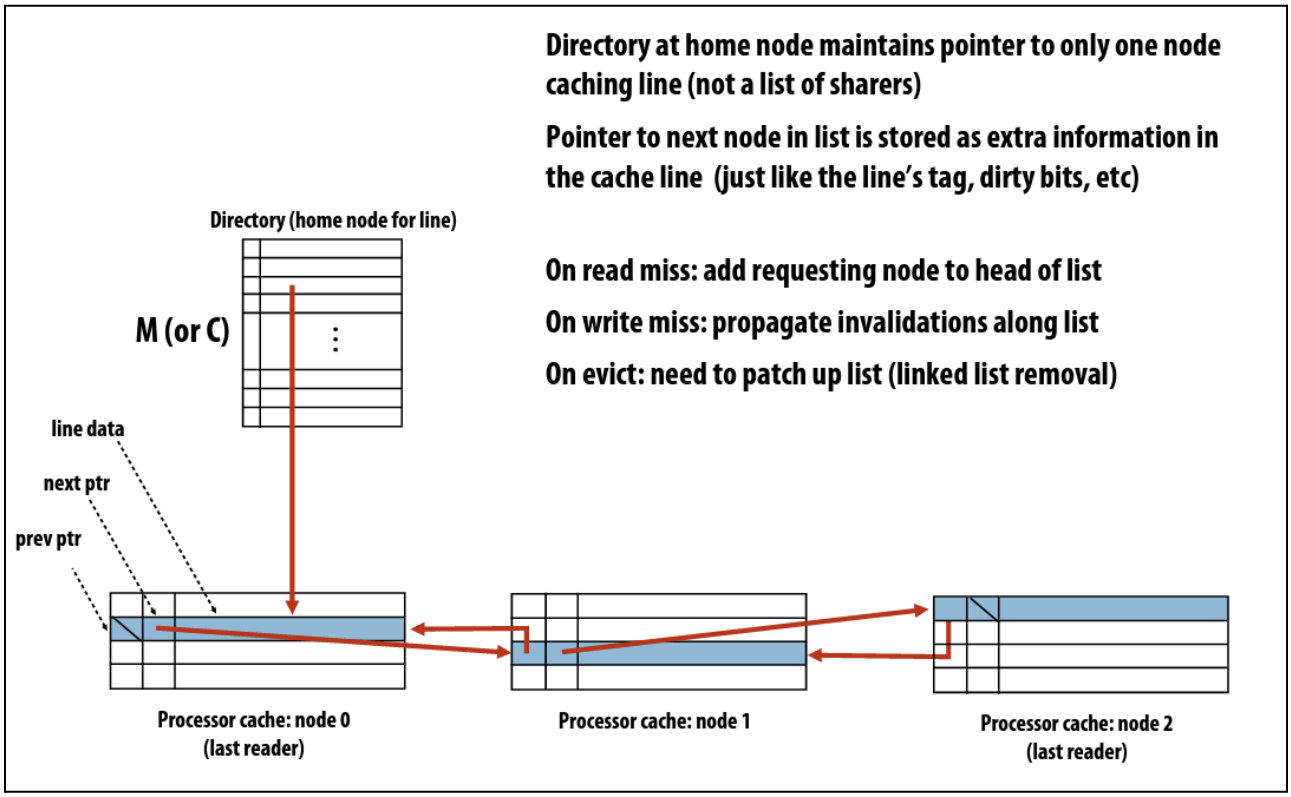


Limited pointer scheme

Limited pointer schemes in directory-based cache coherence improve on some challenges posed by the distributed directory scheme described above, especially in high processor count systems. Instead of storing a bit for every processor per memory line, we store only a small subset of pointers to keep track of caches that contain a copy of the cache line. One challenge for this scheme is to manage the overflow case where the number of sharers exceeds the pointers. However, the primary benefit of this scheme is to optimize for the most common cases, where there are few sharers.

Sparse directory Scheme

Sparse directory-based schemes in cache coherence protocols further optimize the memory requirement by reducing the directory size to the cache size. This significantly reduces the memory requirement as caches are often much smaller than the available memory which leaves most of the directory empty anyway. While this scheme has higher complexity due to some additional work required for managing evictions and the linked list of nodes sharing each cache line, the reduced memory overhead is a beneficial trade-off for most real world scenarios.



APPROACH

Technologies, languages, and APIs used

The codebase simulates a directory-based cache coherence protocol in a multiprocessor NUMA system using C, incorporating standard libraries and custom-defined data structures for modeling caches, directories, interconnect message queues, and tracking performance metrics like hits, misses, evictions and interconnect traffic.

List of deliverables and platforms used

We tested our implementation on the GHC machines. To run our simulator, run "`make`" in the src/ directory, and then run `./directory <trace file>`, which prints the simulation stats on completion. Each scheme is on a different branch in the repo (distributed directory, limited pointer, sparse directory).

IMPLEMENTATION SUMMARY

Key Data Structures and Operations

The codebase centered around variations of a distributed directory based cache coherence system for multicore processors, where **`cache_t`** represents individual caches, including processor ID, set and block information, and performance metrics like hit, miss and eviction counts. All caches use a Least Recently Used (LRU) line replacement / eviction policy. Its scope includes reducing memory overhead across directory based schemes and improving efficiency in handling cache operations. Each **`cache_t`** consists of **`set_t`** structures, each containing multiple **`line_t`** structures representing cache lines, with details like tag, validity, dirtiness, and state (**`INVALID`**, **`SHARED`**, **`EXCLUSIVE`**, **`MODIFIED`**).

The **`directory_t`** structure maintains a directory of **`directory_entry_t`**, keeping track of cache line states (**`UNCACHED`**, **`SHARED`**, **`EXCLUSIVELY_MODIFIED`**) and ownership across processors. Inter-processor communication is handled through messages (**`message_t`**) enqueued in a message-passing system (**`interconnect_t`**) that contains incoming and outgoing message queues. This allows message-passing between caches and memory, and handling operations like read/write requests, invalidations, and acknowledgments. All the above mentioned data structures and function declarations can be found in the include folder of the repository.

Distributed Directory Scheme Implementation

The distributed directory implementation uses the above mentioned data structures representing cache lines (**`line_t`**), sets of lines (**`set_t`**), and the entire cache (**`cache_t`**) for each processor. The directory (**`directory_t`**) maintains the state of each block in memory across caches to ensure coherence. The system operates on a trace of memory access operations, maintaining statistics (**`csim_stats_t`**, **`interconnect_stats_t`**) to evaluate performance metrics like hits, misses, and evictions, crucial for understanding and optimizing cache coherence mechanisms.

Limited Pointer Scheme Implementation

We updated the above distributed directory scheme to implement a limited pointer scheme with a subset of the processors that a directory entry can point to. Key changes compared to the previous version include:

- The **`directory_entry_t`** structure has an **`existsInCache`** array with a fixed size of **`LIM_PTR_DIR_ENTRIES`**, rather than a dynamic presence bit for each processor.
- Directory entry management functions, such as `addProcToDirEntry` and `removeProcFromDirEntry`, have been added to manage the limited pointers efficiently, including the logic to handle eviction if the limit is reached.
- Directory updates now require invalidating cache lines only for processors present in the limited directory entries, instead of checking and invalidating all processors.

Spare Directory Scheme Implementation

In this implementation, we modified the directory structure to use a linked list to manage the presence bits of the cache lines. Here are some of main the differences from the previous schemes discussed:

- Instead of having a fixed array of processor presence bits, the sparse directory entry (**`directory_entry_t`**) now includes a linked list (**`line_t *head`**) to represent the caches that have a copy of the line. This allows the directory to handle an arbitrary number of processors sharing the line without defining a fixed-size array.
- Each directory entry now contains a **`tag`** field, which was unnecessary in the previous schemes. This field is required to identify which cache line the directory entry maps to.
- When a line is added to the cache, it is inserted to the directory entry's linked list at the head (so we maintain some notion of LRU here as well).
- The `updateDirectory` function now loops through the linked list of cache lines to invalidate (and thus remove from the linked list) other copies when a line is modified (state `DIR_EXCLUSIVE_MODIFIED`). Only the linked processors need to be invalidated (and have their lines removed if present) , not all processors.

Implementation Process

Once we had a basic directory, cache and interconnect structure implemented we began implementing the actual communication required for the different cache coherence protocols. We began by implementing a centralized directory based approach but realized that this implementation would not be very performant as the number of processors and memory modules increases, a central directory can become a bottleneck. Each memory access by any processor requires interaction with the central directory, which can limit the system's ability to scale. So we decided that despite distributed directories being more complex to implement and manage it would be a better design choice.

TESTING & RESULTS

List of goals:

1. Implementing a distributed directory based approach.
2. Implementing limited pointer scheme.
3. Implementing sparse directory scheme.
4. Benchmark performance.

Benchmark & Analysis

[1] Memory vs Number of Processors

The Distributed Directory based Approach has the highest memory requirement. The directory has M entries, where M is the number of cache lines in memory, and each entry stores the cache line state (enum- 4 bytes) + owner processor ID (int- 4 bytes) + P presence bits vector (p bytes)). The total size of the directory is M*(8+n) bytes. As n grows, the size of the directory increases.

The pointer based scheme has a lower memory requirement. The directory has M entries, where M is the number of cache lines in memory, and each entry stores the cache line state (enum- 4 bytes) + owner processor ID (int- 4 bytes) + vector of 8 ints containing the IDs of processors that have cached this line (8*4=32 bytes). The total size of the directory using the pointer based scheme is M*40 bytes.

The sparse directory has the lowest memory requirement. The directory has C entries, where C is the number of lines in the cache, and each entry stores the cache line state (enum- 4 bytes) + owner processor ID (int- 4 bytes) + tag (4 bytes) + pointer to the head of a linked list of cache lines (8 bytes)). The total size of the directory is C*20 bytes.

Benchmarking Various Cache Access Patterns

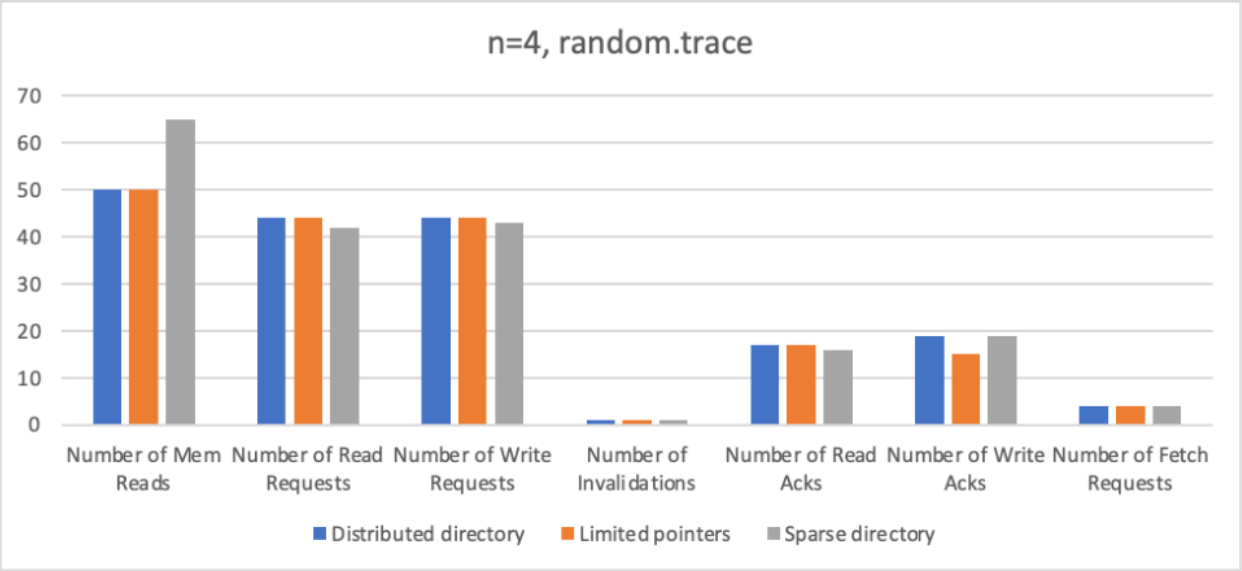
In the following graphs, we plot various metrics that capture the interconnect traffic using the different messages used in the implementation of this protocol. These are specific to our implementations of the above mentioned directory based cache coherence schemes, and they may be optimized for specific performance requirements or hardware specifications. These include:

- Number of Memory Requests: Number of access to memory to get data.
- Number of Read requests: Number of "load" requests processed overall across all caches, including read requests forwarded from other processor's cache.
- Number of Write requests: Number of "store" requests processed overall across all caches, including read requests forwarded from other processor's cache.
- Number of invalidations: Number of cache block invalidations that needed to be requested to maintain cache coherence given a specific set of read/write requests.
- Number of Read Acknowledgments: In specific forwarding based read requests, we use an acknowledgment message to ensure that a transaction has been completed and that the processor that requested data has acquired it and updated its cache.
- Number of Write Acknowledgments: In specific forwarding based write requests, we use an acknowledgment message to ensure that a transaction has been completed and that the processor that requested data has acquired it and updated its cache.
- Number of Fetch Requests: Number of requests to acquire a cache block from a cache or processor that is not the same as the requesting cache or processor which requires the requesting processor to "fetch" the corresponding data into its own cache.

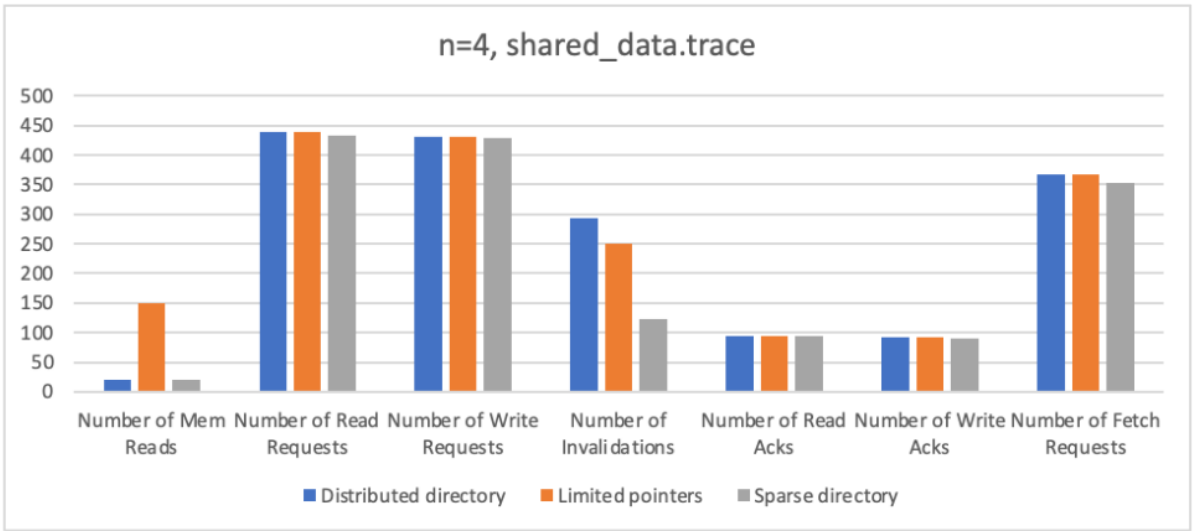
[1] Scheme vs Metrics for each trace file [Number of Processors = 4]

For the following graphs, we used a distributed directory based cache simulation, a limited pointer scheme based simulation, and a sparse directory scheme based simulation each with the same cache size per processor and the limited pointer scheme has a processor-membership sublist of size 2 within each entry of the directory. Each scheme was benchmarked and simulated as a system of 4 processors.

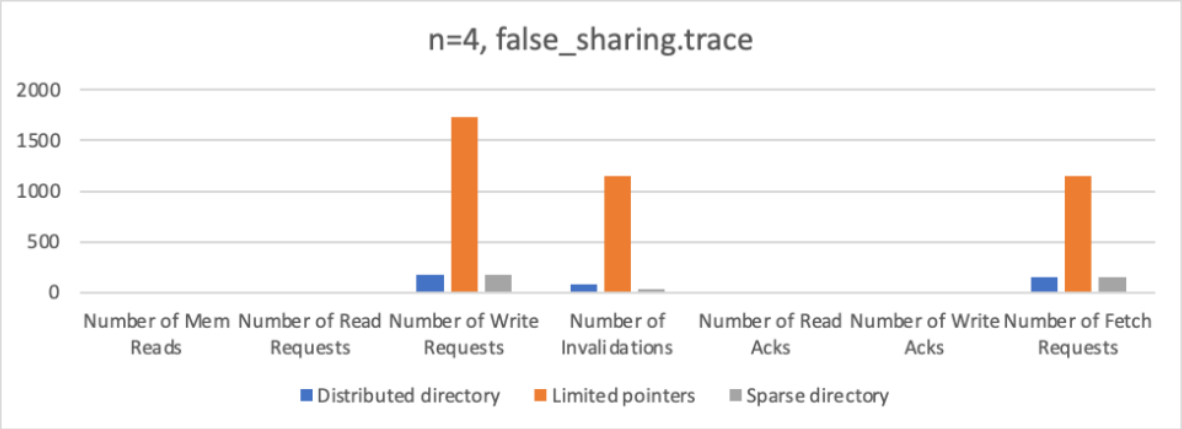
We observed effectively no differences in performance for the random.trace tracefile, likely due to the randomness of the access pattern not particularly stressing any bottlenecks that might appear in these directory based schemes.



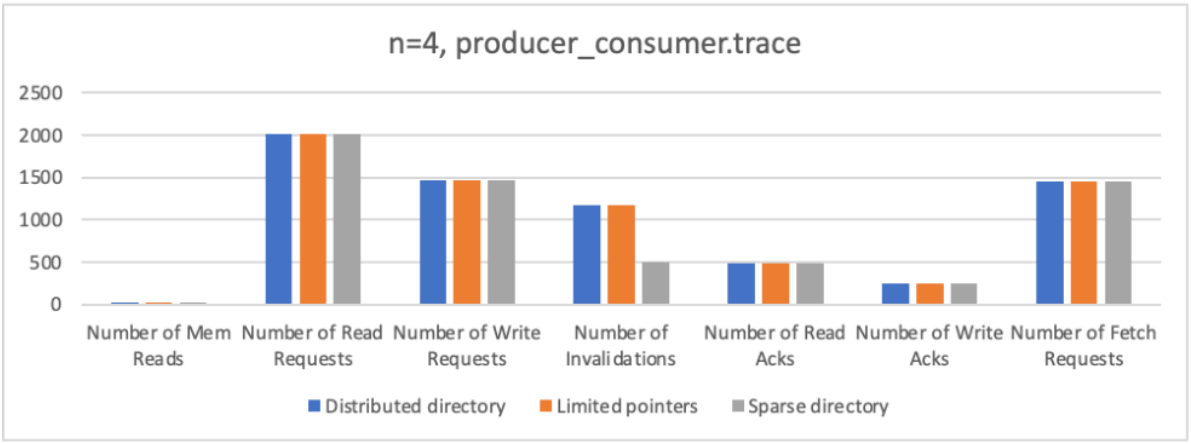
However, in case of shared_data.trace which essentially requires the system to be able to load and store the same cache block across different subsets of the available processors at various points, we can see that the limited pointer based scheme seems to require more memory reads and the sparse directory scheme requires fewer invalidations. This is likely because the limited pointer scheme directory only allows for a subset of the processors (8 processors) to hold a given cache line at the same time and if the number of processors exceeds that maximum, the block needs to be flushed to memory (since it is removed from the directory) before it is used again. In the case of the sparse directory scheme, the slight reduction in invalidations is likely due to the fact that the invalidations are never processed for any additional processors than those that are necessary and there isn't a hard limited on the number of processors that are allowed to hold a given block of memory at the same time.



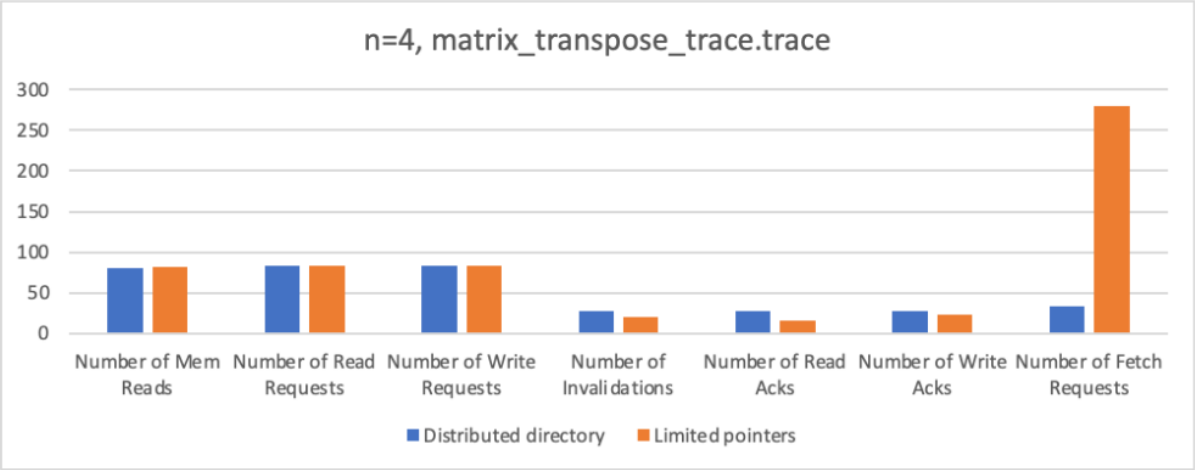
We see a similar effect in the false_sharing.trace case as well, where numerous processors access different addresses but they belong to the same block(s), leading to complex transactions that are necessary to maintain cache coherence.



The producer_consumer.trace simulates one or more producer(s) processors and one or more consumer(s) processors accessing a shared buffer. We do not observe a significant difference in performance across the different interconnect traffic metrics, but the sparse-directory scheme appears to have about half the number of invalidations as the standard distributed directory scheme and the limited pointer scheme, likely due to the the Sparse Directory likely having a reduced breadth of cache line sharing visibility. Thus, it only invalidates that single known sharer rather than broadcasting invalidations to all processors, leading to a lower count of invalidation messages.



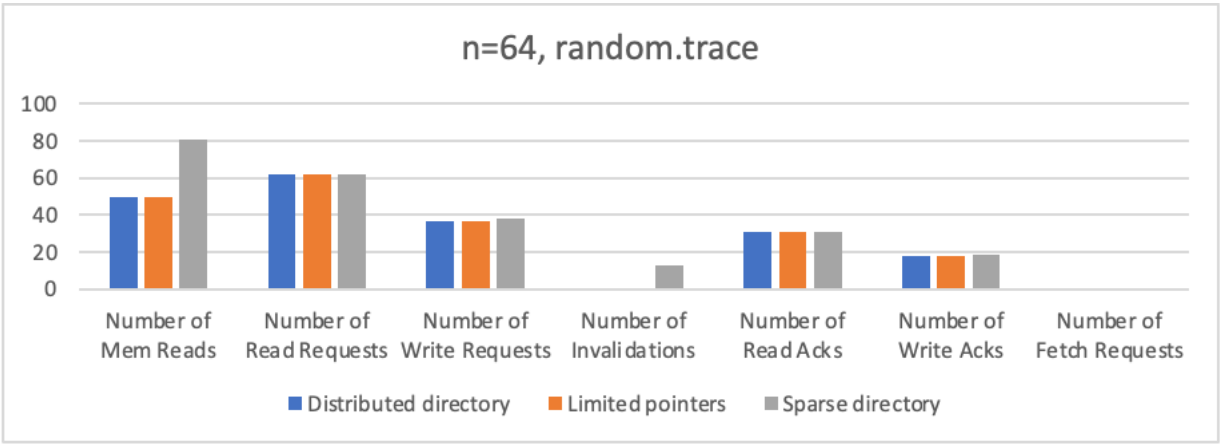
The matrix_transpose_trace.trace simulates the cache access pattern of a parallelized matrix transpose operation on a large matrix. There appears to be no difference between any of the interconnect metrics measured across all three schemes.



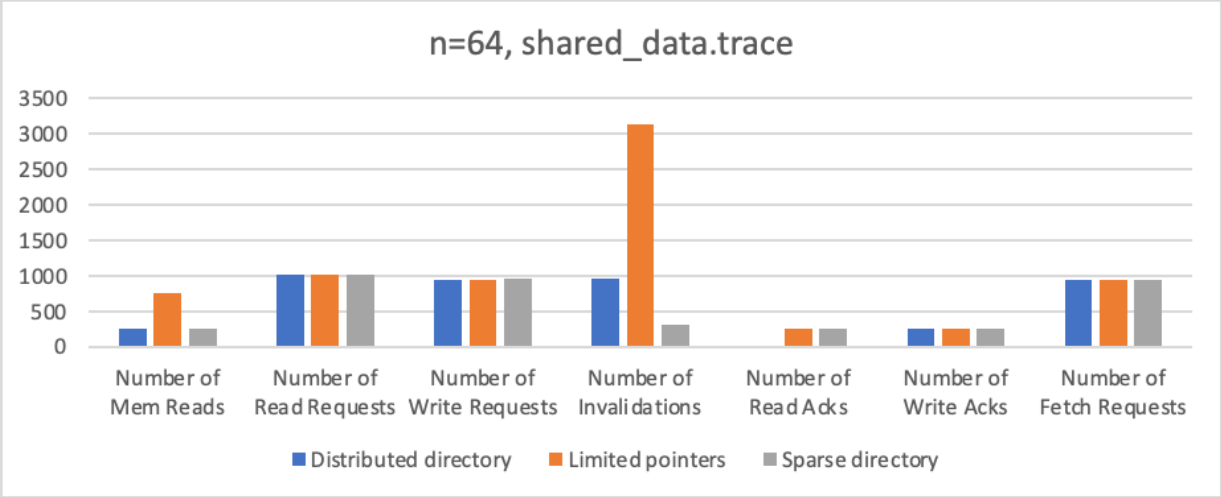
[2] Scheme vs Metrics for each trace file [Number of Processors = 64]

For the following graphs, we used a distributed directory based cache simulation, a limited pointer scheme based simulation, and a sparse directory scheme based simulation each with the same cache size per processor and the limited pointer scheme has a processor-membership sublist of size 8 within each entry of the directory. Each scheme was benchmarked and simulated as a system of 64 processors.

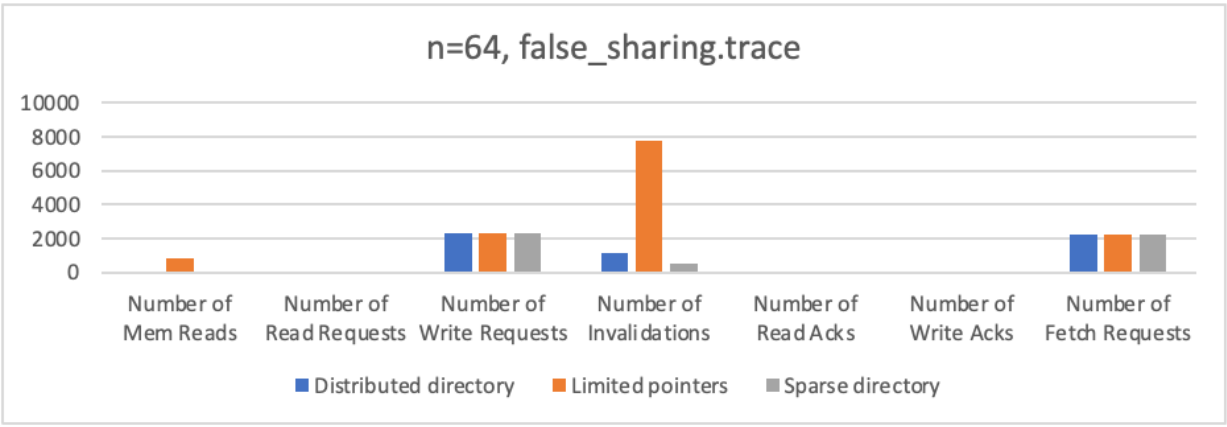
We observed effectively no differences in performance for the random.trace tracefile, likely due to the randomness of the access pattern not particularly stressing any bottlenecks that might appear in these directory based schemes.



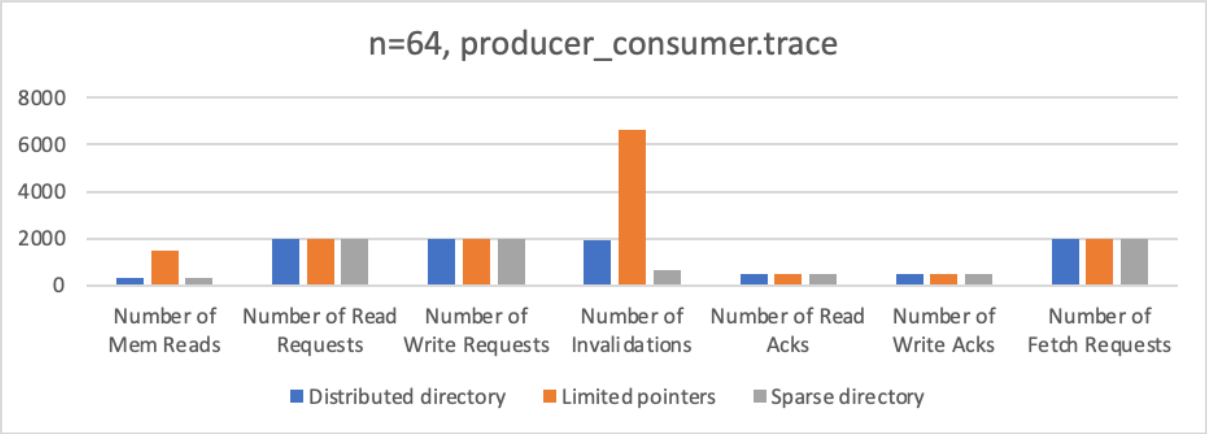
In the shared data case, the main observation was that the limited pointer scheme required multiple times the number of invalidations that the distributed directory based scheme and the sparse directory based scheme needed.



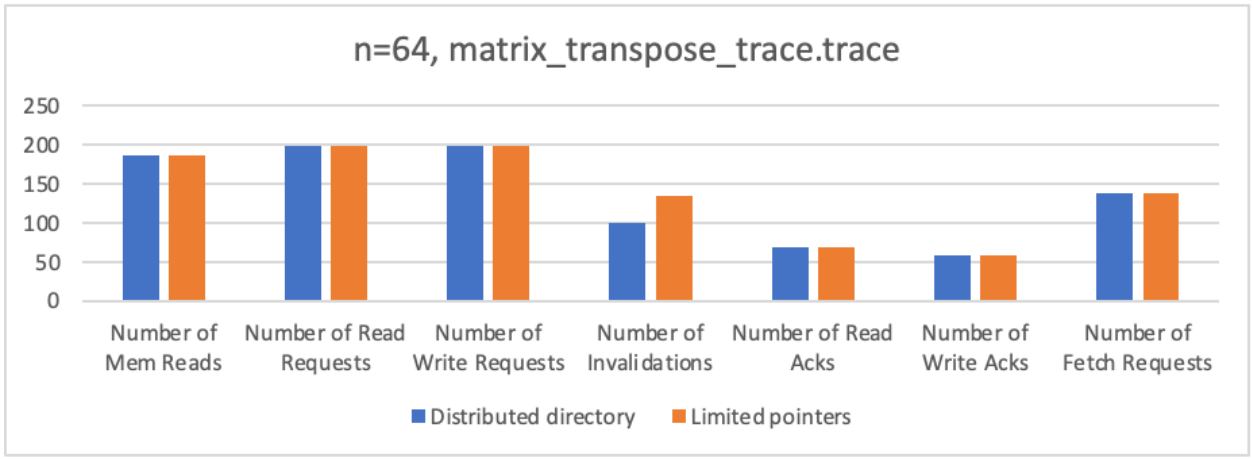
We see a similar pattern in the false sharing trace as well. This is likely because the shared data trace has multiple different sets of processors of different sizes sharing and modifying a given memory block and given that the limited pointer scheme has the directory store only a subset of the processors that have most recently used that block (here the subset is of the size 8 and the total number of processors is 64).



The Sparse Directory scheme likely results in fewer invalidations because it optimizes the way shared data is tracked. Since we use a linked list to track the caches that have a copy of a particular cache line, rather than keeping a full record of all sharers, when a write occurs, only the caches on the chain are invalidated. In this particular case, given the high contention cache or processor activity, the overall number of invalidations can be significantly fewer than the total number of caches, thus reducing the invalidation count compared to schemes that might invalidate more broadly.

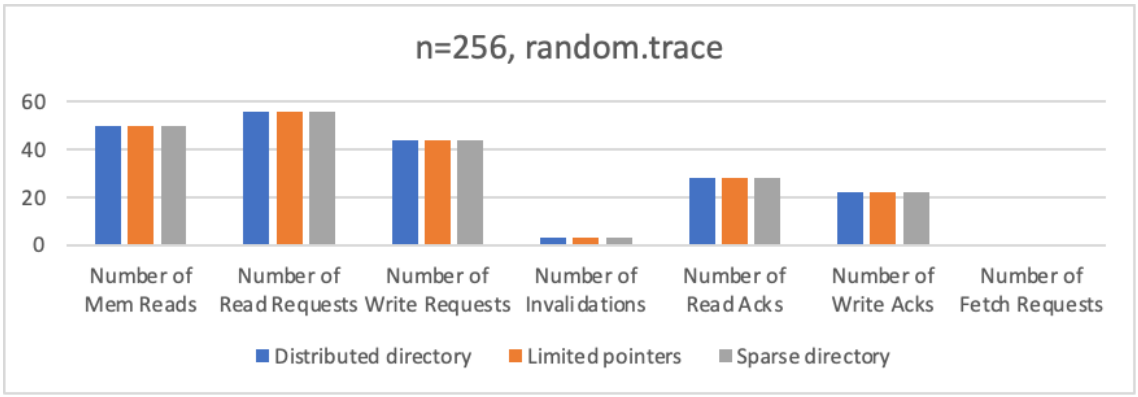


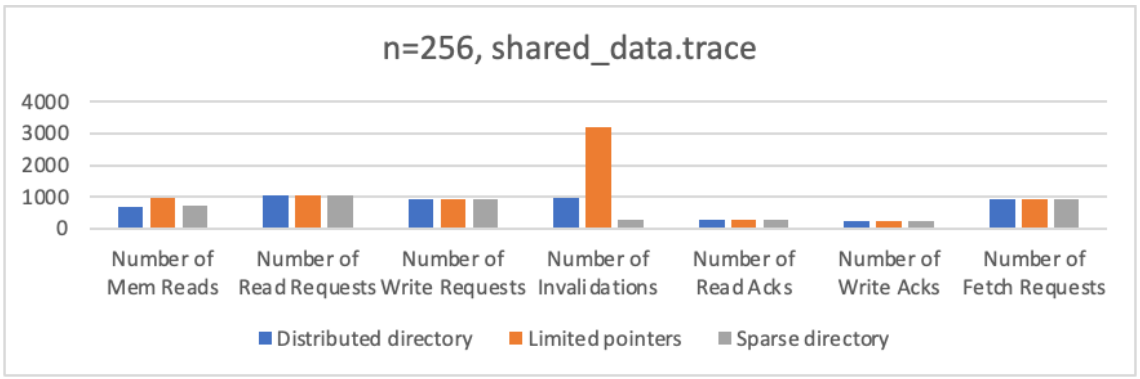
The matrix_transpose_trace.trace simulates the cache access pattern of a parallelized matrix transpose operation on a large matrix. There appears to be no difference between any of the interconnect metrics measured across all schemes. This is likely due to the randomness of the access pattern not particularly stressing any bottlenecks that might appear in these directory based schemes.

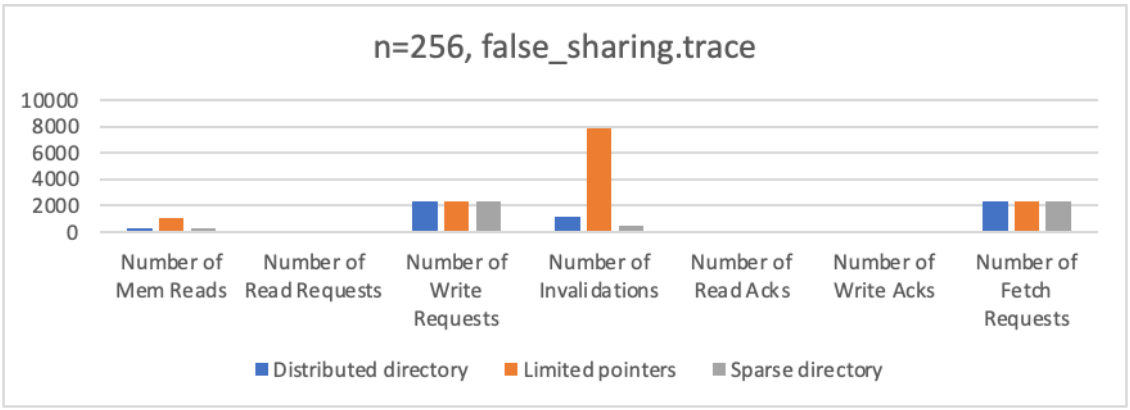


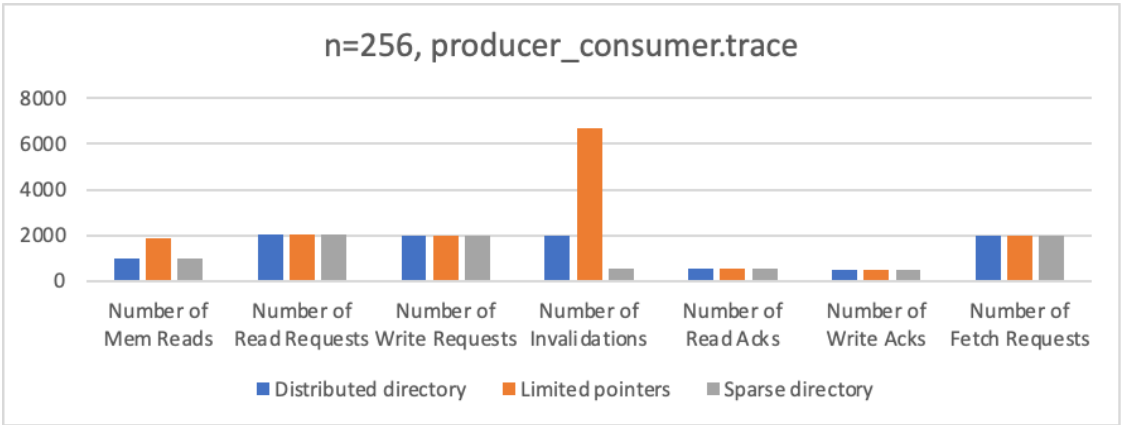
[3] Scheme vs Metrics for each trace file [Number of Processors = 256]

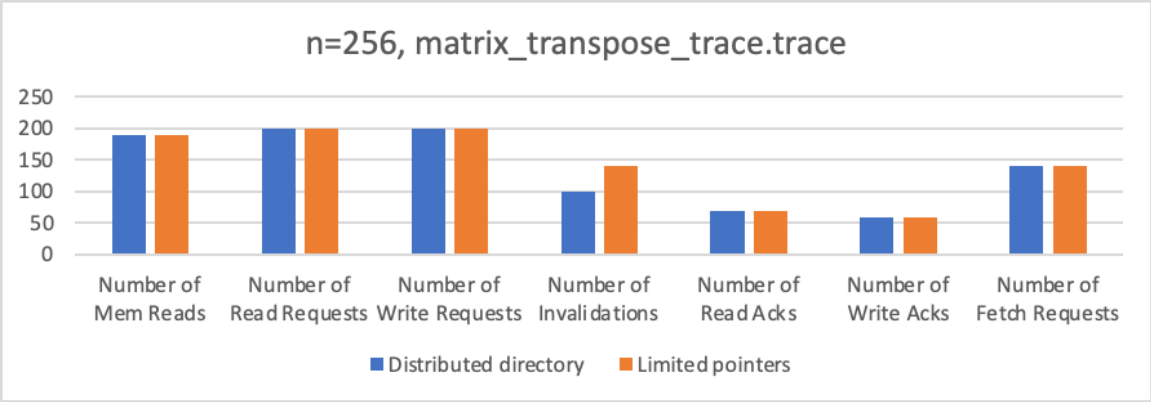
For random.trace, the metrics were the same across directory schemes since this is a random access pattern. In the shared data, producer-consumer, and false sharing traces, the limited pointer scheme had the highest number of invalidations (probably due to there being 8 node ID entries per directory entry, which resulted in more evictions from the cache resulting in more invalidations). The false_sharing.trace scale is much higher than any of the other metrics. It has the most amount of invalidations overall as the same cache line is accessed repeatedly over different cores.


n=256, random.trace
60
40
20
0
Number of Mem Reads
Number of Read Requests
Number of Write Requests
Number of Invalidations
Number of Read Acks
Number of Write Acks
Number of Fetch Requests
Distributed directory
Limited pointers
Sparse directory


n=256, shared_data.trace
4000
3000
2000
1000
0
Number of Mem Reads
Number of Read Requests
Number of Write Requests
Number of Invalidations
Number of Read Acks
Number of Write Acks
Number of Fetch Requests
Distributed directory
Limited pointers
Sparse directory


n=256, false_sharing.trace
10000
8000
6000
4000
2000
0
Number of Mem Reads
Number of Read Requests
Number of Write Requests
Number of Invalidations
Number of Read Acks
Number of Write Acks
Number of Fetch Requests
Distributed directory
Limited pointers
Sparse directory


n=256, producer_consumer.trace
8000
6000
4000
2000
0
Number of Mem Reads
Number of Read Requests
Number of Write Requests
Number of Invalidations
Number of Read Acks
Number of Write Acks
Number of Fetch Requests
Distributed directory
Limited pointers
Sparse directory



Limitations affecting Optimal Simulation / Performance

Some limitations on how when we might be able to simulate a multi-core cache coherent system or protocol includes:

1. Concurrency handling: Our current implementation does not explicitly handle concurrency issues that might come up in distributed cache systems where simultaneous accesses and modifications by multiple processors can lead to race conditions, potentially causing inconsistencies in cache states or directory entries.
2. Design of the interconnect might not be optimal for simulating the data latency constraints that might be expected in real world scenarios for systems with a very high number of processors. This might require the interconnect to have a different network topology or communication model.

DISCUSSION

Expectations and Speculations

Distributed directory scheme: By maintaining a record of which processors have cached or modified a particular memory line, directories eliminate the need for broadcast coherence traffic, leading to more efficient data management. This model is therefore likely particularly beneficial in NUMA systems, where directories are distributed and co-located with the corresponding memory segment they manage, enhancing performance by reducing unnecessary data movement.

Limited pointer scheme: The implementation is expected to reduce the complexity and size of the directory structure at the potential cost of increased eviction handling with respect to the initial distributed directory implementation. This scheme is likely beneficial in scenarios where directory size is a constraint and where it's unlikely that all processors will share the same cache line frequently. While this approach significantly reduces storage overhead, it may compromise efficiency in high cache sharing scenarios (many caches keep requesting the same set of lines from one particular node/processor/cache).

Sparse directory scheme: This sparse representation is more memory-efficient when few processors share the same cache line. This sparse directory approach likely reduces the directory's memory requirement overall by only storing entries for cache lines that are actually shared among processors, rather than reserving space for every possible processor, regardless of whether they share the line. It is particularly effective in systems with many processors where each processor only shares a small subset of cache lines.

Suggestions for future work or improvements

1. Multi-thread the sequential cache simulator implementations.
2. Optimizing directory based coherence using intervention and request forwarding.

REFERENCES

List of all references and sources used in the project:

1. CMU Lecture 11: A Basic Snooping-Based Multiprocessor Implementation https://www.cs.cmu.edu/~418/lectures/11_snoopimpl.pdf
2. CMU 15-418: Lecture 12: Directory-Based Cache Coherence https://www.cs.cmu.edu/~418/lectures/12_directorycoherence.pdf
3. CMU 15-213 / 18-213: Cache Lab
4. Raw data: https://docs.google.com/spreadsheets/d/143WNK1INrW9rxRaMlmOB94sBOwMVhPAs7hkYOLtuXkc/edit#gid=959359735

WORK DISTRIBUTION

Tasks

- Implementing the distributed directory based approach.
- Debug and test limited distributed directory implementation.
- Implementing the limited-pointer based scheme.
- Debug and test limited-pointer implementation.
- Implementing the sparse directory based scheme.
- Debug and test sparse directory implementation.
- Generating trace files with different access patterns.
- Benchmarking performance with different cache & system parameters.
- Generate tables, graphs etc for the final report.
- Write the final report.
- Draft poster slides.

The tasks were evenly distributed between both team members.