Adaptive Write-Update and Write-Invalidate Cache Coherence Protocols for Producer-Consumer Sharing

Bangjie Liu
Language Technologies Institute
School of Computer Science
Carnegie Mellon University
bangjiel@andrew.cmu.edu

Hao Li
Language Technologies Institute
School of Computer Science
Carnegie Mellon University
haol2@andrew.cmu.edu

***Abstract*—Shared memory multicore systems play crucial roles in scientific and enterprise applications. They are efficient in general applications but perform poorly in applications that establish producer-consumer sharing patterns under write-invalidate cache coherence protocol due to a large amount of cache invalidations and cache misses. Write-update protocol mitigates this issue by forwarding all updates on write operations but introduces extra network traffic overhead. In this project, we propose an adaptive cache coherence protocol that speculatively forwards data updated by producer to potential consumers when producer-consumer sharing patterns are detected to avoid unnecessary invalidations and memory accesses. We evaluate the proposed adaptive protocol on applications with varying degree of producer-consumer sharing patterns. As a result, it outperforms write-invalidate protocol by 2% — 21%.**

I. Introduction

Shared memory multicore systems play increasingly important roles in both scientific world and the industry. Cache coherence protocol has great influence on performance of shared memory multicore systems. Producer-consumer sharing refers to situations in multi-process synchronization wherein multiple processes share a common, fixed-size buffer and some processes, as known as producers, keep writing new data to that shared buffer while some readers keep reading data from it [6]. The most popular cache coherence protocol used in modern multiprocessor architecture is directory-based write-invalidate protocol, which is inefficient for producer-consumer sharing due to extensive invalidation traffics and expensive remote misses.

Researches have been done in this area, but they focus more on eliminating unnecessary hops as shown in Figure 1 by using a naive pattern detector and additional hardware for cache directory [1].

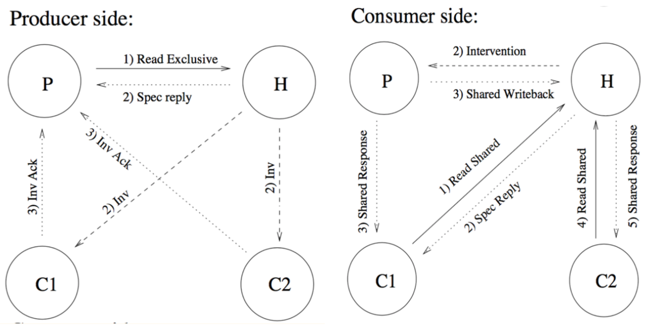

Fig. 1: Producer-Consumer Remote Misses

Instead of mitigating remote misses, this project focuses on mitigate the slowest part — memory accesses caused by invalidation. It assumes that communications among caches on different cores are physically feasible and contributes to the following:

- A sophisticated detector that detects producer-consumer sharing patterns at a fine-grained level without extra hardware support
- An adaptive cache coherence protocol optimized for producer-consumer sharing

The remainder of this report is organized as follows. We start in Section 2 with an overview of our goals and what we have accomplished. Section 3 describes our analysis of the performance of write-invalidate and write-update protocols on a representative producer-consumer application. In Section 4, we present our adaptive protocol, followed by a thorough evaluation on applications with varying degree of producer-consumer sharing in Section 5. Finally, we summarize our works and discuss future directions in Section 6 and 7.

II. Goals

[75%] Implement a cache simulator and testing tools (stack trace, logger, etc.) to fully evaluate and analyze the performance of directory-based write-invalidate and write-update protocols on a representative producer-consumer application.

[100%] Implement the proposed adaptive cache coherence protocol and producer-consumer sharing pattern detector. Evaluate and analyze the its performance on applications with varying degree of producer-consumer sharing.

[125%] Equip the cache simulator with functionalities that can synchronize application threads so as to observe more accurate cache access events (mitigate the effect of pintool instrumentation).

As we write this report, we have accomplished both 75% and 100% goals. And due to time limits, we decide to work on a more thorough analysis of the proposed protocol on benchmarks with varying degree of producer-consumer sharing patterns instead of the 125% goal.

III. Write-Invalidate and Write-Update Protocols Evaluation

In this section, we evaluate the performance, in terms of load hit rates, of write-invalidate and write-update protocols on a application that establishes strong producer-consumer sharing pattern. In the following evaluations, there are 1 producer and 2 consumers running on different cores. It is done on a Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz machine with 32KB L1 data cache. And we only simulate L1 data cache configured as Table I.

TABLE I: L1 Cache Configuration

Number of Sets	Associativity	Line Size	Interconnect
64	8	64B	Directory

A. Representative Application

In the application, there is a producer thread that keeps writing to a shared memory while there are multiple consumer threads reading for it. A global lock is used for synchronization as shown in Algorithm 1 and 2. A such self-defined application works well for evaluation purposes because it offers overall performance evaluation under different cache coherence protocols. And more importantly we can have a fine-grained investigation of the shared data of interest by outputting its memory address and look that up among all access records.

Algorithm 1 Producer Thread

```
1: data refers to a shared memory location
2: lock refers to the lock for data
3: while True do
4:     GetLock(lock)                ▷ get global lock
5:     Produce(data)
```

Algorithm 2 Consumer Thread

```
1: data refers to a shared memory location
2: lock refers to the lock for data
3: while True do
4:     GetLock(lock)                ▷ get global lock
5:     Consume(data)
```

B. Write-Invalidate Performance

Write-invalidate protocol works poorly on producer-consumer sharing patterns. As Figure 2 indicates, the overall load hit rate is acceptable. But for the shared data alone as shown in Figure 3, consumers' load hit rates are relatively low. The reason is that in write-invalidate protocol, write operations will invalidate all copies of this particular cache line and thus read operations after that need to read the latest one from either the owner or memory.

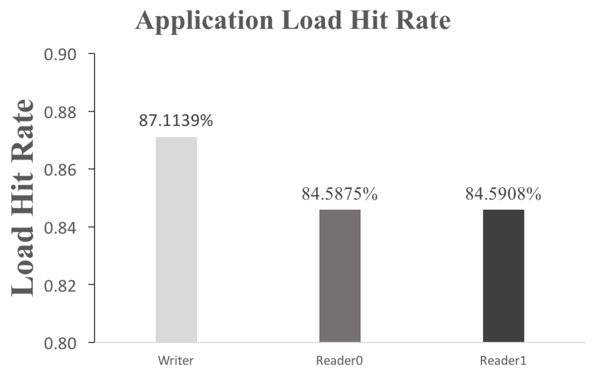

Fig. 2: Write-Invalidate Overall Load Hit Rates

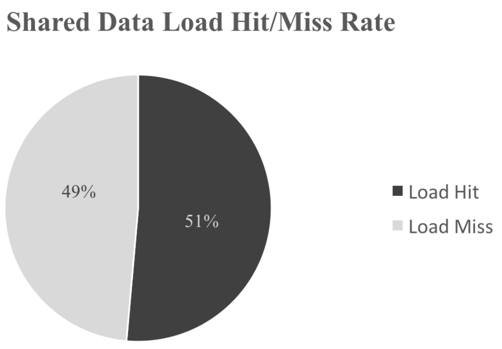

Fig. 3: Write-Invalidate Shared Data Load Hit Rates

C. Write-Update Performance

Write-update works well on producer-consumer sharing patterns. Figure 4 and 5 show that the overall performance is good and consumers have perfect load hit rates. This is because consumers now can directly get data from producer without accessing memory.

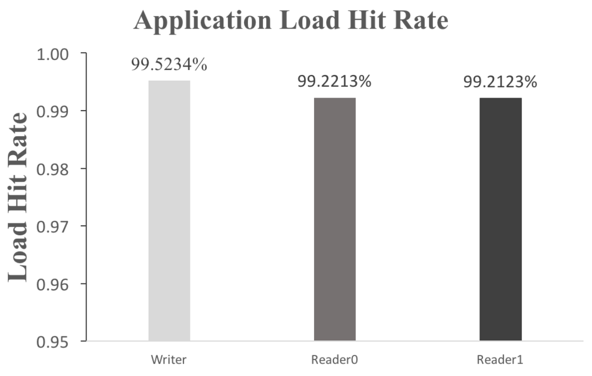

Fig. 4: Write-Invalidate Performance

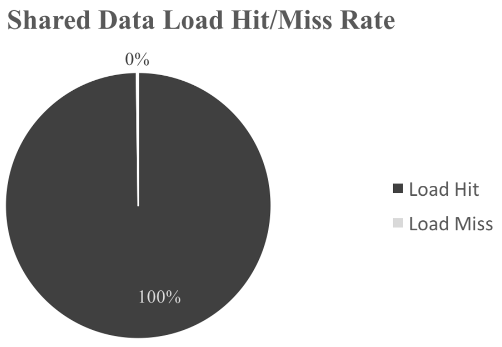

Fig. 5: Write-Invalidate Performance

IV. Adaptive Cache Coherence Protocol

In this section, we introduce an adaptive cache coherence protocol that intelligently switches between write-invalidate and write-update protocols. It can mitigate unnecessary memory access by speculatively forwarding data to consumers once producer-consumer sharing is detected. Producer-consumer sharing patterns are defined as such: when a thread reads a memory location more than once after a thread writes it, the reading thread is considered as one of the consumers of the writing thread for this particular memory location.

A. Detector

Additional bits are needed to track access history of each cache line for pattern identification. They are associated with directory lines which are extended to the structure shown in Figure 6.

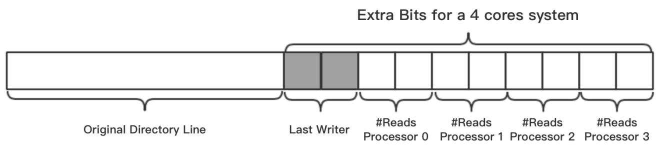

Fig. 6: Extended Directory Line Structure of a 4 Core Machine

The number of extra bits for each cache line is

$$logN + 2 * N$$

where N is the number of processors, $last\_writer$ tracks the last one to write this cache line, and there are 2 saturating bits for every processor to track the number of read operations on this cache line. Cache access will trigger updates on the extra bits as shown in Algorithm 3 and 4. Note that the extra bits will be discarded to save space when its associated directory line is evicted. However, as mentioned in Section 1, we assume large enough directory banks for all line records and so have not simulated the eviction of directory lines.

Algorithm 3 On Read Operations

```
1: Let $pid$ refer to a processor
2: for all $pid$ do
3:     SaturatingIncrease($pid$)          ▷ increase by 1
```

Algorithm 4 On Write Operations

```
1: Let $writer\_id$ be the processor performing writes
2: if $writer\_id$ == $last\_writer$ then
3:     for all $pid \neq writer\_id$ do                ▷ exclude itself
4:         if CountReads($pid$) $>= 1$ then
5:             Cache2CacheDataPush($pid$)              ▷ write-update
6:         else
7:             InvalidateDataCopy($pid$)               ▷ write-invalidate
8: else
9:     for all $pid \neq writer\_id$ do                ▷ exclude itself
10:        SaturatingDecrease($pid$)                   ▷ decrease by 1
11:        InvalidateDataCopy($pid$)                   ▷ write-invalidate
12:    UpdateLastWriter($writer\_id$)
```

B. Replacement Policy

We use LRU (Least-Recently-Used) cache replacement policy. It stays the same even when the detector and speculatively pushing are involved. In other words, the speculative forwarding has no effect on consumers' data locality.

C. Write Strategy

We use write-back allocate as write strategy. In our implementation, there are four cases when a write-back allocate is issued:

- In write-invalidate, when the cache line is in Modified state and receives a read request from other processors (not owner), the owner will issue a write-back allocate
- In write-invalidate, when the cache line is in Modified state and the owner's local cache copy is being evicted, a write-back allocate will be issued by the owner
- In write-update, when the cache line is in Modified state and being evicted, the producer will be responsible for issuing write-back allocate
- In write-update, when the cache line is in Modified state and last writer is updated (meaning another writer is modifying the shared data), the directory will randomly assign a sharer to issue write-back allocate.

Note that when producer speculatively forwards data to qualified consumers, the cache line switches to Modified state and both producer and consumer become the owner.

V. Adaptive Protocol Evaluation

In this section, we evaluate the performance of the proposed adaptive protocol on applications with varying degree of producer-consumer sharing. It is done on a Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz machine with 32KB L1 data cache. And we only simulate L1 data cache.

TABLE II: Environment Configuration

Architecture	CPU op-mode(s)	CPU(s)	Thread(s) per core
x86_64	32-bit,64-bit	16	2
CPU MHz	L1 cache	L2 cache	L3 cache
1200.000	32K	256K	20480K

A. Simulation Environment

We implement a Pintool-based cache simulator from scratch. It has three parts: cache, coherence protocol and profiler. Check *github* for more details.

1) Cache: We implement a configurable cache simulator with portable replacement policy module. For this project, we only simulate L1 data cache and it should be easy to extend to L2/L3 cache. The configuration is shown in Table III.

TABLE III: L1 Cache Configuration

Number of Sets	Associativity	Line Size	Interconnect
64	8	64B	Directory

2) Coherence Protocol: We implement directory-based MSI (Modified-Shared-Invalid) coherence protocol and simulate write-invalidate, write-update, as well as the proposed adaptive protocol based on it. To simplify the implementation, we assume that: (i) ring interconnection network in CPU; (ii) all read/write operations must first request home node before looking up local cache; (iii) directory bank is large enough to hold all line records.

3) Profiler: Our profiler provides detailed cache statistics for multi-thread programs. It monitors load/store hit/miss rate, execution cycles, network messages for each core and the entire program. To simplify the evaluation, we assign cost estimation for each type of operation, as is shown in Table IV.

TABLE IV: Cycle Cost Estimation

Access Mode	Number of Cycles
Local Cache Access	3
Remote Cache Access	7
Cache-to-cache Data Transfer (amortized)	4
Memory Access	100

B. Results

We simulate a 4-processor system and evaluate the performance of proposed adaptive cache coherence protocol on application with varying degrees of producer-producer sharing patterns. Table V shows the input data sets and configuration of the applications we use in the study [8].

TABLE V: Application Input Data Set

Application	Problem Size
Single	read/write operations on shared data
Fmm	two clusters, 512 particles
Barnes	163 nodes, 12 seed
Ocean	non-contiguous partitions, 34x34 grid
LU	non-contiguous blocks ,256x256 matrix, block size 16
Radix	256k (262,144) keys and a radix of 1024

Single is the representative application described in Section 3, in which there is a producer thread that keeps writing to a shared memory location while there are multiple consumers reading from it. It establishes strong single-producer-multiple-consumer sharing patterns. As a result, it achieves more than 20% speedup over write-invalidate protocol with less than 5% more network message and the cache hit rates increase by around 3%. In addition, the extra eviction cost needed for the proposed adaptive protocol is neglectable which means the improvement comes with very little overhead.

Fmm is from the SPLASH-2 benchmark suite [8]. It implements a parallel adaptive Fast Multipole Method to simulate the interaction of a system of bodies, as known as N-body problem [2]. As a result, it has similar performance with the base protocol. The 3% speedup comes with a similar overhead of eviction cost.

Barnes is from the SPLASH-2 benchmark suite [8]. It implements the Barnes-Hut method to simulate the interaction of a system of bodies, as known as N-body problem [2]. It can be stated as *Given the quasi-steady orbital properties (instantaneous position, velocity and time) of a group of celestial bodies, predict their interactive forces; and consequently, predict their true orbital motions for all future times. [7]* Note that the SPLASH-2 implementation allows for multiple particles to be stored in each leaf cell of the space partition and exhibit a stable producer-consumer sharing pattern [3]. As a result, the proposed adaptive protocol has similar performance with base protocol. The speedup is around 3% but it is worth noting that the network traffic and eviction cost are even lower than that in the baseline protocol.

Ocean is from the SPLASH-2 benchmark suite [8]. It is the non-contiguous partitions version and aims to solve boundary-value problem by simulating large-scale ocean movements based on eddy and boundary currents [4]. This version of Ocean partitions grids into square-like subgrids to improve the communication to computation ratio and exhibits single producer single consumer sharing patterns. As a result, the proposed adaptive protocol outperforms the base protocol. It achieves about 9% speedup. Moreover, the network traffic and eviction cost are even lower than that in the baseline protocol.

LU is a kernel application in SPLASH-2 benchmark suite [8]. It aims to factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The factorization uses blocking to exploit temporal locality on individual submatrix elements [5]. There is no obvious producer-consumer sharing patterns in it. And We use the non-contiguous block allocation implementation in this study. As a result, it has the same performance as the baseline protocol. There is no speedup and it introduces 1% more network traffic. However, the cache eviction cost is slightly lower than the baseline.

Radix is a kernel application in SPLASH-2 benchmark suite [8]. It is essentially an integer radix sort program that does not establish explicit producer-consumer sharing patterns. As a result, the proposed protocol outperforms baseline. It achieves around 16% speedup with even less network traffic. Furthermore, the eviction cost is less than baseline by around 2%.

C. Analysis

1) Speedup: As Figure 9 indicates, the adaptive cache coherence protocol improves performance by 2% — 21%, depending on varying problem size and degrees of producer-consumer pattern that programs exploit. For most benchmarks, the major performance benefits come from cache hits and speculative cache-to-cache update. Single-thread programs receive significant performance gain from locality. However, such benefit is traded at a big discount to multi-thread programs due to cache incoherence. Specifically, the incoherent cache copies of frequently-modified shared data become performance bottleneck, which is referred as producer-consumer pattern in our case. In Adaptive Cache Coherence Protocol, the detector identifies such pattern at very early stage and speculatively forwards update to each qualified consumer on write operations, which retains locality benefit of each thread at minimal cost. Therefor, Even a relative small increase of cache hit rate can result in performance speedup.

Note that we only simulate 32K L1 data cache in this project, and as the application size increases there is a diminishing return on performance speedup. For example, we evaluate Barnes benchmark with varying number of particles and seeds, and the results show that the proposed protocol achieves higher speedup on smaller problem size. It is possible that the size of shared memory saturates L1 cache, and thereby cache hit improvement is limited due to inevitable cache evictions. We further note that even for a stable producer-consumer sharing pattern, if frequently-modified shared date is frequently evicted from producer's local cache, the program will not benefit from speculative forwarding because each eviction incurs cache invalidations and multiple cache misses of consumer side.

2) Network Traffic: To investigate the extent to which speculative forwarding impacts network message, we measure the total number of network message and the average number of network message to serve a load. As Figure 7 and 8 indicate, compared to directory-based write-invalidate protocol, our proposed protocol does not incur noticeable network message overhead (-2% — 2%). It speculatively forwards data to qualified consumers, and dynamically updates consumers list on response from directory. From Table VI, the speculative forwarding does not increase average network message needed for a load, which implies that the Adaptive Coherence Protocol (ACP) is able to identify real readers for shared data and thereby minimize the network traffic.

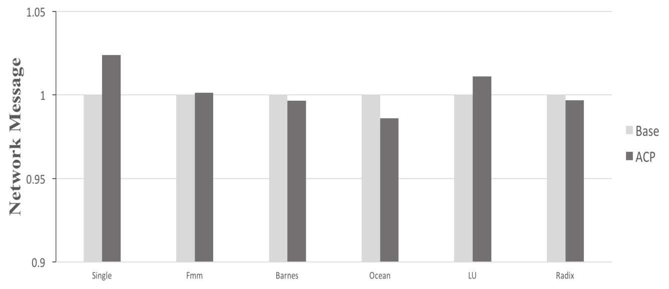

Fig. 7: Network Message

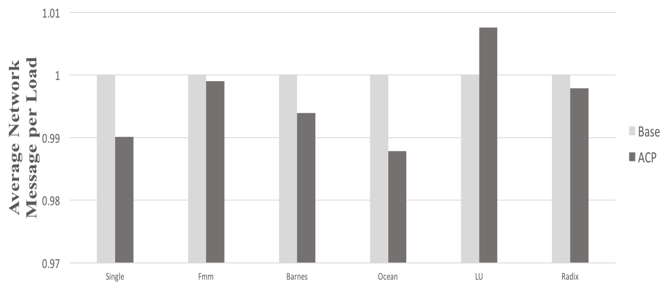

Fig. 8: Average Network Message per Load

TABLE VI: Average Network Message Needed to Serve a Load

	Single	Fmm	Barnes	Ocean	LU	Radix
Base	2.215	1.447	1.417	1.497	1.378	1.504
ACP	2.193	1.445	1.408	1.479	1.388	1.501

3) Write-back Delegation: As introduced in Section 4, we delegate the write-back responsibility to either producer or consumer depending on different situations. Consumer does not need to issue write-back on cache eviction because producer will keeping modifying shared data and forwarding to consumers. By delegating write-back responsibility to producer on cache eviction, we reduce network message and eviction overhead.

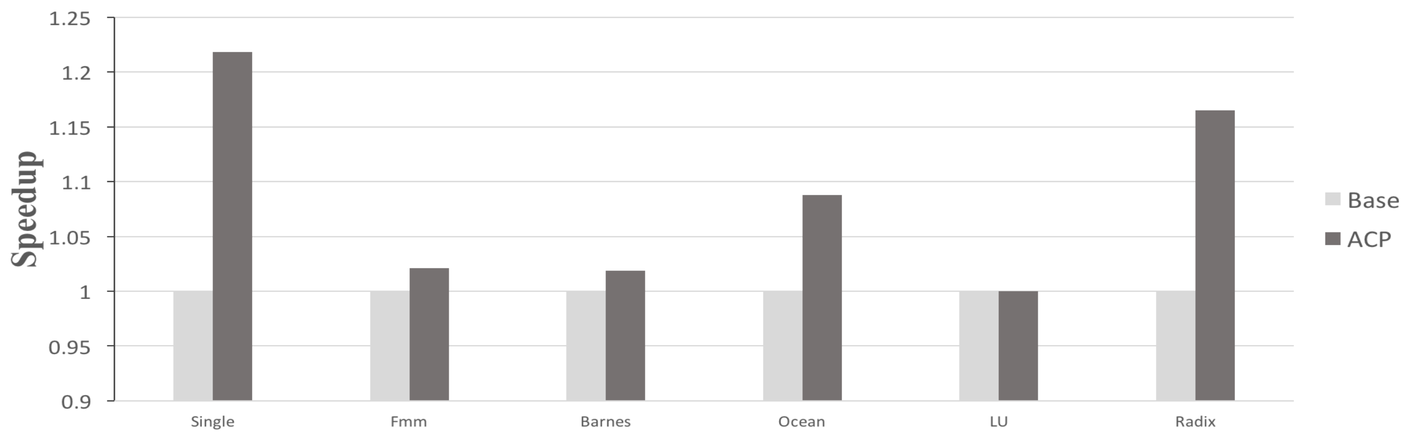

Fig. 9: Speedup

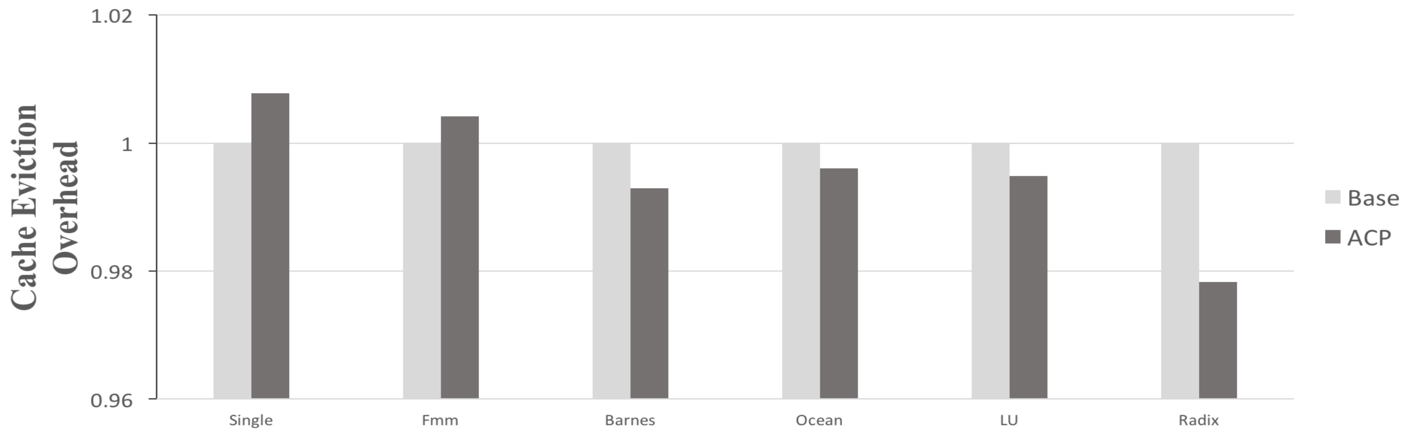

Fig. 10: Evict Cycles

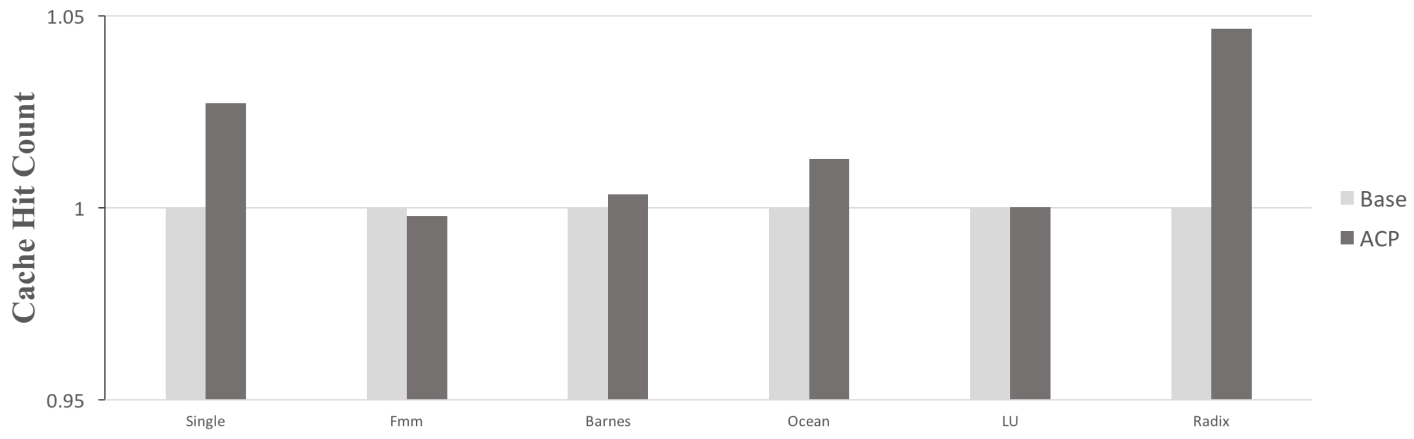

Fig. 11: Hit Count

VI. Summary

In this project, we implement a pintool-based cache simulator that supports multiple cache coherence protocol and offers detailed statistics of cache access behavior. In addition, under the assumption that cache-to-cache communication is physically feasible, we propose an adaptive cache coherence protocol optimized for producer-consumer sharing patterns that intelligently switch from write-invalidate to write-update protocol once stable producer-consumer sharing is detected. We evaluate the adaptive protocol on applications with varying degree of producer-consumer sharing patterns. As a result, the proposed adaptive protocol achieves noticeable speedup with very little network traffic overhead.

VII. Future Works

Pintool-based [9] cache simulator offers a fine-grained investigation of cache access behavior but it comes with a cost — the observation might deviates from the actual behavior due to the extra instructions instrumented by pintool. So one of the future works is to equip the cache simulator with functionalities that can synchronize application threads so as to observe more accurate cache access events to mitigate the effect of pintool instrumentation. In addition, we assume that directory banks are large enough to hold all line records. We propose in Section 4 that when directory lines are evicted, the extra bits for detector should be discarded to save memory space. We want to support this feature in the cache simulator.

References

[1] Cheng, Liqun, John B. Carter, and Donglai Dai. "An adaptive cache coherence protocol optimized for producer-consumer sharing."High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2007.

[2] Singh, J. P. Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors. PhD Thesis, Stanford University, February 1993.

[3] Holt, C. and Singh, J. P. Hierarchical N-Body Methods on Shared Address Space Multiprocessors. SIAM Conference on Parallel Processing for Scientific Computing, Feb 1995, to appear.

[4] Brandt, A. Multi-Level Adaptive Solutions to Boundary-Value Problems. Mathematics of Computation, 31(138):333-390, April 1977.

[5] Woo, S. C., Singh, J. P., and Hennessy, J. L. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), October 1994.

[6] https://en.wikipedia.org/wiki/Producer-consumer_problem

[7] https://en.wikipedia.org/wiki/N–body_problem

[8] http://www.capsl.udel.edu/splash/

[9] https://software.intel.com/sites/landingpage/pintool/docs/76991/Pin/html/