# CSHARP: Coherence and SHaring Aware Replacement Policies for Parallel Applications

Biswabandan Panda
*Department of CSE, IIT Madras, India*
*Email: biswa@cse.iitm.ac.in*

Shankar Balachandran
*Department of CSE, IIT Madras, India*
*Email: shankar@cse.iitm.ac.in*

***Abstract*—Parallel applications are becoming mainstream and architectural techniques for multicores that target these applications are the need of the hour. Sharing of data by multiple threads and issues due to data coherence are unique to parallel applications. We propose CSHARP, a hardware framework that brings coherence and sharing awareness to *any* shared last level cache replacement policy. We use the degree of sharing of cache lines and the information present in coherence vectors to make replacement decisions. We apply CSHARP to a state-of-the-art cache replacement policy called TA-DRRIP to show its effectiveness. Our experiments on four core simulated system show that applying CSHARP on TA-DRRIP gives an extra 10% reduction in miss-rate at the LLC. Compared to LRU policy, CSHARP on TA-DRRIP shows a 18% miss-rate reduction and a 7% performance boost. We also show the scalability of our proposal by studying the hardware overhead and performance on a 8-core system.**

***Keywords*-Last-Level Cache, Replacement algorithm, Parallel Applications**

## I. Introduction

Applications from recognition, mining and synthesis world (RMS) are in the mainstream of computing and many of these applications are multithreaded in nature. Chip multiprocessors (CMPs) play an important role in improving the performance of these applications. In shared memory CMPs, all the cores compete for the last level shared cache. A good cache replacement policy can reduce the miss rate and therefore improve performance of the multithreaded applications. It can also reduce off-chip bandwidth requirement and power. Studying last level cache replacement for parallel applications is thus an important research problem.

State-of-the-art cache replacement policies such as [1], [3], and [2] target multiprogrammed workloads where different applications do not share data. All the cores contend for the shared last level cache. However, for parallel applications, when different threads run on different cores, there are two major issues:

- Threads may share data with each other. Sharing of data across different cores and different levels of cache requires support from coherence protocols to maintain consistency of data .
- Threads can be friendly to each other or contend with each other. If threads use the same shared data, cache lines are shared (called constructive sharing). Threads may also evict each other's cache lines (called destructive sharing) because of the associativity in the last level cache.

These issues are peculiar to parallel applications. This motivated us to propose a cache replacement policy that targets truly parallel applications.

The contributions of the paper are listed below.

- We make a case for using coherence and sharing information in the replacement decisions at LLC to target multithreaded applications.
- We present CSHARP, a generic **hardware** framework to add coherence and sharing awareness to *any* existent replacement policy such as LRU, TA-DRRIP [1]with negligible hardware overhead.

For the first time in literature, we also compare two state-of-the-art cache replacement policies (TA-DRRIP [1] and seg-LRU [2]) on multithreaded applications.

## II. Background

In this section, we present the background material on sharing patterns and coherence protocols. A thorough understanding of these two aspects is necessary to understand how multithreaded applications are different from multiprogrammed workloads.

### A. Sharing Patterns Across Threads

Multithreaded applications exchange data using shared memory locations. The behavior of the data exchange is called *sharing pattern*. Based on the read/write accesses and the threads that are making these accesses, the sharing patterns can be classified into three categories [7]: (i) read-only sharing, (ii) migratory sharing, and (iii) producer-consumer sharing. Let $R_i$ ($W_i$) denote a read (write) request of some cache line from thread $i$.
(1) Read-only sharing: The regular expression for a sequence that exhibits read-only sharing is

$$(R_i|W_i)^+(R_{j,j\neq i})^+$$

(2) Migratory sharing: The regular expression for a sequence that exhibits migratory sharing is

$$R_i^+ W_i (R_i|W_i)^* R_{j,j\neq i}^+ W_j (R_j|W_j)^* \cdots$$

(3) Producer-consumer sharing: The regular expression for a sequence that exhibits producer-consumer sharing is

$$W_i^+ R_{j,j\neq i}^* \cdots$$

Sharing pattern is the characteristic of only a slice of a program. A single multithreaded application can exhibit several sharing patterns at different phases of its execution. Since all the sharing patterns involve writes, data coherence becomes an issue.

### B. Data Coherence

Multicore architectures maintain private L1 data caches for each core. Local modification done by a core to a data line in its L1 cache can result in an inconsistent view of the shared memory location for other cores. This is because the other cores may have maintained their own local copies of the data in their L1 cache. This inconsistency is called the data coherence problem. Cache coherence protocols enforce `single-writer-multiple-reader` (SWMR) invariant [8]. A widely used coherence protocol is the `MOESI` protocol.

### C. Cache Replacement

Fundamental to any cache replacement policy is the concept of reuse of a cache line. If there is a cache miss, some existing cache line has to be evicted to make way for the missed line. Among the cache lines that are in a set, the line that is going to be reused furthest in the future is a candidate for eviction. An ideal replacement policy such as Belady's OPT [10] always "knows" which line will be reused the furthest and will evict it. However, this "knowledge" of reuse is impossible. Practically, cache replacement policies "predict" the line which has the furthest reuse and evict it. This is usually done with the help of a *reuse-register* that is associated with each line.

In general, the values assigned to the reuse-registers are picked from the range $[R_{min}, R_{max}]$ and the range varies across policies. Higher the value of the reuse-register of a line, further is the time for potential reuse. On a miss, all the reuse registers in a set are considered and the one with the highest value is usually evicted.

## III. Motivation

We provide two motivating factors for this work. We first motivate the need for assessing "true" multithreaded applications. We later show how typical multithreaded benchmarks share data, a factor that is not considered by the existing methods. Finally, we make a case for using coherence states by showing the states of cache lines for typical sharing patterns. Using the observations made here, we motivate and propose our replacement framework. We first draw attention to the differences between multiprogrammed workloads and multithreaded applications based on occupancy time of cache lines. The time period between the time of insertion and the time of eviction or invalidation of a cache line is called the "occupancy time" of the cache line. In a multiprogrammed setup, a cache line is used only by one program. However, in a multithreaded application, a cache line that is inserted can be used by more than one thread during its occupancy time.

**In multiprogrammed workloads, the sharing of LLC is always of the "destructive sharing" type. In a multithreaded application, the sharing of LLC by the threads can be "constructive" as well as "destructive".**

### A. Data Sharing in PARSEC

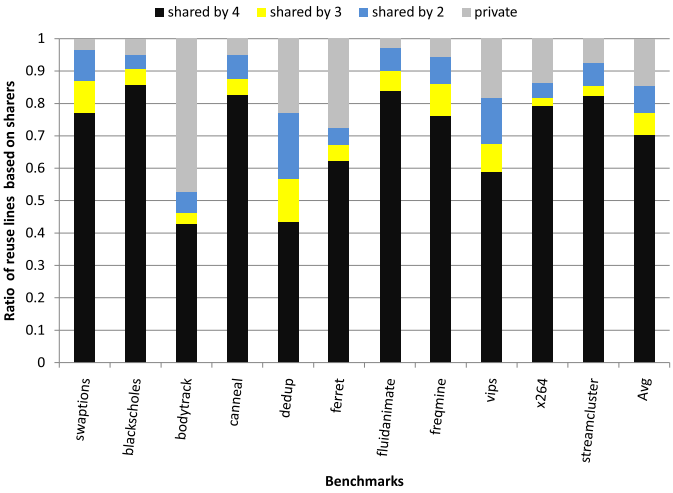


Figure 1: Reuse of cache lines for a 4 core system

PARSEC [5] is a benchmark suite of multithreaded applications targeting shared CMPs. We studied the behavior of programs in their parallel regions which form the Regions of Interest (ROI). We ran PARSEC benchmarks on a 4-core simulated system and present some observations on how data is shared and used in the ROI.

For a given cache line, the number of accesses made beyond the first access is called the "reuse" of the line. Some cache lines may be brought in and used just once before they got evicted. Such lines are said to have zero reuse. All the other lines are called "reused lines". A line that is reused by only one thread is called a *private* line. A *shared line* is a line reused by more than one thread.

In Figure 1, we plot the percentage of reused lines classified based on the number of different threads that accessed them. We can see that only 15% of the reused lines are private lines. In all the benchmarks, shared lines constitute the majority fraction of reuse. This leads us naturally to the question:

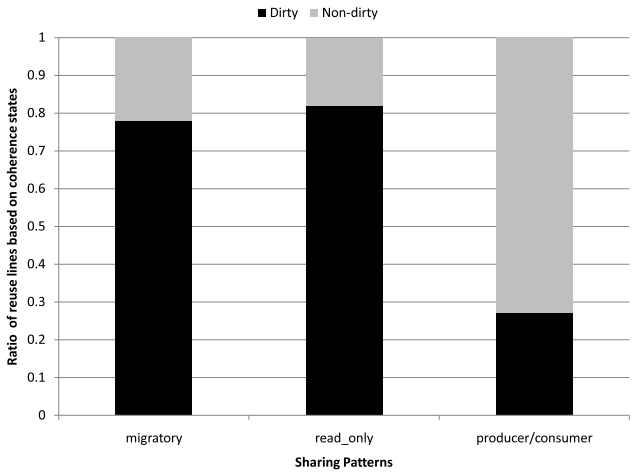


Figure 3: Reuse of Dirty and Non-dirty Lines for different sharing patterns

**Question 1: Is there any way to use the sharing information in making LLC decisions? If so, how?** Yes. Shared lines could be kept for more time in the LLC than the private lines to reduce the LLC miss rate.

### B. Coherence States in Multi-threaded Applications

To make a case for using coherence states, we show how the coherence states of cache lines change for the different sharing patterns that we discussed in Section II-A. We take a single cache line and trace the changes in coherence states in Figure 2. In Figure 2(a), the cache line is read-only shared. If the first access is $R_1$ ($W_1$), the cache line moves to the `E`(`M`) state. On $R_2$, the line moves to `S`(`O`) state. The line remains in the same state if further requests are all reads. Figure 2(b) shows a line that is shared in a migratory pattern. The line starts with state `E`, moves to `M` and toggles between `O` and `M` henceforth. Figure 2(c) shows a line that is shared in a producer-consumer pattern where the line moves to the `O` state and stays there till the next write from the producer. Thus, if an application shows any sharing at all, the cache lines are more likely to be in `O` , `M` or `S` states.

We further analyzed the reuse of cache lines according to their coherence states. In Figure 3, we plot the percentage of reused lines classified based on the coherence states and the program's sharing pattern. The plots are aggregated across all the benchmarks of each sharing-pattern. Except for the only producer-consumer benchmark in the suite (`fluidanimate`), dirty cache lines (lines with `O` or `M` states) are more reused than non-dirty cache lines (lines with `E` and `S` states). These two observations on coherence states, lead us to another question.

**Question 2: Is there any way to prioritize lines based on the coherence state for improving LLC decisions? If so, how?** Yes. Dirty lines have high reuse and could be prioritized over non-dirty lines.

### C. Coherence and Sharing Awareness

Answer to Question 1 brings sharing awareness and answer to Question 2 brings coherence awareness. Combining these two, we suggest that the replacement policies retain the cache lines using the following order of priority:

`O > M > E > S > I`

Because of the dirtiness and the high reuse associated with the `O` and `M` lines, we assign these lines the highest priority. Cache lines in `S` state denote sharing but the corresponding reuse-counts are low and hence is assigned low priority. Invalid lines do not contain any useful data and get the least priority.

Our coherence-and-sharing aware replacement framework is based on the philosophy that

*If cache lines evicted in the reverse order of this prioritization, cache misses at the L2 level may decrease and therefore improve program performance*.

## IV. CSHARP (CSHARP): Adding Coherence and Sharing Awareness

We present CSHARP, a framework to add **C**oherence and **SH**aring **A**wareness to **R**eplacement **P**olicies that target parallel applications at the LLCs. CSHARP assigns priorities to cache lines based on sharing and coherence information and tries to retain high priority lines at the LLC as much as possible.

CSHARP is a framework that sits on top of an existing replacement policy and we use the notation CSHARP(`policy name`) to reflect this thought process. When we apply CSHARP on a policy or a sub-policy we call it CSHARPing.

### A. Sr and Pr Lines

At a given instance of time, CSHARP defines a cache line to be in one of two categories: `Sr` or `Pr`. To distinguish between Sr and Pr lines, each cache line is associated with a single bit register called $Sr/\overline{Pr}$. On the very first access to a cache line, CSHARP sets $Sr/\overline{Pr} = 0$. On a subsequent access if the number of sharers in the coherence vector is more than one or if there is change in the ownership, we set $Sr/\overline{Pr}$ to 1. CSHARP uses the $Sr/\overline{Pr}$ bit to track the sharing behavior of a cache line until the cache line is evicted or invalidated.

### B. CSHARPing Three Sub-policies

Any cache replacement policy consists of three sub-policies: i) eviction, ii) insertion and iii) promotion. These three sub-policies can be explained in terms of reuse-register values which was introduced in Section II-C. On a cache miss, a line is "chosen" for eviction based on the reuse values. The new line is "inserted" into the set and is assigned an appropriate reuse-register value. On subsequent

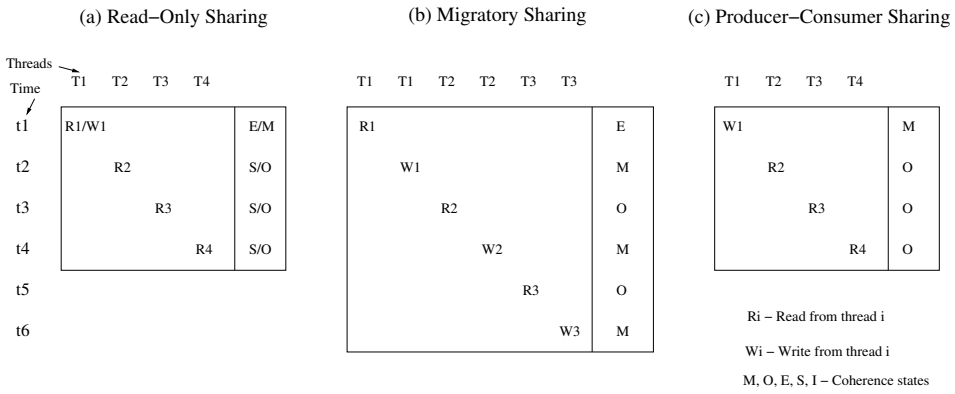


Figure 2: Coherence State of a Cache Line Under Different Sharing Patterns

hits, the line gets "promoted" by reducing the reuse-register value.

**CSHARPing Eviction Sub-policy:**
We define the eviction window as the last $w$ ways of a set. Lines in the eviction window are considered for eviction. Within the eviction window, CSHARP evicts the line with the highest reuse-register value. If there is a tie among many lines, CSHARP will pick a line in the following order of tie-breaking criteria: i) lines that are in `S` state ii) `Pr` lines that are in `E` state iii) `Pr` lines that are in `M` state and finally iv) other `Sr` lines.

**CSHARPing Insertion Sub-policy:**
At the heart of CSHARP are two Bloom filters, **BF1** and **BF2**, that track the recently evicted lines and their owners. A Bloom filter is a probabilistic data structure which is space efficient and is used to test the membership of an element in a set. BF1 is indexed using a combination of tag and index information of a cache line. *BF1 is used to track cache lines that were evicted recently*. If the lookup of a cache line on BF1 fails, the cache line has not been evicted in the recent past and is treated as a "new" cache line – a line that has not been seen before. BF2, on top of tag and index fields of the cache line, uses the owner field of the line. *BF2 tracks the previous owners of the line before the line was evicted*.

Since there is no way of knowing whether the to-be-inserted line is going to be accessed by multiple threads or not, we consider the likelihood of it being a `Sr` line. If the line is more likely to be a `Sr` line, we call the new line a `mSr` line. The prefix `m` stands for *maybe*. If the line is more likely to stay as a `Pr` line, we call it a `mPr` line. We show how we can label a line as `mSr` or `mPr` based on the outcomes of the lookups on the BFs.

A lookup of a to-be-inserted cache line with tag $T$ and index $I$ on BF1, and the requesting thread-id $Q$ along with $T$ and $I$ on BF2, can result in four possible outcomes. The implication for the four cases are shown in the last column of Table I.

Table I: The Implication of Lookup of a Cache Line on Bloom Filters BF1 and BF2

| Name | On BF1 | On BF2 | Interpretation | Label |
|---|---|---|---|---|
| Case 1 | Miss | Miss | Line not evicted recently | `mPr` |
| Case 2 | Miss | Hit | False Positive | `mPr` |
| Case 3 | Hit | Miss | Evicted recently but not by $Q$ | `mSr` |
| Case 4 | Hit | Hit | Evicted recently and by $Q$ | `mPr` |

CSHARPing the insertion sub-policy is to set the reuse-register values of `mPr` lines to be closer to $R_{max}$ and the values of `mSr` lines closer to $R_{min}$. This will try to keep the `mSr` lines longer in the cache than the `mPr` lines. Once the reuse register values are set, the labels are discarded. The choice of the actual values to put in to the reuse registers are left to the implementation. CSHARPing at the insertion can be nullified by the implementation if the same reuse-register values are assigned to both `mPr` and `mSr` lines.

False positives are possible in the last three cases shown in Table I. Cases 2 and 4 take a conservative stance and Case 3 takes a liberal stance on the state of the cache line.

**CSHARPing Promotion Sub-policy:**
Promotion sub-policy kicks in only on a hit of a cache line. At this juncture, the cache line could be a `Pr` or a `Sr` line. Promotion is achieved by setting the reuse-register value to $\rho$. CSHARPing the promotion sub-policy is to pick the value of $\rho$ based on the line and the thread that is making the request. For a `Pr` line which gets a hit from the same thread as the current owner, $\rho$ is set closer to $R_{max}$. If the thread that is making the request is different from the owner or if the line is a `Sr` line, $\rho$ is assigned a value closer to $R_{min}$. As before, the effects of CSHARP can be nullified by setting the same value in both cases.

### C. Working of CSHARP

We explain the working of CSHARP by considering the sequence of events after a cache miss at the LLC. Let thread

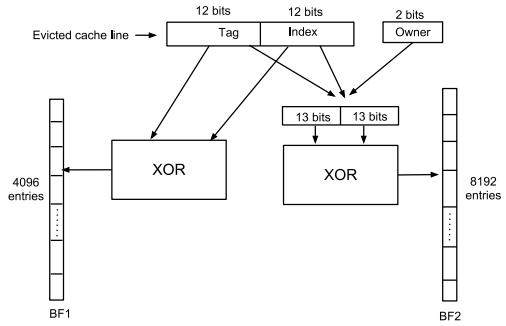


Figure 4: Structure of BF1 and BF2 for a 4-core 4MB LLC

$Q'$ be the requester for the new cache line. Let the cache line chosen for eviction by CSHARPed eviction sub-policy be $l$, with tag $T$ and index $I$. We make an entry in BF1 using $T$ and $I$, and an entry on BF2 using the owner field along with $T$ and $I$. The new line $l'$ is ready to be brought in. The CSHARPed insertion sub-policy decides the reuse-register values for $l'$. $Sr/\bar{Pr}$ is set to 0. The thread can now continue. CSHARPed promotion sub-policy will be brought to action whenever there are hits.

## V. Implementation of CSHARP

We first describe the implementation of the hardware structures namely BF1 and BF2. The implementation is illustrated in Figure 4 for a 4-core simulated system with 4MB 16-way LLC, line size of 64B, and 1GB of physical memory. We use the bitwise XOR of tag and index bits to index BF1 (with bit stuffing to match the widths, if necessary). BF1 contains 4096 entries. We concatenate the tag, index and owner field (and stuff 0's to make the size even) and perform bitwise XOR of the top half and bottom half of this to index BF2. BF2 contains 8192 entries. Due to the false positives, we reset both BF1 and BF2 after every 2048 evictions by using a 11-bit eviction counter. The hardware overhead due to the BFs and the counter is presented in Section VI-E.

We took TA-DRRIP [1] as a case study for introducing CSHARP. We chose TA-DRRIP over other policy such as seg-LRU [2] because TA-DRRIP is thread-aware and achieves good performance with less hardware overhead. We discuss the implementation of CSHARP(TA-DRRIP).

### A. TA-DRRIP

DRRIP predicts the re-reference interval of each cache line by using a $M$ bit reuse-register called the Re-Reference Prediction Value (RRPV) register. In practice, the value of $M$ is 2 and RRPV ranges from 0 to 3. It retains the cache lines with smaller RRPVs and evicts the cache lines with the larger RRPVs. DRRIP uses set-dueling to chose between two polices: SRRIP and BRRIP. Cache lines with RRPV=3 are searched from way-0 and the first such line is evicted. If no cache line with RRPV=3 is found, it increments the value of RRPV until a cache line has RRPV=3 and the process is repeated. SRRIP inserts cache lines with RRPV=2. BRRIP inserts 95% of cache lines with RRPV=3 and the rest with RRPV=2. On a hit to a cache line, the cache lines are promoted by setting RRPV to 0. The prefix *TA* in TA-DRRIP stands for "Thread Aware". In TA-DRRIP, each thread decides whether it should use SRRIP or BRRIP independently with the help of policy selection counters called PSELs.

### B. CSHARP(TA-DRRIP)

We CSHARPed eviction, insertion and promotion eviction policies of TA-DRRIP.

**Eviction Policy:** TA-DRRIP can evict a line from any of the ways in a set. We set the eviction window size to the number of ways in a set (i.e. all lines are candidates for eviction). Within the eviction window, CSHARP evicts the cache line with RRPV=3. It starts a search for `S` cache lines and evicts the first such line In the absence of such a line, it picks a `Pr` line (`E`) with RRPV=3 and in the absence of (`E`) lines with RRPV=3, it picks a line with `M` with RRPV=3. If there are no `Pr` lines, it evicts the first `Sr` line with RRPV=3 starting the search from the way=0. If there are no cache lines with RRPV=3, it increments the RRPV values until there is at least one line with RRPV=3.

**Insertion Policy:** CSHARP modifies the RRPV values assigned by TA-DRRIP. `mSr` lines are inserted with RRPV=1 and `mPr` lines are inserted with RRPV=2 or 3 based on whether SRRIP or BRRIP was used by the thread.

**Promotion Policy:** CSHARP sets RRPV to 1 for a `Pr` line which gets a hit from the same thread as the current owner. If the thread that is making the request is different from the owner or if the line is a `Sr` line, CSHARP sets RRPV to 0. We summarize CSHARPing of TA-DRRIP in terms of the sub-policies in Table II.

Table II: Summary of CSHARP(TA-DRRIP)

| Sub-policies | CSHARPed Decisions |
|---|---|
| Eviction | Lines with RRPV=3 in the following order:<br>`Sr` with `S` before (`Pr` with `E` before `Pr` with `M`) before `Sr` |
| Insertion | if `mSr` then RRPV=1,<br>else (RRPV=2 for SRRIP, 3 for BRRIP) |
| Promotion | if `Sr` or (`Pr` and change in ownership) then RRPV=0<br>else RRPV=1 |

## VI. Experiments and Results

We first explain the simulation framework and the characteristics of the benchmarks in this section. We then show our results and provide analysis of the results.

### A. Simulation Framework

We added CSHARP features to gem5 [4], a full system simulator. We also added TA-DRRIP and seg-LRU [2] to gem5.

We show the effectiveness of CSHARP using 4-core and 8-core CMPs. Table III lists the parameters used in the simulated system.

Table III: Parameters of Simulated Machine

| Processor | ALPHA 21264 |
|---|---|
| Fetch/Decode/Commit width | 8 |
| ROB/LQ/SQ/Issue Queue | 192/96/64/64 entries |
| TLB | SW Managed, 256 entries |
| L1 D/I Cache | $32KB$, 4 way, 2 cycle latency |
| L2 Unified Cache | #cores * 1 MB, 16 way, 16 cycle latency, Inclusive |
| Line size | 64B in L1 and L2 |
| DRAM | Avg latency=200 cycles |
| Coherence Protocol | MOESI Directory-based |

We characterized the multithreaded benchmarks from the PARSEC suite [5], [6]. *We pinned the threads to the cores to avoid variability in performance*. Table IV lists 11 programs used in our study. We chose `sim-medium` [5] input sets for our evaluation. Our work shows the results for the ROI. *We use improvement in miss-rate and the execution time as the performance measures.* We do not consider instructions per cycle (IPC) as a performance metric because of the variations due to synchronization primitives such as locks and barriers that are used in multithreaded applications.

Table IV: Characteristics of PARSEC Benchmarks

| Program | Parallelism Model [5] | Sharing Patterns [7] |
|---|---|---|
| `ferret` | Pipelined | Migratory |
| `fluidanimate` | Data | Producer/ Consumer |
| `dedup` | Pipelined | Migratory |
| `canneal` | Unstructured | Read Only |
| `x264` | Pipelined | Read Only |
| `swaptions` | Data | Read Only |
| `streamcluster` | Data | Read Only |
| `blackscholes` | Data | Read Only |
| `bodytrack` | Data | Read Only |
| `vips` | Data | Read Only |
| `freqmine` | Data | Read Only |

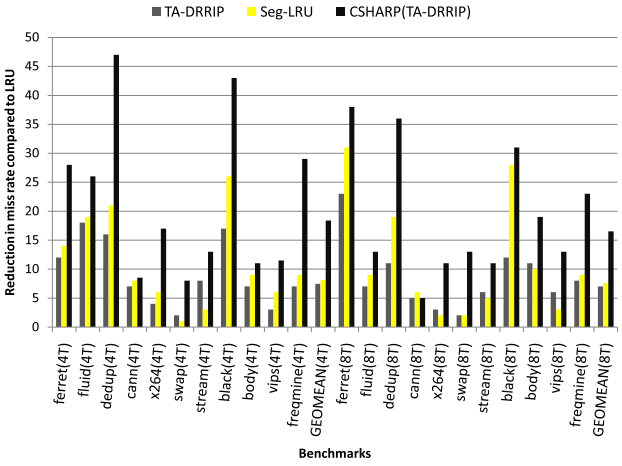


Figure 5: Improvement in Miss Rate over LRU for 4 and 8 Cores (with 4 and 8 MB LLC respectively)

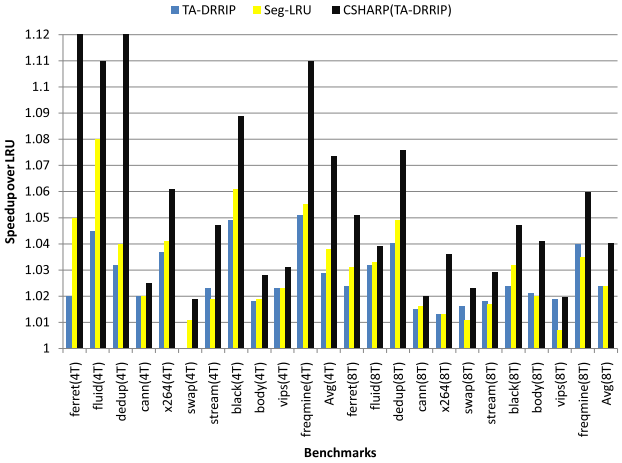


Figure 6: Speedup over LRU for 4 and 8 Cores (with 4 and 8 MB LLC respectively)

## B. Results

We compare CSHARP(TA-DRRIP) with TA-DRRIP, Segmented-LRU [2] and baseline LRU policy. Segmented LRU is the winner of the first JILP Cache Replacement Championship Workshop held in conjunction with ISCA 2010. For the first time in literature, we present the comparisons between TA-DRRIP and segmented-LRU (seg-LRU) policies for multithreaded applications. Using LRU as the baseline policy, we present %reduction in miss rate at LLC in Figure 5. We measured speedup of the three policies compared to LRU and present the results in Figure 6. The terms 4T and 8T are used for 4-core and 8-core simulated systems with one thread pinned per core. GEOMEAN(4T) and GEOMEAN(8T) are the geometric mean across all the 11 benchmarks. The results are summarized in Table V.

For 4T, TA-DRRIP reduces miss rate by an average of 7.43% across the benchmark suite. Segmented-LRU is slightly ahead with a 8.12% average reduction. CSHARP(TA-DRRIP) emerges as a clear winner across all the benchmarks and reduces the miss rate by as much as 18.38% on an average, an extra 10% reduction compared to other methods. Largest reduction is seen in `dedup` (47%). For 8T, TA-DRRIP reduces miss rate by an average of 7.01% and seg-LRU provides a 7.58% reduction. CSHARP(TA-DRRIP) still leads with better reduction in 9 of the 11 benchmarks and shows an average reduction of 17.07%, maintaining more than 8% reduction in miss-rate over TA-DRRIP and seg-LRU. The largest reduction in miss rate for 8T is for `ferret` (38%) and is provided by CSHARP(TA-DRRIP).

In terms of execution time, the average speedup provided by TA-DRRIP, seg-LRU and CSHARP(TA-DRRIP), for 4T, are 2.8%, 3.8% and 7.3% respectively. For 8T, the average speedup provided are 2.2%, 2.3% and 4.01% respectively. CSHARP(TA-DRRIP) provides more than 10% speedup for `ferret`, `fluidanimate` and `dedup` for 4T. In the 8T

case also, ferret and dedup see more than 5% speedup because of CSHARP(TA-DRRIP).

Table V: Comparison of Replacement Policies for LLC with LRU as the Baseline

| Policy | % Reduction in Miss-rate for 4T(8T) | %Reduction in Execution Time for 4T(8T) |
|---|---|---|
| **TA-DRRIP** | 7.43(7.11) | 2.8(2.3) |
| **seg-LRU** | 8.12(7.58) | 3.8(2.4) |
| **CSHARP(TA-DRRIP)** | 18.38(16.49) | 7.3(4.0) |

### C. Analysis of Results

We analyze the results of benchmarks by grouping them on the basis of parallel programming model and the sharing patterns as listed in Table IV.

The largest reduction in miss rate seen in dedup and ferret is because of their use of the pipelined parallel programming model. A write by a thread at stage $i$ of the data pipeline will be consumed by the thread at stage $i+1$. The data is thus migratory. There is a tight coupling between threads at different pipeline stages. The data from one stage to the other will be in cache lines that are mostly in M/O state. These lines are retained by CSHARP(TA-DRRIP) because of prioritization of coherence states and sharing. x264, contrary to the categorization in [5], is only partially pipelined. Also, as opposed to migratory sharing seen in other pipelined applications, it is read-only shared. For these reasons, miss rate reduction is not as phenomenal as in the other pipelined applications. fluidanimate is the only benchmark with producer-consumer kind of sharing which sees a small reduction in miss rate because the non-dirty lines are mostly reused as compared to dirty lines.

swaptions, bodytrack, streamcluster, and canneal have read-only sharing pattern in which most of the cache lines at the LLC are in the O or S state. Since CSHARP breaks ties among the four coherence states and only two of them are predominant in the cache lines in these benchmarks, the miss rate reduction is lesser than what we observed in the previous set of benchmarks. blackscholes, in spite of having read only sharing pattern, is almost comparable to benchmarks with migratory pattern. This is because the application is not memory intensive and only a few thousand accesses are made to the LLC.

### D. CSHARP(LRU)

To demonstrate that CSHARP is a framework that can be applied to any cache replacement policy, we CSHARPed the LRU policy which we call CSHARP(LRU).

LRU policy uses the timestamp registers as re-use registers. For a M bit re-use register, the cache lines can have values ranging from 0 to $2^M - 1$. Cache line which has re-use register value of 0 is a MRU line. The line with reuse-register value of $2^M - 1$ is the LRU line. So for a 16-way cache, MRU corresponds to way=0 and LRU corrsponds to way=15.

We discuss the 3 sub-policies in CSHARP(LRU).

**Eviction policy:** We set the eviction window size to 4 (the tradition LRU policy uses eviction window size $w = 1$). CSHARP, as before, evicts the S lines, before Pr lines with E, before Pr lines with M, before Sr lines and it does so by searching lines within the window from right to left(way=15 to way=12).

**Insertion policy:** mSr lines are inserted in MRU position and mPr lines are inserted in $LRU/2$ position by setting the reuse-register to these values.

**Promotion policy:** On a hit to a Pr line, if the thread that is making request is different from the owner, CSHARP promotes the line to MRU position. For rest of the Pr lines, if the lines are in between $LRU/2 + 1$ and LRU positions, CSHARP promotes them to $LRU/2$ position. If the Pr lines are in between $LRU/2$ and MRU positions, CSHARP promotes them to MRU position. On a hit to a Sr line, CSHARP promotes it to MRU position. We performed experiments on the same set of benchmarks for 4-cores and 8-cores. On an average, CSHARP(LRU) improves the performance by 4.12% and 2.63% for 4T and 8T as compared to LRU. CSHARP(LRU) outperform both TA-DRRIP and seg-LRU in terms of performance. We strongly believe that other cache replacement policies can also benefit from sharing and coherence based decisions that CSHARP provides.

### E. Other Effects of CSHARP

Besides reduction in miss rate and execution time, we also observed other effects of CSHARP that are of interest.

**Sensitivity to LLC Cache Size:** At LLC, increase in cache size is expected to improve the performance because of reduced capacity misses and increased sharing of data. We increased the cache size/core at the LLC from 1MB/core to 2MB/core. CSHARP(TA-DRRIP) reduces miss rate by an average of 16.1% for 4T and 9.2% for 8T compared to LRU. TA-DRRIP reduces miss rate by an average of 5.81% for 4T and 2.88% for 8T compared to LRU. In terms of execution time, CSHARP(TA-DRRIP) improves the performance by 5.12% and 3.27% compared to LRU for 4T and 8T respectively. TA-DRRIP achieved 3.88% and 1.55% performance improvement compared to LRU for 4T and 8T respectively. For both CSHARP(TA-DRRIP) and TA-DRRIP, we observe diminishing returns when we move to 8 cores with larger caches.

**Hardware Overhead:** For all the policies, the hardware overhead incurred should include the reuse-registers and any other supporting hardware. For CSHARP(TA-DRRIP), the Bloom filters constitute the overhead with respect to TA-DRRIP. We calculate all the overhead in terms of number of bits. We present the comparisons in Table VI.

The hardware overhead for a 4MB(8 MB) cache is 4% of

Table VI: Comparison of Hardware Overheads for 4- and 8-cores with 4MB and 8MB LLC Respectively

| Policy | Bits per cache line($P$) | Bits per set (S=Associativity * $P$ + extra bits per set, if any) | Additional hardware(A) | Total(T=#of sets*S + A) |
|---|---|---|---|---|
| 4core (Associativity=16, # of sets=4k) | | | | |
| LRU | 4 | 64 | - | 262.14 kbits |
| seg-LRU | 5 | 80 + 22=102 | ATD=93.5kbits, others=51 bits | 501.50 kbits |
| TA-DRRIP | 2 | 32 | PSELs=64 bits | 131.13 kbits |
| CSHARPed TA-DRRIP | 3 | 48 | PSELs=64 bits, BFs= 6kbits | 202.00 kbits |
| 8core (Associativity=16, # of sets=8k) | | | | |
| LRU | 4 | 64 | - | 524.2 kbits |
| seg-LRU | 5 | 80 + 22=102 | ATD=191.4kbits, others=51 bits | 1027.123 kbits |
| TA-DRRIP | 2 | 32 | PSELs=128 bits | 262.2 kbits |
| CSHARPed TA-DRRIP | 3 | 48 | PSELs=128 bits, BFs= 10.2kbits | 405.2 kbits |

TA-DRRIP's hardware requirements. Thus, CSHARP uses almost the same area as the TA-DRRIP policy but provides significant reduction in miss rate and speedup.

### F. Limitations of CSHARP

With increasing number of cores and emerging truly parallel applications, the level of sharing at the LLC is expected to be high. Thus, our categorization of lines as `Sr` and `Pr` may be insufficient. Instead of the "one" vs "many" approach for categorizing cache lines based on sharing, we have to consider thresholds and categorize lines as "heavily shared" and "lightly shared". In the future, we plan to extend CSHARP to many-core scenario.

To CSHARP a new cache replacement policy, two aspects have to be considered. The assignment of values to the reuse-registers during eviction, insertion, and promotion policies have to be experimented and analyzed. Another aspect to consider is the eviction window size. CSHARP is general enough to be adapted for any replacement policy but the implementation requires these studies.

## VII. Related Work

Chen et al. [11] proposed a LLC replacement technique which dynamically partitions a set for private and shared lines. They observed differences in locality of shared and private lines. The sharing access patterns and working set sizes were also found to be different for these lines. Using these observations, they proposed a new insertion policy that is streaming-aware and a replacement policy that is sharing aware. They show a 8.7% reduction in miss rate compared to LRU policy. CSHARP on TA-DRRIP shows 18% and 16% reduction in miss rate for 4T and 8T as compared to LRU.

## VIII. Conclusion

In this paper, we proposed a new framework called CSHARP that can be applied to any LLC replacement policy to make them sharing and coherency aware. We apply CSHARP on TA-DRRIP and LRU. In both the cases, CSHARPing reduce the miss rate at the LLC significantly. With a negligible increase in cache area, our new framework shows very good promise.


## Acknowledgment

We would like to thank krishna kumar rangan for insightful discussions on this work. We gratefully acknowledge the anonymous reviewers for their valuable reviews and suggestions.


## References


[1] A. Jaleel et al., *"High Performance Cache Replacement using Re-reference Interval Prediction (RRIP)"*, in ISCA 2010, pp.60-71.

[2] H. Gao et al., *"A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing"*, in 1st JILP Workshop on Computer Architecture Competitions, 2010.

[3] A. Jaleel et al., *"Adaptive Insertion Policies for Managing Shared Caches"*, in PACT 2008.

[4] N. Binkert et al. *"The gem5 simulator"*, SIGARCH Comput. Archit. News, Aug 2011.

[5] C. Bienia et al., *"The PARSEC Benchmark Suite: Characterization and Architectural Implications"*, in PACT 2008, pp. 72-81.

[6] M. Gebhart et al., *"Running PARSEC 2.1 on M5"*, The University of Texas at Austin, Department of Computer Science, Technical Report #TR-09-32, October 2009.

[7] N. Barrow-Williams et al., *"A Communication Characterization of SPLASH-2 and PARSEC"*, in IISWC 2009, pp 86-97.

[8] M. K. Martin et al., *"Token Coherence: Decoupling Performance and Correctness"*, in ISCA 2003.

[9] C.J.Wu et al., *"Adaptive Timekeeping Replacement: Fine Grained Capacity Management for Shared CMP Caches"*, in ACM Trans. Archi. Code. Opti., Volume 8, February 2011.

[10] L. A. Belady,*"A study of replacement algorithms for virtual storage computers"*, in IBM Systems Journal 1966, pp. 78-101.

[11] Y. Chen, et al., *"Efficient Shared Cache Management through Sharing-Aware Replacement and Streaming-Aware Insertion Policy"*, in IPDPS 2009, pp. 1-8.