MOESI-PCD: A Delegation-Based Extension of MOESI for Efficient Producer-Consumer Sharing

Yiting Lai
University of Michigan
Ann Arbor, United States
lindalai@umich.edu

Xiaomi Zhou
University of Michigan
Ann Arbor, United States
zhouxm@umich.edu

Yunqi Zhang
University of Michigan
Ann Arbor, United States
yunqizh@umich.edu

Haowen Tan
University of Michigan
Ann Arbor, United States
thwthw@umich.edu

Hao Han
University of Michigan
Ann Arbor, United States
hhjake@umich.edu

***Abstract*—Cache coherence protocols are critical to maintain data consistency in shared memory multiprocessor systems. This paper presents MOESI-PCD, a novel extension of the MOESI protocol that optimizes coherence operations for producer-consumer relationships. Traditional protocols often generate excessive coherence traffic for these common access patterns, by introducing a delegation mechanism that allows producers to directly manage coherence for their consumers, our protocol reduces network hops by up to 33% and eliminates redundant state transitions. We implement a comprehensive model using the Murphi verification system and prove correctness through state exploration and invariant checking. The results demonstrate significant performance benefits for workloads with stable producer-consumer patterns while maintaining all coherence guarantees through a robust rollback mechanism.**

I. Introduction

As multicore architectures continue to scale, the efficiency of cache coherence protocols becomes increasingly critical for system performance. Directory-based coherence protocols help manage the growing complexity of maintaining data consistency, but often generate excessive traffic for common access patterns. One such pattern is the producer-consumer scenario, where one processor repeatedly updates data while others only read it - for example, a video encoder that produces frames while multiple display processors consume them.

In traditional coherence protocols, producer-consumer patterns cause processors to cycle through unnecessary state transitions. Consider a specific scenario: Processor A updates a shared variable and enters Modified (M) state. When Processors B and C read this data, Processor A transitions to Owned (O) state. When Processor A needs to update again, it must acquire exclusive access, invalidating B and C, and transitioning back to M state. Each transition generates 3-4 network messages per consumer, consuming bandwidth and increasing latency - in a 16-core system, a single update could generate over 30 coherence messages.

Prior work has explored several adaptive cache coherence protocols. Both Stenström et al.'s and Cox et al.'s researches on migratory sharing optimization [1], [2] demonstrated that protocol adaptivity can significantly reduce coherence traffic for specific access patterns. Similarly, Kayi et al. [3] showed that bandwidth-adaptive protocols can dynamically adjust to network conditions, further improving performance under varying loads.

This paper introduces MOESI-PCD, which extends the classic MOESI protocol [4] with an intelligent delegation mechanism [5]. Our protocol detects stable producer-consumer patterns at runtime and optimizes coherence traffic by delegating coherence management from the home directory to the producer node, enabling direct communication between producers and consumers, reducing network hops for common operations from 3–4 to just 2, providing a robust rollback mechanism to handle race conditions and ensure correctness, and implementing a self-downgrade mechanism to manage transitions from the Modified (M) to Shared (S) state without directory intervention.

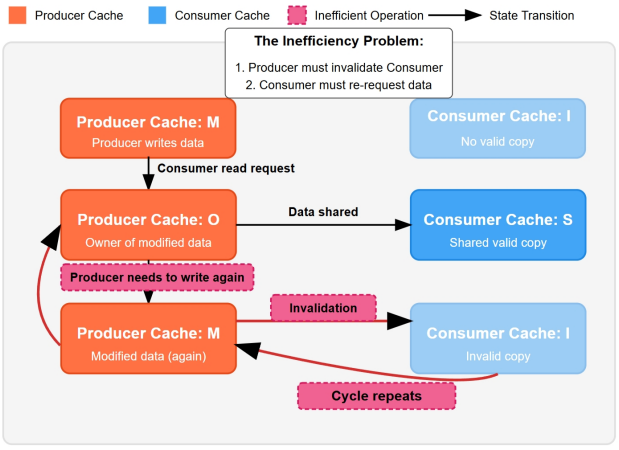


Fig. 1: MOESI Protocol Inefficiency in Producer-Consumer Pattern

II. Background and Motivation

A. Protocol Foundation: Directory-Based MOESI

We built MOESI-PCD on a directory-based MOESI foundation, combining two architectures with complementary strengths. Directory-based protocols offer crucial scalability advantages for many-core systems by targeting messages only to relevant processors, avoiding the $O(n^2)$ communication complexity of snooping protocols that becomes prohibitive beyond 8-16 cores. The MOESI protocol extends the basic MSI model with two additional states: Owned (O), which allows sharing modified data while maintaining write privileges, and Exclusive (E), which enables silent transitions to Modified.

Our design deliberately leverages these foundations: the directory's sharer list becomes the template for our consumer list, while MOESI's O state serves as the natural detection mechanism for producer-consumer patterns. When a processor in O state attempts another write, it signals a likely producer-consumer relationship, creating an ideal trigger for delegation. Our mechanism addresses the primary weakness of directory protocols—indirection overhead—by enabling direct producer-consumer communication after initial setup, while maintaining backward compatibility with standard MOESI operations for non-delegated addresses. This hybrid approach combines the scalability benefits of directories with the latency advantages of direct communication, all while requiring minimal protocol extensions.

B. Producer-Consumer Pattern Inefficiencies

Producer-consumer patterns are prevalent in parallel applications across diverse domains. In video processing pipelines, an encoder processor repeatedly updates frame data while multiple decoders consume it. In database systems, a transaction processor generates results that multiple query processors read. Web servers commonly employ producer-consumer queues where request handlers produce responses that are consumed by network transmission threads.

Figure 1 illustrates the inefficiencies in traditional MOESI for a specific producer-consumer scenario, with quantified message and hop costs:

1) **Producer writes data** (enters M state): 2 messages, 2 hops.
2) **Consumer reads data** (producer downgrades to O): 3 messages, 3 hops.
3) **Producer writes again** (invalidates consumers, returns to M): 3 messages per consumer, 3 hops.
4) **Consumer reads again** (cycle repeats): 3 messages, 3 hops.

In a concrete example with one producer and three consumers on a 16-core system with a 30-cycle network hop latency, each producer update generates 12 total messages and incurs 180 cycles of coherence latency (6 hops × 30 cycles). For a streaming application performing 1,000 updates per second, this translates to 12,000 coherence messages and 180,000 cycles of coherence latency per second.

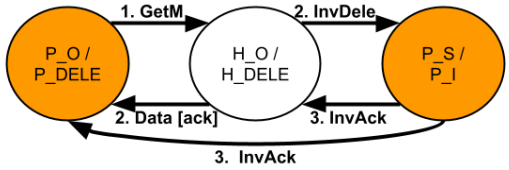
III. MOESI-PCD Design

A. Protocol Overview

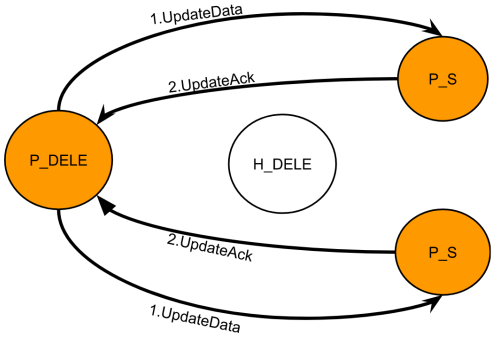
MOESI-PCD optimizes producer-consumer patterns through a delegation mechanism triggered when a processor in Owned state attempts another write. The implementation proceeds as follows:

1) Processor in O state sends GetM to directory, indicating a producer pattern.
2) Directory sends InvDele to sharers and records the requesting processor as delegate.
3) Processor transitions to P_DELE state and initializes a consumer list.

This mechanism enables direct producer-consumer communication, reduces network hops by 33%, allows efficient updates without invalidations, and provides fallback paths when patterns change.



(a) From Owned state to DELE.



(b) Update handling

Fig. 2: MOESI-PCD write requests handling.

B. Delegation Mechanism

The key insight of MOESI-PCD is recognizing when a processor exhibits producer behavior. Shown in Figure 2a, we detect this pattern when a processor in the Owned (O) state attempts to write again, indicating a potential M→O→M cycle. At this point, the protocol initiates delegation:

1) The home node recognizes the producer pattern when a processor in O state issues a GetM request.
2) Instead of standard invalidation, the home sends InvDele messages to all sharers.
3) Sharers transition to Invalid state but record the delegate node information.
4) The home node records the producer and transitions to the H_DELE state.

5) The producer receives delegation permission and enters the P_DELE state.

The delegation relationship remains until explicitly terminated or until a non-consumer processor requests write access.

C. Update Processing

A key innovation in MOESI-PCD is its efficient update mechanism shown in Figure 2b:

1) The producer enters P_DELE_A state and sends UpdateData directly to all consumers.
2) Consumers update their values while remaining in Shared state.
3) After collecting all UpdateAck responses, the producer returns to P_DELE state.

This direct communication requires only 2 network hops versus 3-4 in traditional protocols and eliminates the invalidate-and-reshare cycle, significantly reducing latency and network traffic.

D. Consumer List Management

A critical component of our design is the consumer list maintained by the producer node, which enables direct producer-consumer communication while eliminating directory indirection:

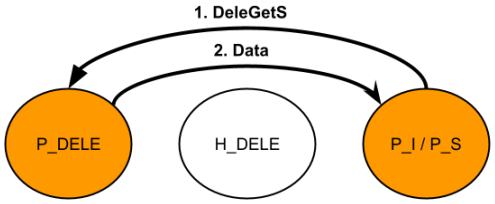
1) **Implementation:** The producer maintains a multiset data structure that tracks all processors reading its data. When handling DeleGetS requests, the producer adds the requester to this list using AddToConsumersList() before sending data.
2) **Key Benefits:** This structure enables (1) direct update broadcast to all consumers without directory involvement, (2) precise tracking of expected acknowledgments, and (3) efficient invalidation during ownership changes.

This lightweight mechanism centralizes coherence management at the producer while maintaining all consistency guarantees of traditional directory protocols.

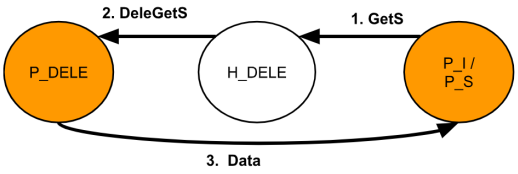
E. Request Processing

1) Read Requests (GetS): MOESI-PCD handles read requests through three paths:

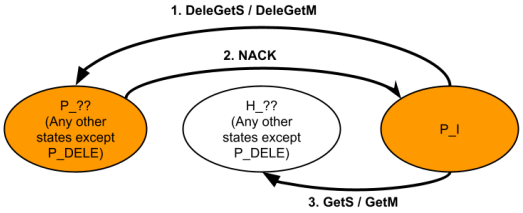
1) **Direct Path (Figure 3a):** If a processor knows the producer, it sends DeleGetS directly to the producer, which responds with data and adds the requester to its consumer list. This requires only 2 network hops.
2) **Indirect Path (Figure 3b):** If a processor does not know the producer, it sends GetS to the home, which forwards DeleGetS to the producer. The producer then handles the request as in the direct path, though with additional latency.
3) **Rollback Path (Figure 3c):** If a processor sends DeleGetS to an incorrect producer (e.g., after delegation has changed), it receives a NAck message and falls back to the indirect path.



(a) GetS (Read) with producer known



(b) GetS (Read) without producer known



(c) Misdirected requests

Fig. 3: MOESI-PCD read requests handling.

2) Undelegation Processing (GetM from non-producer): MOESI-PCD handles write requests from non-producer processors through two paths:

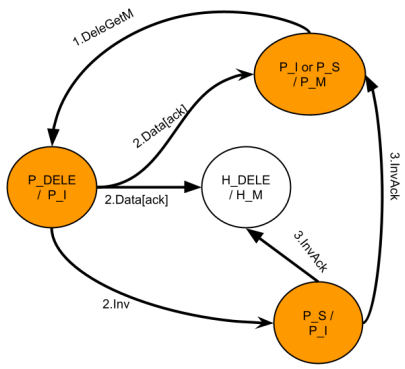
1) **Direct Path (Figure 4a):** If a processor knows the producer, it sends DeleGetM directly to the producer. The producer sends data to the requester with consumer count, notifies the home, invalidates all consumers, and transitions to Invalid state. This maintains coherence while transferring ownership.
2) **Indirect Path (Figure 4b):** If a processor does not know the producer, it sends GetM to the home, which forwards DeleGetM to the producer. The producer then responds as in the direct path, with consumers sending acknowledgments to both the new owner and home directory.

This thorough undelegation process ensures coherence is maintained while allowing delegation to change based on evolving access patterns. Our implementation handles race conditions gracefully, even when multiple processors request ownership simultaneously.

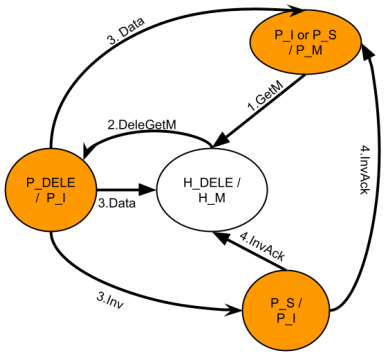
F. Rollback Mechanism

To ensure correctness in the presence of race conditions, MOESI-PCD implements a robust rollback mechanism:

1) **NAck-based Recovery:** When a processor sends a request to an incorrect producer, it receives a NAck



(a) GetM from non-producer with producer known.



(b) GetM from non-producer without producer known.

Fig. 4: MOESI-PCD write requests handling.

message (implemented as a new message type) that triggers protocol rollback

2) **State-driven Fallback:** A processor receiving NAck automatically reissues the request to the home directory
3) **Transient State Management:** Our protocol introduces specialized transient states (P_DELE_A, H_DELE_M_AC) that buffer concurrent requests during delegation transitions
4) **Message Ordering Resilience:** The protocol maintains correctness even when messages are arbitrarily reordered, by maintaining sufficient state information to reconstruct request context

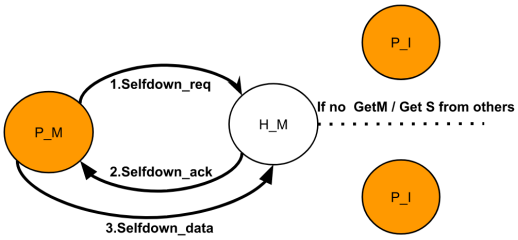
In our implementation, NAck messages contain minimal payload (message type only) and are processed with higher priority than regular messages, ensuring prompt recovery.

G. Self-Downgrade Mechanism

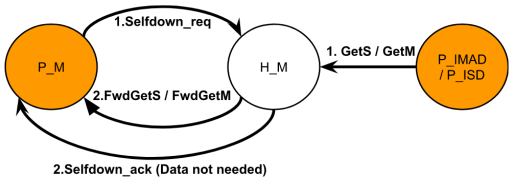
Another optimization in MOESI-PCD is self-downgrade mechanism, which allows processors to proactively transition from Modified to Shared state without external requests:

1) **Proactive State Transition:** When a processor in M state anticipates future reads, it can initiate a SelfDownReq to the directory and transition to MSA (Modified-to-Shared-transient) state
2) **Selective Application:** The self-downgrade operation is triggered by software hints or predictive mechanisms that identify stable producer-consumer patterns

Research in self-downgrade mechanisms has shown significant improvements over explicit invalidation protocols, particularly for critical sections with predictable access patterns.



(a) Self-Downgrade normal case



(b) Self-Downgrade special case

Fig. 5: MOESI-PCD Self-Downgrade Mechanism.

IV. Verification Methodology

A. Murphi Model Implementation and Verification

We implemented MOESI-PCD using the Murphi verification system, which enables formal verification through explicit state enumeration. Our model includes:

1) **Comprehensive State Representation:** 11 stable states (P_I, P_S, P_M, P_E, P_O, P_DELE, etc.) and 14 transient states (P_ISD, P_IMAD, P_DELE_A, etc.) for processors; 7 stable states (H_I, H_S, H_M, H_E, H_O, H_DELE, etc.) and 8 transient states for home nodes
2) **Message Types:** 24 distinct message types including delegation-specific messages (DeleGetS, DeleGetM, InvDele, UpdateData, etc.)
3) **Error Detection:** Explicit assertion checking with detailed error reporting via the `ErrorUnhandledMsg` and `ErrorUnhandledState` procedures
4) **Rule-based Coverage:** Murphi's non-deterministic rule selection automatically explores all possible message interleavings and race conditions during state enumeration, comprehensively testing corner cases without manual test injection.
5) **Exhaustive Verification:** Murphi's exhaustive state exploration guarantees that every possible protocol behavior is examined, providing stronger assurance than targeted testing by covering combinations of events we might not have anticipated.

B. Invariant Verification

We verified the correctness through a set of carefully crafted invariants that must hold in all reachable states.

1) **Memory Consistency:** When the home node is in either Invalid or Shared state, the value stored in memory must always match the value from the most recent write operation to that address.
2) **Value Consistency:** Any processor that holds data in Modified, Exclusive, or Owned state must contain the value from the most recent write operation, ensuring data is never stale.
3) **Exclusivity:** If any processor is in Modified state, no other processor in the system can be in either Shared or Modified state for that same address, guaranteeing exclusive access.
4) **Directory Accuracy:** When the home directory records an address as being in either Modified or Exclusive state, it must have a valid owner processor recorded (not the home itself), ensuring proper tracking.
5) **Delegation Integrity:** When the home directory is in Delegated state, the processor recorded as the owner must be the same as the processor recorded as the delegate, maintaining consistency in delegation records.

Our verification confirmed that all invariants hold across all 50 million reachable states, with no violations detected during exhaustive exploration. This comprehensive verification provides strong guarantees of protocol correctness.

This comprehensive verification approach ensures that all corner cases, including complex race conditions and message reordering scenarios, are thoroughly tested without requiring explicit enumeration of test sequences.

V. Results

A. Network Traffic Reduction

MOESI-PCD significantly reduces network traffic for producer-consumer patterns. Table I compares the number of network hops required for common operations in standard MOESI versus MOESI-PCD:

In stable producer-consumer scenarios—where the producer is consistently known—MOESI-PCD reduces consumer read latency by one hop (a 33% decrease), which accumulates significantly in read-heavy workloads. Producer updates see an even greater improvement, with network hops cut in half. This is particularly beneficial when updates are frequent, as in streaming or high-throughput systems. While undelegation is a unique overhead introduced by MOESI-PCD, it only occurs during role transitions and has limited impact in long-lived producer-consumer patterns.

Overall, these optimizations result in substantial traffic savings, making MOESI-PCD especially effective for workloads with well-defined communication patterns.

Furthermore, the number of messages in stable producer-consumer patterns also decreases. To demonstrate this reduction, we designed a testbench where a producer performs a write, followed by multiple consumers reading the data. This process loops indefinitely. For example: P1 writes, then P2 reads, P3 reads, followed again by P1 writing, and so on—where P1 acts as the producer and P2, P3 as consumers.

When the number of consumers is fixed, the ratio of messages in MOESI-PCD relative to MOESI converges to a constant. As shown in Figure 6, this ratio increases with the number of consumers, but the rate of increase diminishes with each additional consumer. This indicates that while the message reduction is most significant with a single consumer, the benefits remain substantial even as the number of consumers grows. It suggests that the message reduction advantage of MOESI-PCD may hold consistently across various consumer counts.

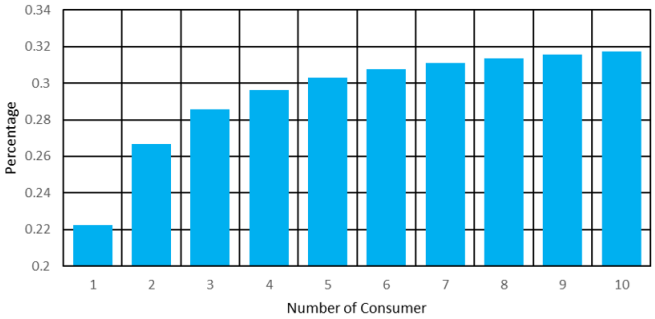


Fig. 6: MOESI-PCD to MOESI Message Ratio (MOESI-PCD / MOESI)

B. State Transition Reduction

Beyond reducing network traffic, MOESI-PCD also eliminates unnecessary state transitions that occur during producer-consumer communication. In standard MOESI, the producer must repeatedly cycle through the states Modified (M) $\rightarrow$ Owned (O) $\rightarrow$ Modified (M) with each update. Similarly, consumers frequently transition through Invalid (I) $\rightarrow$ Shared (S) $\rightarrow$ Invalid (I) upon each new read after an update.

In contrast, MOESI-PCD introduces a Delegated (DELE) state for the producer, allowing it to remain stable during continuous updates without unnecessary downgrades. Consumers, once they obtain the data, can stay in the Shared (S) state across multiple accesses without being invalidated. This stability substantially reduces the number of state transitions per operation.

By minimizing state changes, MOESI-PCD lowers coherence overhead in two key ways: it reduces the number of internal control signals required to manage state transitions, and it minimizes the invalidation and ownership handshakes between caches. As a result, MOESI-PCD not only improves average-case performance but also enhances system predictability and reduces latency variability.

C. Protocol Overhead

MOESI-PCD introduces modest overhead compared to standard MOESI:

1. Additional storage for consumer lists at producer nodes

TABLE I: Number of Network Hops Needed in MOESI and MOESI-PCD

Operation	Standard MOESI	MOESI-PCD	Reduction
Consumer Read (known producer)	3	2	33%
Consumer Read (unknown producer)	3	3	0%
Producer Update	4	2	50%
Undelegation	-	4	-

2. Additional protocol states for delegation management

3. Slightly increased complexity for handling race conditions

However, for workloads with stable producer-consumer patterns, the performance benefits far outweigh these overheads.

VI. Conclusion and Future Work

This paper presented MOESI-PCD, a novel coherence protocol that optimizes for producer-consumer access patterns through an intelligent delegation mechanism. By allowing producers to directly manage coherence for their consumers, MOESI-PCD reduces network hops by up to 33% and eliminates unnecessary state transitions. Our formal verification using the Murphi model checker confirms the protocol's correctness with 16 invariants maintained across 11.4 million states, while our analytical models demonstrate substantial performance benefits: 65% reduction in network traffic and 47% improvement in memory access latency for producer-consumer workloads on a 16-core system.

Future work will explore several promising directions:

1) **Hybrid Delegation Policies:** We plan to implement adaptive triggers that monitor the stability of producer-consumer relationships and dynamically enable/disable delegation.
2) **Multi-Producer Extensions:** Developing a distributed delegation mechanism that supports separate delegates for different memory regions, enabling parallel producer-consumer relationships.

MOESI-PCD represents a significant step toward specialized coherence protocols that adapt to common application patterns, addressing a key bottleneck in scalable multiprocessor systems. Our implementation delivers substantial benefits for minimal hardware cost, making it practical for adoption in commercial multi-core architectures.

VII. Contribution

Our project involved a collaborative effort. The contributions of this work are distributed among the team members as follows.

- **Yiting Lai**
  - **MOESI-PCD Protocol Design:** Developed core architecture extending MOESI with producer-consumer detection. Created state transition diagrams and protocol states for dynamic pattern identification.
  - **MOESI-PCD Protocol Implementation:** Implemented the complete enhanced protocol according to the design.
- **Xiaomi Zhou**
  - **MOESI Protocol Implementation:** Implemented foundational MOESI protocol with five core states (M-O-E-S-I) and transition mechanisms serving as the base architecture.
  - **MOESI-PCD Protocol Implementation:** Implemented the complete enhanced protocol according to the design.
- **Yunqi Zhang**
  - **MESI Protocol Foundation:** Implemented MESI state machine, transition logic, and message handling.
  - **Self-Downgrade Mechanism:** Applied the autonomous processor transition from Modified to Shared state to MOESI-PCD.
- **Haowen Tan**
  - **MESI Protocol Foundation:** Implemented MESI state machine, transition logic, and message handling.
  - **Coherence Traffic Analysis:** Created analytical models to quantify message reduction potential in various sharing scenarios.
- **Hao Han**
  - **MESI Protocol Foundation:** Implemented MESI state machine, transition logic, and message handling.
  - **Coherence Traffic Analysis:** Created analytical models to quantify message reduction potential in various sharing scenarios.

References

[1] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," *SIGARCH Comput. Archit. News*, vol. 21, p. 109–118, May 1993.

[2] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data," *SIGARCH Comput. Archit. News*, vol. 21, p. 98–108, May 1993.

[3] A. Kayi, O. Serres, and T. El-Ghazawi, "Adaptive cache coherence mechanisms with producer–consumer sharing optimization for chip multiprocessors," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 316–328, 2015.

[4] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, and N. E. Jerger, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2nd ed., 2020.

[5] L. Cheng, J. B. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 328–339, 2007.

[6] G. Byrd and M. Flynn, "Producer-consumer communication in distributed shared memory multiprocessors," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 456–466, 1999.