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 invalidation and memory reads. Write-update protocol mitigates this issue by updating all copies on write operations but introduces unnecessary traffic. In this project, we propose an adaptive cache coherence protocol that speculatively pushes data updated by producer to potential consumers when producer-consumer sharing patterns are detected to avoid unnecessary invalidation 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 in terms of cache hit/miss rates and cycles needed on applications with producer-consumer sharing patterns and general purpose applications.**

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.

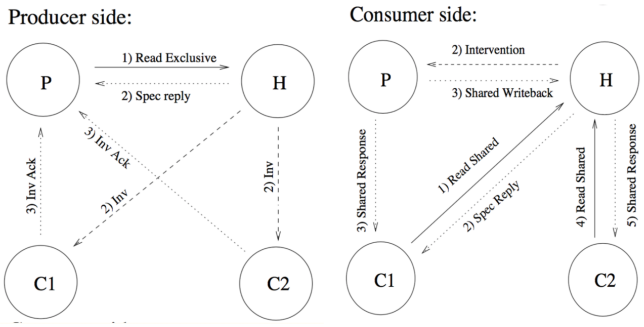

Fig. 1: Producer-Consumer Remote Misses

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]. 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
- An adaptive cache coherence protocol optimized for producer-consumer sharing patterns

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 several applications in Section 5. Finally, we summarize our works and discuss future directions in Section 6 and 7.

II. Goals

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

[100%] Implement proposed adaptive cache coherence protocols 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 multiple benchmarks instead of the 125% goal.

III. Write-Invalidate and Write-Update Protocols Evaluation

In this section, we evaluate the performance, in terms of load/store hit rates, of write-invalidate and write-update protocols on a representative application that establishes producer-consumer sharing patterns. 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.

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 as shown in Algorithm 1 and 2. A such self-defined application works well for evaluation purposes because it offers overall performance evaluation and more importantly a fine-grained investigation of shared memory behavior under different cache coherence protocols.

Algorithm 1 Producer Thread

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

Algorithm 2 Consumer Thread

```
1: data refers to a shared memory location
2: while True do
3:     GetLock()                    ▷ get global lock
4:     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 low due to a large amount of invalidation.

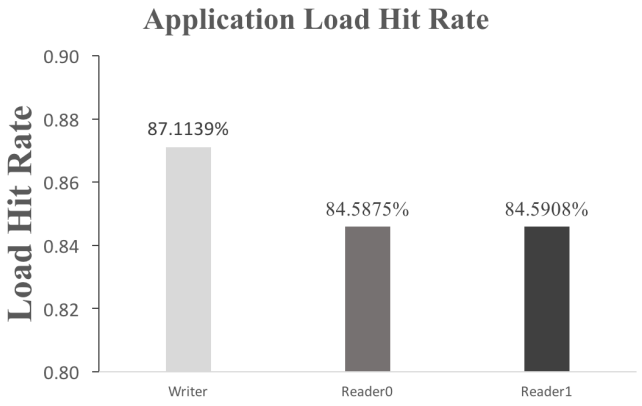

Fig. 2: Write-Invalidate Overall Load Hit Rates

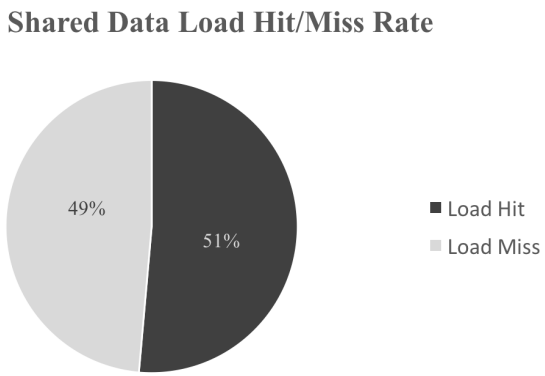

Fig. 3: Write-Invalidate Shared Data Load Hit Rates

C. Write-Update Performance

Write-update works well on producer-consumer sharing patterns. As Figure 4 and 5 shows, 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.

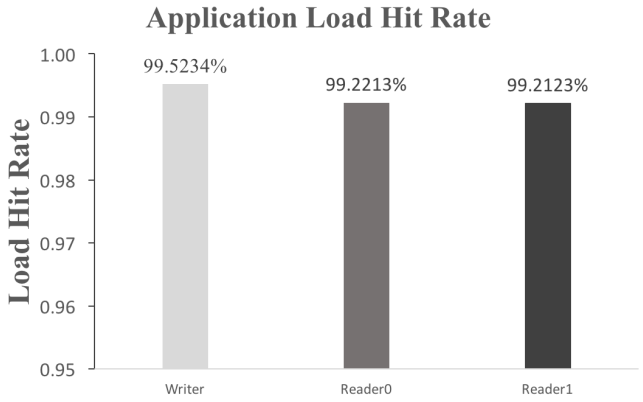

Fig. 4: Write-Invalidate Performance

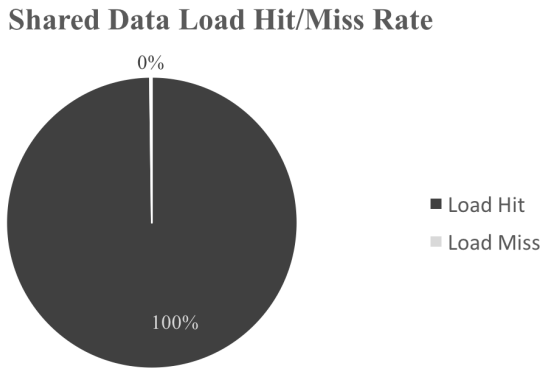

Fig. 5: Write-Invalidate Performance

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 accesses by speculatively pushing data to consumers once producer-consumer patterns are 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 threads are considered as consumers of the writing thread on 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.

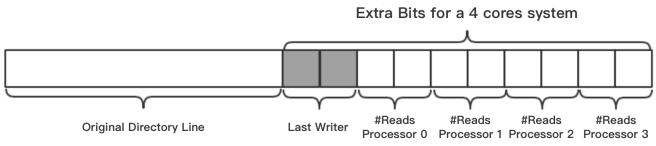

Fig. 6: Extended Directory Line Structure of a 4 Core Machine

The number of extra bits needed for every 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 accesses 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.

Algorithm 3 On Write Operations

```
1: Let writer_id be the processor performing writes
2: if writer_id == last_writer then
3:     for all pid ≠ 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:     SaturatingDecrease(pid)                   ▷ decrease by 1
10:    for all pid ≠ writer_id do                ▷ exclude itself
11:        InvalidateDataCopy(pid)               ▷ write-invalidate
12:        ClearCount(pid)                       ▷ reset read counts
13:    UpdateLastWriter(writer_id)
```

Algorithm 4 On Read Operations

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

B. Replacement Policy

We use LRU (Least-Recently-Used) cache replacement policy. When a processor receives forwarded data from producer, its cache replacement policy will not be updated. As a result, the speculative forwarding will not disrupt consumer's locality.

One thing to note is that to maximize the performance gain of speculative forwarding, we pin the producer's shared data copy to its local cache as shown in Algorithm 5. We notice 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. By pinning shared data to producer's local cache, frequently-modified shared data takes the first priority during cache replacement, thereby avoiding costly cache evictions.

Algorithm 5 On Write Operations

```
1: When cache state is Shared or Modified
2: Let writer_id be the processor performing writes
3: Let addr be the address of data of a processor write
4: if writer_id == last_writer then
5:     EnableCachePin(laster_writer, addr)
6: else
7:     DisableCachePin(laster_writer, addr)
```

C. Write-back Policy

We use write-back allocate as write-back policy. In our implementation, there are four cases where a write-back allocate will be 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.

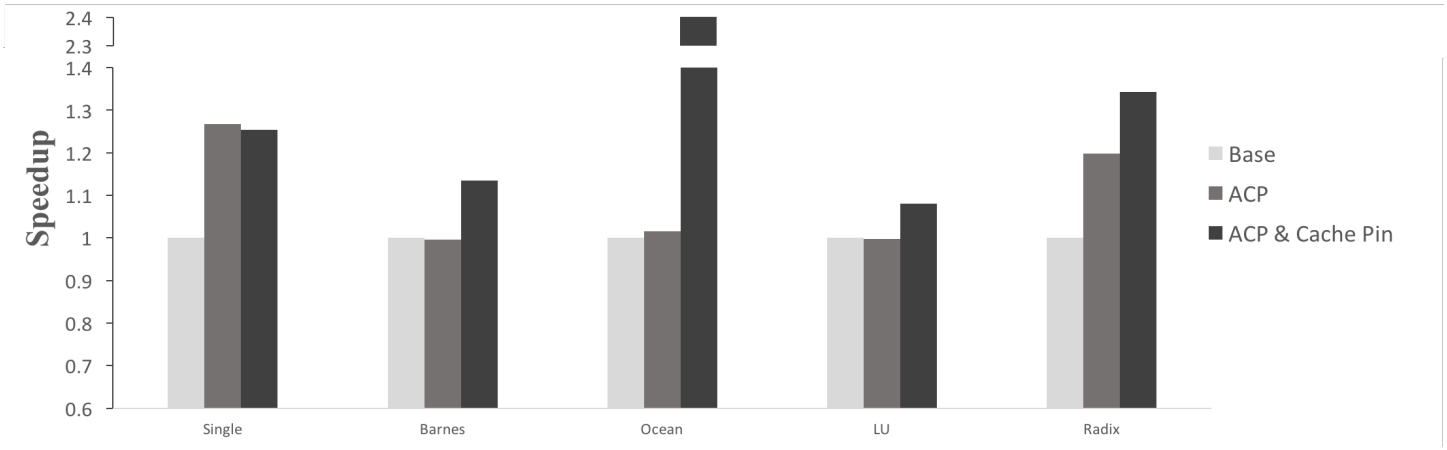

Fig. 7: Speedup

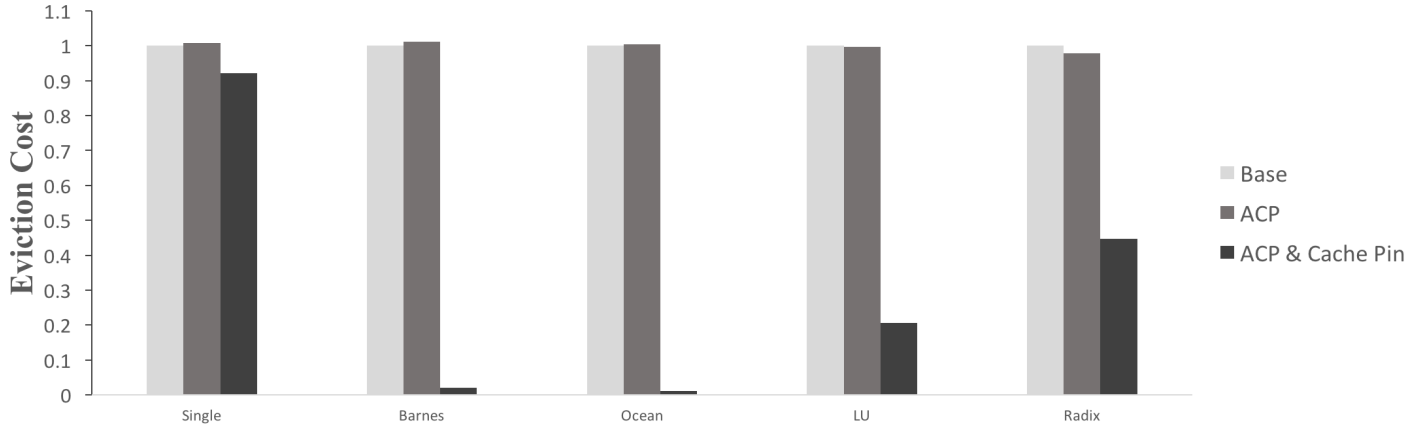

Fig. 8: Evict Cycles

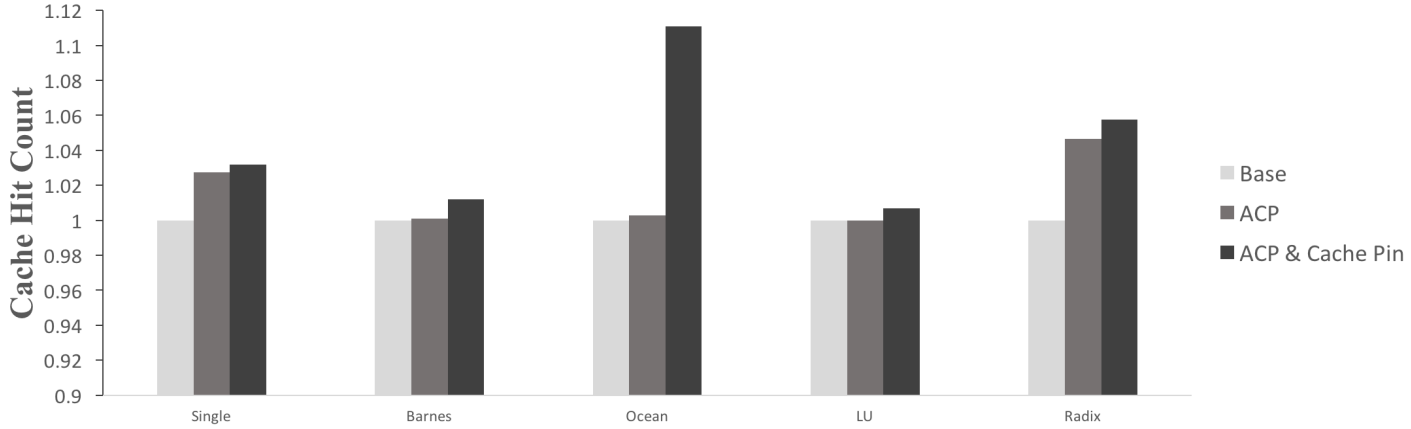

Fig. 9: Hit Count

V. Adaptive Protocol Evaluation

In this section, we evaluate the performance of the proposed adaptive protocol on various applications. 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.

A. Simulation Environment

We implement a cache simulator and coherence protocols including write-invalidate, write-update, as well as the proposed adaptive versions of directory-based MSI (modified-shared-invalid) from scratch. The number of processors, cache sizes, number of sets, associativity, and line size are configurable. And it currently only simulates L1 data cache and assumes directory banks are sufficient to hold all line records. In addition, it offers load/store hit rate statistics of the entire program, each core, and even all memory accessed. Besides, it supports cycle-accurate statistics by the cost estimation in Table I.

TABLE I: Cycle Cost Estimation

Access Mode	#Cycles
local cache access	3
remote cache access	7
cache-to-cache data transfer	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 II shows the input data sets and configuration of the applications we use in the study [8].

TABLE II: Application Input Data Set

Application	Problem Size
Single	read/write operations on shared data
Barnes	16384 nodes, 123 seed
Ocean	non-contiguous partitions, 258x258 grid
LU	non-contiguous blocks ,512x512 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 average network message per load operation needed for the proposed adaptive protocol is neglectable which means the improvement comes with very little overhead.

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. But if we pin cache lines in cache, it achieves about 15% speedup, higher cache hit rates with less than 3% more network traffic.

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 has similar performance with base protocol. However, the pin cache version substantially outperforms baseline. It achieves 240% speedup and 12% cache hits with little overhead.

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 does not outperform the baseline but introduce about 5% more network traffic.

Radix is a kernel application in SPLASH-2 benchmark suite [8]. It is essentially an integer radix sort program that does not establish producer-consumer sharing patterns. As a result, the proposed protocol outperforms baseline. It achieves around 20% speedup with even less network traffic. Moreover, the pin cache version works even better. The speedup is about 50% and the total network traffic is even less.

C. Analysis

1) Speedup: As Figure 7 indicates, the adaptive cache coherence protocol (with cache pin) improve performance by 11% — 240%, depending on varying problem size and degrees of producer-consumer pattern that programs exploit. For 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 update each qualified consumer on modification, which retains locality benefit of each thread at minimal cost. Therefor, Even a relative small increase of cache hit rate can result in significant performance speedup.

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 10 and 11 indicate, compared to directory-based write-invalidate protocol, our proposed protocol does not incur noticeable network message overhead (-5.3% — 3.5%). It speculatively forwards data to qualified consumers, and dynamically updates consumers list on response from directory. From Table III, the speculative forwarding does not increase average network message needed for a load, which implies that ACP is able to identify real readers for shared data and thereby minimize the network traffic.

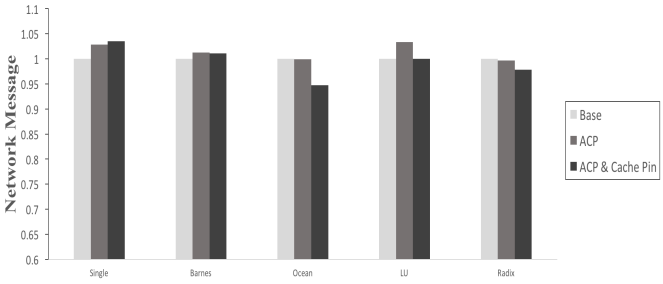

Fig. 10: Network Message

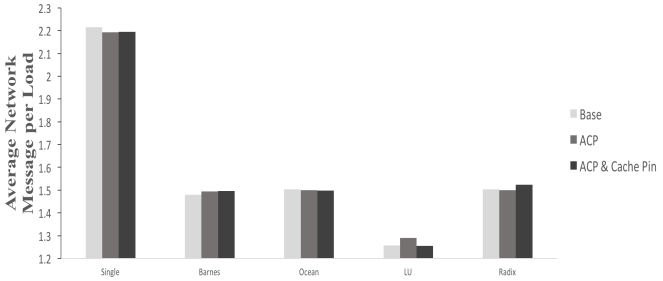

Fig. 11: Average Network Message per Load

TABLE III: Average Network Message Needed to Serve a Load

Benchmark	Base	ACP	ACP&Cache Pin
Single	2.215	2.193	2.195
Barnes	1.479	1.495	1.495
Ocean	1.504	1.500	1.499
LU	1.257	1.291	1.255
Radix	1.504	1.501	1.523

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 eviction, we reduce network message and eviction overhead.

VI. Summary

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/