Optimizing for Producer-Consumer Relationships in Multi-Core Coherent Systems

Matthew Barondeau, Chester Cai, Divija Gogineni, Ying-Wei Wu

***Abstract*—Distributed shared memory system offers the programmer a flat view of memory and provides ease of programmability over message-passing machines. The communication overhead is handled in hardware with coherence mechanisms. Standard coherence mechanisms lazily move data across cores (i.e. the data movement happens when the requester needs the data), which leads to significant performance overhead in a producer-consumer relationship. In the paper, we present a mechanism that uses global information across all cores to orchestrate data movement. We show that our mechanism can effectively reduce the overhead of coherence protocols for workloads with a producer-consumer relationship and speedup execution by 2.39X compared to a 1.26x speedup using only local prediction.**

I. Introduction

Distributed shared memory systems provide a global address space while the actual memory is physically distributed among all processor nodes. As in a single processor, data brought in from remote memory are also cached to improve locality and reduce the long network latency to fetch data. When multiple processor nodes operate on some common memory region, they will all maintain a local copy of the same piece of data. The duplication and sharing of data necessitates a coherence protocol to ensure program correctness. Traditional coherence protocol allows different processor nodes to share read-only copies of cache lines and forces processor nodes to obtain an exclusive copy when writing, which complicates both the reading and writing process than simply bringing in the data. In the case of reading, a cache miss cannot simply be serviced by the next level in the cache hierarchy, but instead needs to be serviced by the directory. In the case of a write, a cache miss, or a hit on a cache line in a shared state, the directory must invalidate all other copies in the system. This invalidation overhead can be a significant portion of total execution time if the data moves across cores frequently.

Prior works have focused on using local information available to each core to predict dead blocks or blocks that might be accessed in the future. The prediction is then used either as to early flush or prefetch the particular line. In this paper, we will focus on using global information across all cores to orchestrate data movement to achieve optimal performance. The prediction made with the global information is then used to move data across cores to help accelerate the producer-consumer relationship.

We use the Structural Simulation Toolkit[8] to simulate our modification to the coherence protocol. The result shows that coupling a local predictor with global information from the directory can efficiently decrease the coherence overhead in a producer-consumer relationship workload.

II. Related Work

Coherence traffic can account for significant overhead in modern many-core systems. There have been several optimizations in reducing traffic by adding additional states to the coherence protocol [7], but these approaches require modifying the entire protocol. Rather than modifying the coherence states, recent proposals have favored predictive actions to mitigate worst-case coherence latency.

One of the first major works was Dynamic self-invalidation (DSI) [1], which was first proposed to reduce the overhead of a writeback from a modified state. The writeback, shown on the left of Figure 1, involves 4 serial coherence messages, which greatly increases load delay. Another source of delay is invalidations of all shared copies of a line when a cache wants to modify a line, which is shown on the right of Figure 1. The primary goal of DSI is to reduce both of these delays by self-invalidating a line whenever the last write or read to a line has occurred as shown in Figure 2.

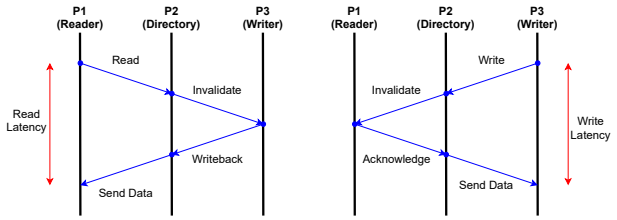

Fig. 1. Overhead of coherence protocol

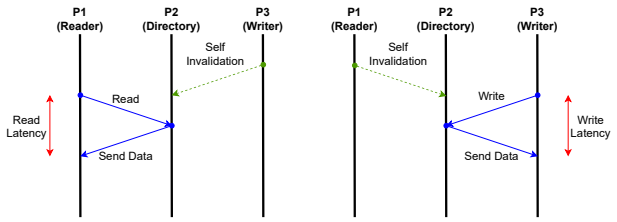

Fig. 2. Coherence overhead with self-invalidation

The original DSI protocol paper utilizes extra bits to keep track of the version of a cache block and introduce additional states to the coherence protocol. With these two bits of metadata serving as history information, the directory can form a prediction of the likelihood that the block will be invalidated in the future and determine whether it should self-invalidate. As for the time to self-evict the remote cache line,

each processor node maintains a fixed-size FIFO or linked list. Whenever an entry in the FIFO or linked-list is replaced, the self-invalidation of the victim is performed. Followup work examined two aspects of the DSI mechanism:

1) Identifying which block to self-invalidate at directory
2) Identifying when to self-invalidate on the processor side

One such work was the Last-Touch Predictor (LTP) [2] that focused on predicting dead blocks within the cache and invalidating these lines. This approach used trace-based correlation and associated a last touch with the sequence of instructions touching the block until the block is invalidated. LTP then uses this trace to match against future accesses and predict when blocks will not be used again before invalidation. Since this approach uses trace-based correlation, accuracy results are higher than DSI for determining when a block should be invalidated. However, this approach neglects opportunities for evictions that do not follow the exact instruction trace the predictor is expecting. Furthermore, since tracking entire traces is expensive, the predictor maintains a small encoding of a trace called that can result in aliasing and reduces accuracy.

An optimization for LTP was the Sampling Dead Block Predictor [5] that predicting dead blocks based on sampling rather than the exact history trace. One of their key observation is that memory access patterns are consistent across sets, and thus you need much less information to perform accurate predictions. It further reduces metadata costs by using the address of the last memory access instruction instead of keeping track of access patterns for all blocks. Overall, both LTP and Dead Block Prediction are an improvement over DSI, but both rely entirely on the actions of the local processor and thus misses other activity in the system that may affect line state.

One approach that utilized knowledge of other nodes [3] tried to optimize the case of an upgrade request given many other caches had the line in a shared state. This approach stored the predicted caches that would hold the line and tried to bypass the directory access to send requests. While this approach does reduce coherence traffic and utilizes knowledge of other node state, it only predicts sharing nodes and still requires the cache to send out the invalidate requests upon access.

Recent work has begun to focus on using multiple sources of information to reduce coherence overhead. [4] uses both a Downgrade Predictor and Upgrade Predictor to hide both the producer and consumer latency on request. Their Downgrade predictor uses a trace based approach whereas their upgrade predictor for the subsequent readers of the line use a simple pattern matching predictor. Since their upgrade predictor is so simplistic and because they require caches accesses the directory for each request, their potential gains are limited.

Other work, [9] [10] utilizes a dedicated hardware structure to enable fast communication between processors. Using a specialized instruction, data is sent to a VirtualLink table where the consumer can remove the information rather than having to rely on coherence overhead. The extension to this work, SPAMeR, speculatively sends information from the central table to the consumers, but this speculation is unidirectional. In our efforts, we focus on working within the existing coherence network and allowing speculative upgrades to the producer and consumer.

Overall, all prior work focuses heavily on predictors that utilize only local information, or a simplistic approach to invalidate potentially many sharers for an Upgrade predictor. In our work, we examine how global knowledge and programmer configurations can improve coherence overhead over that of even an ideal local predictor.

III. Motivation and Implementation

When examining prior work, we find that the work focuses almost exclusively upon a single aspect of the system when making predictions. In particular, each system except for VirtualLink predicts, without programmer or compiler input, when it is finished with line and self-invalidates. Since the programmer is aware of when a last write will occur in a conventional producer-consumer relationship with a shared buffer, we examine improving the prediction accuracy using workload information. Secondly, since the programmer is often aware of both the producer and consumer operation in these systems, we can inform the directory of the core that will next request the line, and the requesting state, and further reduce coherence latency. We model our system as a single core producer and a separate single consumer with a circular buffer in shared memory as the primary means of communication between the two.

A. Cache Side Predictors

We implement an ideal last-write predictor for a producer such that, whenever the producer writes to the last word in a cache line, the line will not be used again and can be flushed. As a result, our ideal predictor will always know when the last touch is performed and can both invalidate the line and send a specially tagged message to the directory to move the line to shared along with the dirty data.

Similar to the last-write predictor for a modified line, our last-read predictor operates on the end of a cache line read. Since we have a streaming access pattern, once the consumer has finished reading the contents of a cache-line, the producer must write new data to the line before the next valid read. Therefore, we invalidate each line after reading the last line in the block and send a dedicated message to the directory. This dedicated message removes the consumer as a sharer and, given that it has a special tag, can inform the directory that it can take additional action. This is an idealized version of the DSI and LTP as the consumer know exactly when the last touch for each block will occur, and is done to determine the maximum advantage of such a scheme.

While these approaches are simple and highly accurate given our workload, there exist other loads and stores not to the circular buffer as part of our workloads. Since these loads and stores do not exhibit the same pattern of behavior as the well-defined circular buffer, the predictor must be disabled for these instructions. As the circular buffer exists in a defined region of memory, we configure each cache with a register defining region of memory that the predictor should pay attention to, while ignoring all other regions to solve this issue.

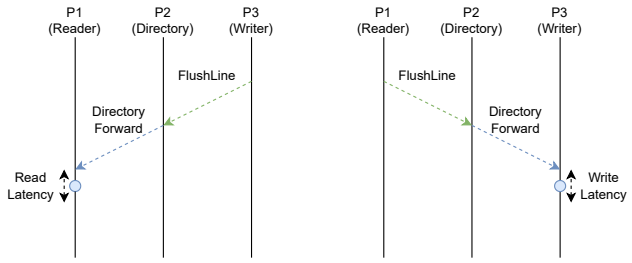

Fig. 3. Updated Scheme Timing Diagram

B. Directory Forwarding Logic

In the prior work, all schemes relied on predicting solely using local information. Since we have global knowledge of the workload in this system, namely which processor is writing to a circular buffer and which processor is reading, we can forward information after a self-invalidation. We add a user programmable producer and consumer register so the directory knows which cache should receive the data. We add four new messages to the coherence system to support our forwarding and self-invalidation.

The first two messages, *InvSpecial* and *FlushLineSpecial* are commands from the caches to the directory that are used for the self-invalidation. *InvSpecial* is used by the consumer to indicate it is done reading a cache line and that the directory can remove the consumer as a sharer as well as upgrade the producer. *FlushLineSpecial* is used by the producer after writing a line, as it flushes the data contents to the directory and moves to a shared state. SST implements the baseline *Inv* and *FlushLine*, but we add these two additional messages to indicate that the producer and consumer have sent the messages and additional action should be taken.

The other messages added to the system are used by the directory to upgrade the cache lines without receiving a request first. These two messages, *SRespSpecial* and *XRespSpecial* are used to upgrade lines to shared or exclusive states respectively and send data payloads. These messages are not triggered by requests from the caches, so the next access need not go to the directory. As a result, we can remove the latency of the request to the directory as well as the response, and the proposed timing diagram is shown in Figure 3.

While this removes some delay, as a result of not having the cache make the request, there is the possibility of the line not currently being allocated in the cache. There are two solutions to this issue:

1) Evict a cache entry and insert the directory payload
2) Reject the message and have the cache message the directory when it wants to read or write to the line

The first solution runs into the issue of buffer overflow. That is, given a circular buffer that is larger than the L1 cache size, the directory can potentially overwrite all elements of the cache by and even evict other buffer entries by sending too many messages. The solution to this is to restrict the number of buffer entries that the directory is able to allocate at any given time using a programmable counter. Whenever the directory succeeds in a push to the cache, the counter is decreased. Conversely, whenever the directory receive a *Flush* message, the counter is incremented.

The second approach to solving the speculative upgrade issue is much simpler, but it decreases the benefit of the directory messages and results in many unused coherence messages. There may be ways to mitigate this overhead, but they are not explored here. Overall, the intention is to have the directory forward useful information from the producer to the consumer and vice versa to outperform even an ideal local predictor. This is done with a small overhead of 4 new message types, simple predictors in the producer and consumer caches with base and bound registers, and two counters at each directory to prevent cache overflow. All the hardware changes are shown in Figure 4

IV. Methodology

A. SST Overview

The Structural Simulation Toolkit (SST) [8] is an ongoing project developed by the Sandia National Laboratory. Unlike Gem5 or Scarab which focus on single core simulation, this project aims to offer a flexible architectural simulation framework suitable for high performance computing (HPC) research. The fully modular design allows for extensive exploration of an individual system component without the need for intrusive changes to the whole simulator environment. In our case, we would like to modify the directory implementation while utilizing the provided element library to instantiate other essential components, making SST a perfect fit for our purpose.

Within the parallel simulation environment, the system is modeled as shown in Figure 5. There are two processors each with a private L1 cache and two piece of distributed memory, all of them connected to the network separately. Each piece of memory has a directory associated with it that controls a portion of the address space. The parameters of the system configuration are shown in Table I.

We initially launched our workloads on the RISC-V ISA simulator core Sandia provided, and later moved on to a PIN based CPU model that dynamically capture and simulate a memory trace from a running binary, but eventually settled with a simplistic pattern generation core. This was mainly due to issues mapping openMP workloads that shared the same address space onto different processor cores for simulation. With the pattern generator, we could work entirely with physical addresses and not run into this issue. After all, the memory access pattern is all that matters in this work.

B. Synthetic Workloads

We use synthetic memory access patterns to simulate the effect of producer and consumer relationships in a multicore environment. In each benchmark, there is a single producer and a single consumer. The producer continues to write to a shared data structure which the consumer reads from. We simulate a perfect locking scheme where the consumer always has perfect information about what has been written by the producer without any overhead. The consumer will not issue reads until there are items produced by the producer. This is

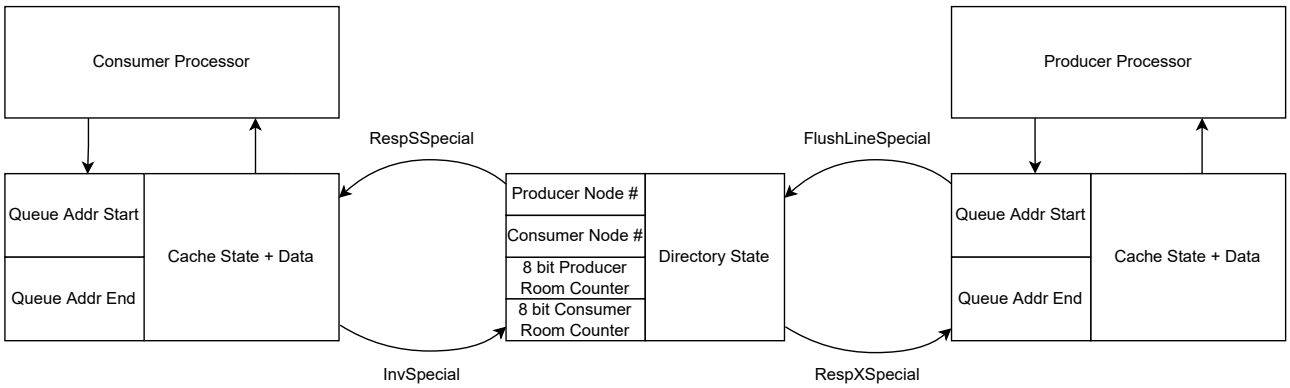

Fig. 4. Directory Overflow Example

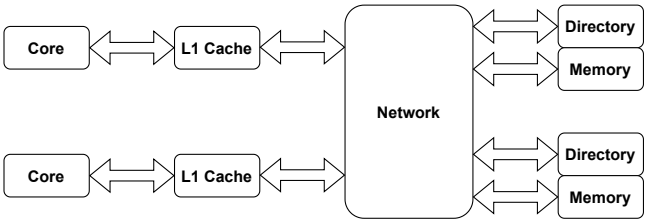

Fig. 5. System Model

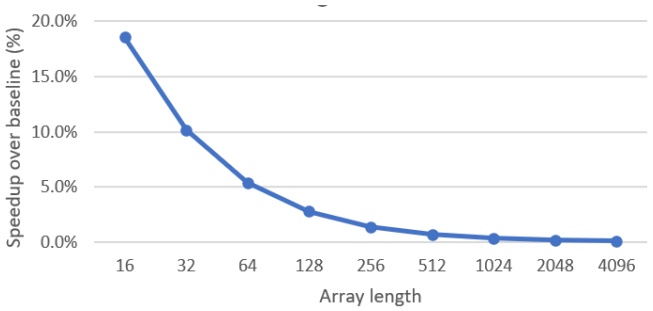

Fig. 6. Array Size Sweep

TABLE I
SPECIFICATION OF THE SYSTEM USED

Core	2 Miranda pattern generator cores 2.4 GHz
L1 Cache	2 KB per core 2-way set associative 64 B cache line size 3-cycle fixed latency
Memory	1 GB (500 MB per core) 50 ns fixed latency 500 MHz controller
Network	Crossbar 60 GB/s bandwidth

a realistic assumption as this assumes the producer runs faster than the consumer, causing no locking overhead.

The first benchmark uses an array as the shared data structure between the producer and the consumer. The array is written only once and the benchmark finishes at the end.

The second benchmark uses an array as the shared data structure between the producer and the consumer. When we reach the end of the array, we then start from the start of the array again. This process is repeated 1000 times before the simulation finishes.

We decided to implement these synthetic benchmarks rather than finding some queue benchmarks due to issues mapping the application to SST. We later expanded our baseline benchmark to include other memory access patterns, as each producer would need to load data before putting it into the queue and each consumer should have to perform some processing on each queue element.

V. EXPERIMENTAL RESULTS

We compare all our optimizations to a baseline without any predictive mechanisms using SST. We first compare our dual predictor system in against the baseline. Finally, we compare the directory forwarding capability to our comparison.

Originally, we found little to no advantage to our scheme over the baseline, as the size of the queue was quite small. As a result, the producer core can quickly perform all writes and the majority of the time it will be waiting for the reads to complete. We sweep the number of cache lines used from 16 to 4096, given a 64kB data cache, and found that our benefit decreased exponentially given larger arrays as shown in Figure 6. We found this benefit to be due to the relatively fast network latency of 10ns. In this experiment, consumer starts reading as soon as 16 cachelines are written by the producer. Hence 16-32 cacheline runs the maximum benefit and for the larger queue sizes, only small portions of their execution time spent waiting on messages, there was little advantage to accelerating.

To determine the impact when we scale up the network delay, we sweep the network delay from 0.1ns to approximately 32 microseconds. The results of the sweep are shown in Figure 7. For any network delay less than 50ns, the network is sufficiently fast as to hide most of the coherence delay. However, as we increase the delay, the self-invalidation begins to have a benefit up to a maximum of 50% over the baseline. Figure 7 shows that the maximum benefit we can achieve is less than the benefit of both flushing and directory forwarding.

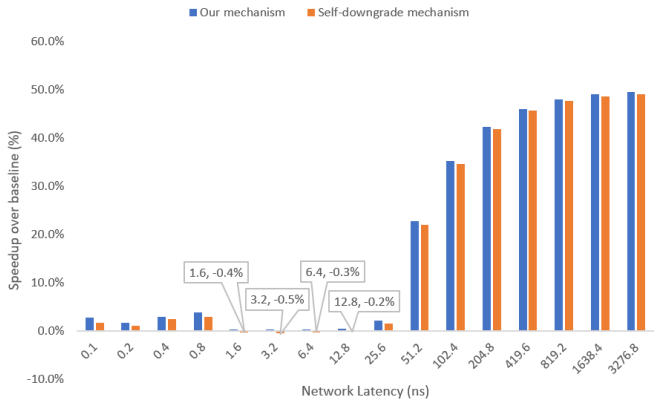

Fig. 7. Network Latency Sweep

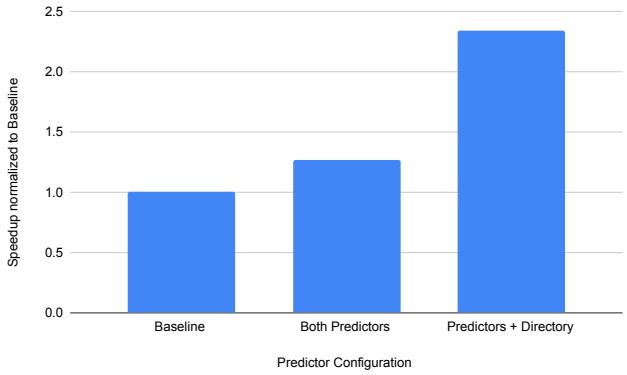

Fig. 8. Execution Time comparison for long Write-Read Delay

We then removed the directory forwarding to determine the benefit of just the idealized local predictors.

Running the comparison of the three techniques, we found that the predictor only scheme is actually slower than the baseline for some network delays. The benefit of a predictor flush is to hide the coherence latency, but once a line is flushed from the cache, it is written to memory. Therefore, if the network latency is less than half of the memory latency, it is faster to keep the data in the other cache and not perform the self-invalidation. This case appears between the 1.6 and 12.8 ns network latencies in Figure 7. In these instances, the network is sufficiently faster than memory latency, but not fast enough to present the data from being written out to the DRAM. When network latency is much smaller, the request from the consumer can arrive before the directory can trigger the writeback. However, we do see some benefit with our forwarding mechanism here as we send the cache line to the consumer in addition to writing it to memory, so we are not paying the memory latency on all requests.

In addition to preventing memory writebacks due to self-invalidations, the directory forwarding mechanism should enable higher performance than the standalone predictors. This use case should occur when the delay between data being produced and consumed is four times the network delay. Without this delay, the consumer will still send the request to the directory for a read as it did not have the data in the cache when the processor requested it. We create a program in which the delay between producer and consumer satisfies this requirement and set the network latency to 100 ns. The result of this run is shown in Figure 8 and we expect that as network latency increases, our advantage over both the baseline and local predictors will increase.

Overall, we show that coupling local predictors with a knowledgeable global directory can decrease the coherence overhead for a producer-consumer relationship. In addition to reducing the number of coherence transactions between the cache and the directory, the forwarding mechanism prevents reduces memory accesses due to self-invalidations. Whenever there is a sufficiently slow network, this secondary benefit is reduced, but with a fast interconnect, self-invalidations can reduce overall performance without the directory forwarding.

VI. Conclusion

In this project we have demonstrated the benefit of adding additional forwarding logic at the directory for producer and consumer relationships in a multi-core coherent system. While most of the benefit can be achieved using highly accurate local information, the addition of logic to the directory to forward the information to the next users reduces the number of coherence transactions. In a system with large network delay, the reduction in transactions can lead to a significant performance increase. Beyond the reduction in traffic, the forwarding by the directory reduces the number of cache misses due to self-invalidation. The directory logic can be later extended for additional workloads, with programmer knowledge, to accelerate communication between processors.

References

[1] Alvin R. Lebeck and David A. Wood, *Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors*, Proceedings of the 22nd International Symposium on Computer Architecture (ISCA), 1995, pp. 48-59 vol.2.

[2] An-Chow Lai and Babak Falsafi, *Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction*, Proceedings of the 27th International Symposium on Computer Architecture (ISCA), 2000, pp 139-148

[3] Manuel E. Acacio, Jose Gonzalez, Jose M. Garcia and Jose Duato, *The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors*, Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT), 2002

[4] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki and Babak Falsafi, *Memory Coherence Activity Prediction in Commercial Workloads*, Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI), 2004, pp. 37-45

[5] Samira M. Khan, Yingying Tian and Daniel A. Jiménez *Sampling Dead Block Prediction for Last-Level Caches*, Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010, pp. 175-186

[6] Samira M. Ajorpaz, Elba Garza, Sangam Jindal and Daniel A. Jiménez *Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer*, 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018, pp. 518-532

[7] Papamarcos, Mark S. and Patel, Janak H. *A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories*, Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA), 1984, pp. 348-354

[8] A. F. Rodrigues, K. S. Hemmert, B. W. Barret, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, *The structural simulation toolkit*, SIGMETRICS Perform. Eval. Rev., 38(4):37–42, 2011.

[9] Q. Wu, J. Beard, A. Ekanayake, A. Gerstlauer and L. K. John, *VirtualLink: A Scalable Multi-Producer Multi-Consumer Message Queue Architecture for Cross-Core Communication*, 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2021, pp. 182-191

[10] Q. Wu, A. Ekanayake, R. Li, J. Beard, and L. K. John, *SPAMeR: Speculative Push for Anticipated Message Requests in Multi-Core Systems*, The IEEE International Conference on Parallel Processing (ICPP), 2022