Project Report

Cache Replacement Algorithms for Shared Memory

Multicore Processors & Architecture
Spring Semester 2023
New York University

Keyur Panchal
kp3159@nyu.edu

Vashu Raghav
vr2326@nyu.edu

1. Abstract

Cache replacement is a very highly researched problem, in both single- and multi- programming environments. It directly affects the performance of programs, either speeding them up considerably or slowing them down to a halt. Here, we propose an adaptive algorithm that uses Bimodal Insertion and Dynamic Eviction, (BI-DE), that outperforms the algorithms we tested against in throughput, weighted speedup and average latency. We also extend this to be application aware (AA-BI-DE), which maximises the fairness in the presence of many cores.

2. Introduction

Cache eviction is a problem that has plagued researchers for decades. While the optimal solution for a single core, private cache is known, it is difficult to come up with a solution that performs optimally in an online system. While certain cache replacement policies come close to achieving this under certain conditions, the problem of selecting a cache replacement policy has only become more complex with the advent of multi-core processors and shared caches.

In this project, we first provide a general background about the cache replacement problem, with an emphasis on the problems faced in a shared memory system. Next, we explain several cache replacement algorithms and how they build on each other, in both a single and multi core environment. Building on these ideas, we then propose our own cache replacement policy, and extend it to be used in a multiprogramming enviroment. Finally, we analyze the results of the experiments on our new policy, and showcase it against existing methods.

3. Literature Review

The Cache Replacement Problem

Caches are an essential part of modern computer architecture. They store frequently accessed data in order to reduce the latency in retrieving a data item from the main memory. However, caches tend to be very small as compared to the size of memory, in the order of a few kilobytes, meaning they become full fairly quickly. The problem of which block to evict when a full cache requires a new block of data is the cache replacement problem.

It is absolutely critical since a poor cache replacement policy can tank performance, leading to a lot of cache misses and massive latency arising from multiple main memory accesses. Many Cache Replacement Policies have been proposed, in order to optimize performance.

Considerations when Designing a Cache Replacement Policy

There are three major parts to a Cache Replacement Policy

1. Insertion Policy: This defines what to do with a new block that is entering a cache set. It may define at what position a block should be placed — for example, the “Least Recently Used” eviction policy says that an incoming block should be placed in the “Most Recently Used” (MRU) position.
2. Victim Selection Policy: This defines which block to select as a ‘victim’ i.e. how to decide which block should be evicted next during a cache miss. Using the LRU example again, the block in the “Least Recently Used” position should be evicted.
3. Promotion Policy: This defines how to ‘refresh’ or ‘promote’ a block when a cache hit occurs. For example, on a cache hit, a block is promoted to the MRU position under LRU Replacement Policy.

What Makes a Cache Replacement Policy “Good”?

Before we talk about different policies that have been proposed, let us first see what factors to look for in a “good” cache replacement policy.

A good cache replacement policy aims to do the following:

1. Maximise hit rate / minimize cache misses: The primary goal of a cache replacement policy is to keep frequently accessed data in the cache, thus reducing the need to access the main memory. Thus, we want a high hit rate i.e., very few accesses should be directed to the main memory.

2. Adapt to varying workloads: A good policy should perform well under different workloads and adapt to the changing access patterns efficiently. Broadly, we can divide applications into “cache-friendly” and “cache-averse” categories, based on their data access patterns. Cache-friendly applications tend to access only a few blocks with higher frequency, while cache-averse applications access many different blocks only a few times, such as a streaming or thrashing access pattern.
3. Have a low overhead: An overly complex policy may require extensive resources, leading to a higher hit rate, but lower overall performance, due to the performance cost of enacting the policy.
4. Be scalable: It should scale well with increasing cache size, multiple cores, and shared caches, without compromising performance.

Private vs. Shared Caches

With the introduction of multi-core processors and multiple cache hierarchies, the cache replacement problem gained new dimensions. There are multiple levels of caches in a system (usually L1, L2, and last-level cache (LLC)) with increasing sizes and access latencies. Usually, L1 and L2 and private caches, i.e., each core has its own cache, while the LLC is shared among all cores. Traditional cache replacement policies are pretty viable in private caches.

However, in shared caches, these policies often faced contention and suboptimal performance. This is due to multiple cores ‘polluting’ the cache and interfering with the expected operation of another core. Other factors to consider are ‘Fairness’ — how much cache space is each core allocated — and ‘Throughput’ — how many accesses is each core able to do in the presence of other cores. Thus, these are also considered when designing a cache replacement policy on a multi-core system.

Policies for Private Cache

We now talk about some traditional cache replacement policies, that focus on a single-core or private cache.

Belady’s Algorithm:

Let us first talk about the optimal cache replacement policy. It is proven that, given the order of cache accesses, the optimal eviction policy is the one where the block that is needed ‘furthest in the future’ is removed. This minimizes the number of cache misses.

This, of course, is impractical: in a real system, it is quite impossible to know the order of cache accesses beforehand. However, it does give us a good theoretical baseline for other policies.

Least Recently Used (LRU):

LRU Replacement Policy is pretty straightforward: any block that is inserted or accessed is placed at the most recently used position. (MRU). Evictions are done from the Least Recently Used position. The idea behind this is that blocks that have been used recently are likely to be used again in the future. If they haven't been accessed in some time, they may not be needed in the near future, and can thus be evicted.

Bimodal Insertion Policy (BIP):

It was observed that often when a cache block is not immediately re-used in the LRU replacement scheme, it ends up *never* being used. For example, in streaming workloads. This led to the creation of the Bimodal Insertion Policy, where sometimes, a new block is placed in the *LRU* position rather than the *MRU* position, meaning that it is immediately evicted when another block comes in. This gives better performance in many workloads which are considered non-LRU friendly, such as sequential accesses.

Dynamic Insertion Policy (DIP):

The limitation of BIP is clear: it performs worse than LRU in cases where the workload is LRU-friendly. Thus, we have DIP, which aims to address the limitations of both LRU and BIP by dynamically determining the best insertion policy based on the workload. DIP monitors the cache hit rate of the two separate policies and then selects the one that provides a higher hit rate. This allows it to adapt to different workloads and achieve better overall performance.

This is done by using **Set Duelling Monitors**.

Set Duelling:

Set Duelling is a technique often used in adaptive policies to monitor the performance of LRU and BIP on different sets of the cache, which is a mechanism to implement *Dynamic Set Sampling(DSS)*.[ref] Out of all the sets in a cache, some are designated as *dedicated* or *leader* sets. These sets always follow either LRU or BIP, and track the hit rate under the respective policy.

The remaining sets in the cache are called "follower sets." The policy they follow is determined based on which policy is performing better at the moment, under the current workload.

This works because it has been shown that cache behaviour of a workload can usually be surmised confidently using DSS by sampling just a few sets in the cache. It has been shown that 64 dedicated sets are sufficient to make this selection correctly with a high probability.

Not Recently Used (NRU):

NRU is similar to LRU in that it also considers the approximate 'age' of a cache block. Each block is associated with a flag called the "referenced" flag, which is set to 1 when the block is accessed. In fixed intervals, all the referenced flags are reset to 0. When a replacement is

needed, NRU simply selects a block with a referenced flag value of 0, indicating that it was not recently used. This is a simple and efficient approximation of the LRU policy with lower overhead.

The main issue with this policy is that it only maintains a single bit to look at how old a block is — there is no way to break ties between blocks with flag 0.

Re-Reference Interval Prediction (RRIP):

RRIP is a more advanced cache replacement policy that aims to predict the future reuse of a cache block more accurately based on its past behavior. The main idea behind RRIP is to associate each cache block with a "Re-Reference Prediction Value" (RRPV). This is usually a 3-bit value.

When a block is accessed, its RRPV value is set to 0. When a replacement is required, the block with the highest RRPV value is evicted. The RRPVs of other blocks decay as more cache accesses are done. This approach allows RRIP to adapt to different workloads and focus on retaining cache blocks that are likely to be re-referenced soon, leading to better overall performance.

Depending on the insertion policy, RRIP may be either static or bimodal. In the static case, a block is inserted with the maximum RRPV value., while in the bimodal case, a block is sometimes inserted with the maximum RRPV - 1. The idea behind this is, a block that is inserted will not be a candidate for eviction immediately; rather, it will be given a chance to be re-referenced. This gives better performance for workloads where blocks are not *immediately* re-referenced but are referenced in the near future.

Finally, you can use set dueling to figure out which of the two to use, leading to Dynamic Re-Reference Interval Prediction (DRRIP).

Policies for Shared Cache:

While most of the policies above can be used for the last-level cache (LLC), they do not perform very well, as they do not account for the shared nature of the LLC. There may be accesses and evictions between a single core accesses a shared cache. As described earlier, this leads to problems such as cache pollution, lower fairness, and throughput — all of which the previous policies do not account for. The following try to take this into consideration, either by extending a previous policy or by using a new approach altogether.

Near-Optimal Replacement Policy (NOPT):

It is important to observe that, while Belady's is optimal for a private cache system, it does not readily extend itself to a shared cache system. There are several factors for this, chief among them being the fact that there are multiple *different* cores performing accesses and causing

evictions, and that there *cannot* be a unique optimal policy given that more than one application is running. Based on future knowledge of cache accesses, however, several 'near-optimal' replacement algorithms can created. One of these is 'NOPT-bmiss', a fairly complex algorithm which attempts to minimize the miss rate of a theoretical algorithm.

Thread-Aware Dynamic Insertion Policy (TA-DIP):

Instead of using a global policy vis-à-vis DIP, TA-DIP attempts to make a decision between LRU and BIP for each application. Depending on whether it makes this decision independently by assuming it is the only application running, or by taking into consideration the presence of other applications, TA-DIP can be either 'isolation' or 'feedback' based.

Application Aware RRIP (AA-RRIP):

This work extends the idea of static RRIP for shared cache in two ways.

Firstly, it designs the algorithm to be application aware. This means that, whenever an eviction is necessary due to a cache miss by an application, it firsts checks for a block that was inserted by the same application. Only when such a block is not found, will it try to evict cache blocks of some other application. Eviction of a block is still based on the RRPV of that block.

Secondly, this work also introduces the idea of an 'EbIS': Evicted Block Information Storage. This is an 'on-chip' module, that stores the metadata of recently evicted blocks (it does *not* store the data in that block). When a block is evicted, the metadata of that block is inserted into the EbIS.

When a new block is inserted, the policy checks to see if the block is present in the EbIS i.e. if it was recently evicted. If so, the block is inserted with RRPV=0, rather than the maximum value of RRPV. This fixes cases where some blocks were evicted prematurely and then re-referenced.

The policy of eviction of the metadata from the EbIS itself is also defined, based on the number of blocks each application inserted into the EbIS.

AA-RRIP tries to be more 'fair' in its approach, specifically trying to help cache-friendly applications perform better in the presence of cache-averse applications.

Partition Aware Cache Replacement:

Some cache replacement policies try to 'partition' the cache, by allocating each application some number of blocks in the cache. Of course, if this was done in a fixed, static way, this would effectively ruin the idea of a shared cache. Hence, the partitioning is done dynamically i.e, the size of each application's 'private' cache depends on how many cache misses occur in its private cache.

One policy to specially mention here is the “Partition-Aware Eviction, Thread-Aware Insertion/Promotion” (PAE-TIP) policy. It extends the earlier TADIP-feedback to be used during promotions and insertions but uses partition-aware eviction, i.e., the block to be evicted will be based on if the ‘miss-causing core’ has greater or fewer number of blocks in the cache than were initially allotted to it.

Special Mention: Machine-Learning Based Policies

In today’s AI-obsessed world, machine learning has even been applied to help figure out the cache replacement policy. This is usually done by training a model on some number of cache accesses, so that, in the future, it may perform optimal replacements.

These policies tend to perform *very* well. One example is Glider, a policy that arose by training an LSTM-based model on offline data and then modifying it to become an SVM-based predictor that works well on online data as well. It even outperformed the top-finishers of the 2nd Cache Replacement Championship.

There are two main issues with ML-based policies. Firstly, the overhead of maintaining, training and improving a ML model is massive, as compared to cache access latencies. Secondly, they only perform very well on specific workloads — ones that they are trained on. This makes them very impractical to use in the general case, even when modified to be lower-cost online SVMs.

Proposed Idea

LRU cache replacement performs fairly well in cache-friendly applications, since cache-friendly access patterns cause recently used blocks to be reused in the near-future. When cache-friendy and cache-averse applications run in the same environment, the cache-averse application pollutes the cache, causing the cache-friendly application to behave worse.

On the other hand, RRIP tends to perform fairly well on such applications, since, during insertion, it assumes most blocks will *not* be re-referenced in the near future. It only promotes these blocks once they are re-referenced.

We observe these to be on two ends of the spectrum: while each policy individually tries to mitigate its shortcomings using bimodal and dynamic insertion, we suspect that better performance could be obtained in a shared environment by dynamically selecting which one to use.

Thus, we propose an RRPV+LRU based Bimodal Insertion/Dynamic Eviction Policy (BI-DE), an adaptive policy that selects between dynamic re-reference interval prediction, and bi-modal insertion with LRU. The idea behind this is to figure out which of these two “opposite” policies is best suited for a given mix of workloads, and use it accordingly.

This policy works as follows:

1. During insertion, a block is inserted using bimodal insertion to either the MRU or the LRU position in the set. Then, its RRPV is set to be either *distant* or *long,* another bi-modal decision. Note that both bi-modal decisions are made independently of each other, to capture the oppositeness of each policy.
2. During the promotion of a block, a block is moved to the LRU position, and its RRPV is set to 0.
3. We use Set Duelling Monitors (SDMs) to determine which policy to follow during the eviction of a block. If RRIP is followed, then the block with maximum RRPV is evicted. If LRU is followed, then the block at the LRU position will be evicted.

Each policy is tracked using its own SDM for each core. We thus allocate $2nz$ sets to be dedicated sets, where n is the number of cores in the system and z is the size of each SDM.

The SDMs themselves are generated using random replacement, with each core getting its own sets. Half of these sets are assigned to follow RRIP and the other half BIP.

We set z=64 since this has been shown to be optimal in previous works.

Note that, the only consideration of this being a policy for shared cache is done when determining SDMs independently for each core, that contributes to the global policy. The eviction of a block in each part of the policy itself is application-unaware.

Next, we wanted to account for the idea that, in a shared cache environment, certain blocks may be prematurely evicted. Thus, we extended this above algorithm to use an Evicted Block Information Storage (EbIS) — BI-DE-EbIS. The EbIS stores cache metadata only such as the set and tag of a block that was evicted recently. The idea of this policy is to perform better in the cases where BI-DE prematurely evicts blocks, due to the shared cache.

Thus, BI-DE-EBIS is essentially the same as BI-DE, except that during insertion, we first check if the block was recently evicted (by checking if it is in the EbIS). If so, we insert the block at LRU position with an RRPV value of 0. Any evicted blocks are inserted into the EbIS in FIFO order, with no application information being stored.

Finally, we extend BI-DE to be application aware during RRIP-based eviction — AA-BI-DE. The idea behind this is similar to the idea behind the original AA-RRIP work, that is, to attempt to ensure that cache-averse applications do not pollute the cache of cache-friendly applications.

For insertion, promotion, and eviction under LRU, this algorithm works exactly like BI-DE. However, during eviction under RRIP, this works as follows:

1. First, we check if there are any blocks with maxRRPV.
    1. Among all such blocks, we first try to evict a block of the same application as the block to be inserted, failing which we evict a block with maxRRPV of any application (this application is chosen in a round-robin fashion.)
2. If no blocks with maxRRPV are present, then we first check if the offending application has any such blocks in the cache. If not, all blocks are decayed till a random block with maxRRPV is found.
3. Otherwise, only the blocks of the offending application are decayed, leaving the blocks of other applications untouched.

It also uses an application-aware EbIS, which, during eviction from EbIS, first selects a target application, then removes the first inserted block of that application from the EbIS. Insertions are FIFO.

Experimental Setup

We perform simulations using ChampSim, the simulator used for the second and third Cache Replacement Championships. This is a trace-based simulator, that allows us to simulate branch predictors, prefetchers and replacement policies. For all simulations, we used the default branch predictor with prefetching disabled.

Each core has its own set of L1 cache (split into instruction and data caches) and L2 cache. There is a shared last-level cache. We assume that the last level cache is a 16-way set associative cache, which is quite commonly seen.

The important configurations of the caches across all simulations are given in a table below. Only the number of cpus *n* and the workload (mix of traces) was varied for each replacement policy that we tested.

Parameter	Value
Block size	64 bytes
L1I.sets	64
L1I.ways	8
L1D.sets	64
L1D.ways	12
L2.sets	1024
L2.ways	8
LLC.sets	2048
LLC.ways	16

Since we are using a simulator, it can be run on any system. They produced the same results on both CIMS (crunch5.cims.nyu.edu) and our local system (Macbook Air M1).

We test the three aforementioned replacement policies (BI-DE, BI-DE-EbIS, and AA-BI-DE) against our implementations of LRU, BIP, DRRIP and AA-RRIP on differing workloads. Each

workload is a mix of traces, where each individual trace is a SPEC2006 benchmark. There are a total of 8 traces. Each trace represents an application that may be either cache-friendly or cache-averse. In a mix of workloads, each trace runs on a single core independently.
We test on one-core, two-core, four-core and eight-core systems, on mixes that contain the same number of traces as cores. Each experiment is run with 200,000 warmup instructions and 5M instructions to be executed on each core.

We evaluate the following metrics for each core: weighted speedup (WS), throughput, and harmonic mean fairness (HMF) and average cache miss latency.

WS is calculated as follows: $WS = \Sigma \frac{IPC^{shared}}{IPC^{alone}}$, where IPC-alone is the instructions per cycle of an application when it runs alone while IPC-shared is the IPC of that application in a multiprogramming setup.

Throughput is just the sum of the IPCs of all cores.
HMF is calculated as follows: $n/\Sigma \frac{IPC^{alone}}{IPC^{together}}$ where *n* is the number of processors in the simulation.

Average Latency is calculated as follows: $Avg\ Lat = hit\ rate + miss\ rate * miss\ latency$
This gives us a good idea of cache performance by accounting for both hit rate of the cache as well as the performance overhead. We use the direct results of ChampSim output for the same.

The given below are traces that we used for different experiments.

Mix No.	Benchmark
1.	milc.xz
2.	namd.xz
3.	leslie3d.xz
4.	povray.xz
5.	soplex.xz
6.	hmmer.xz
7.	omnetpp.xz
8.	sphinx3.xz
9.	milc.xz namd.xz
10.	leslie3d.xz povray.xz

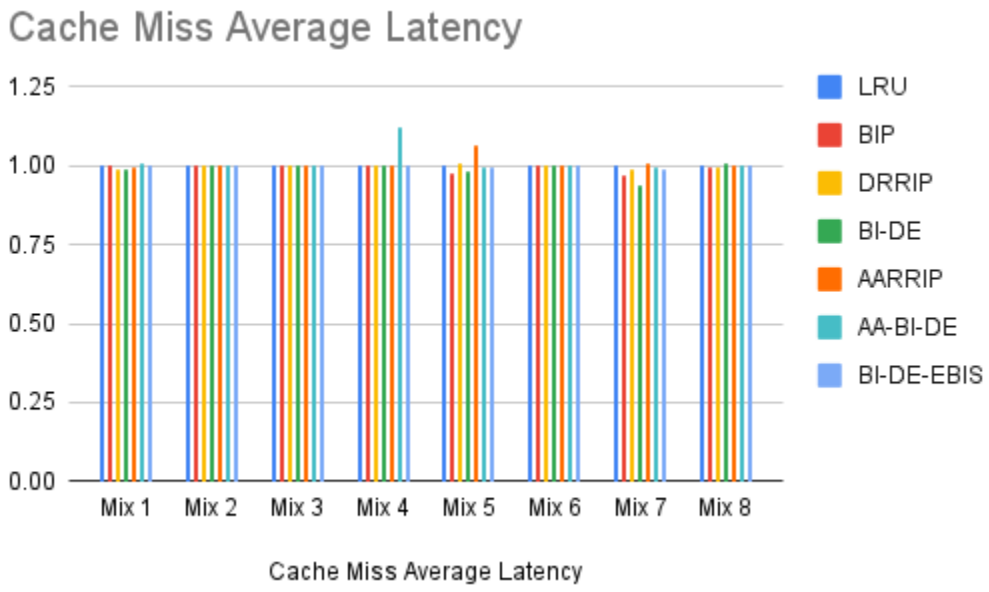
11.	soplex.xz hmmer.xz
12.	omnetpp.xz sphinx3.xz
13.	milc.xz namd.xz leslie3d.xz povray.xz
14.	soplex.xz hmmer.xz omnetpp.xz sphinx3.xz
15.	milc.xz namd.xz leslie3d.xz povray.xz soplex.xz hmmer.xz omnetpp.xz sphinx3.xz

Results and Analysis

We share the results of the experiments as follows:

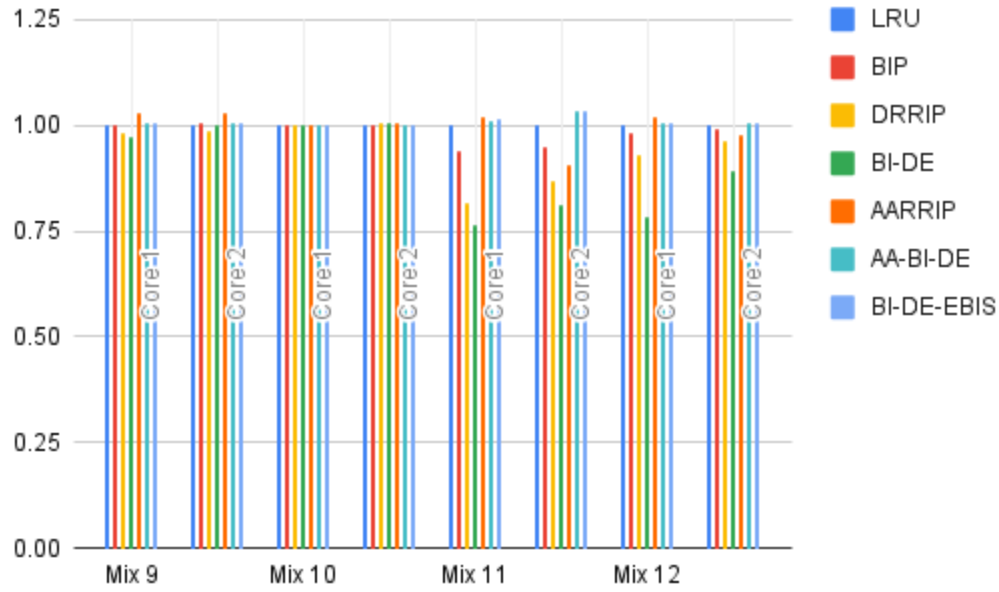
Core 1 (MIx 1 to Mix 8)

The given below is cache miss latency relative to the LRU cache.

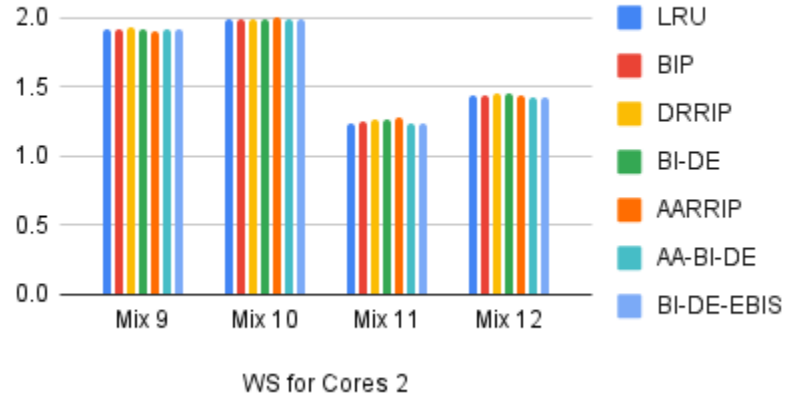


Core 2 (Mix 9 to Mix 12)

The given below is cache miss latency relative to the LRU cache.

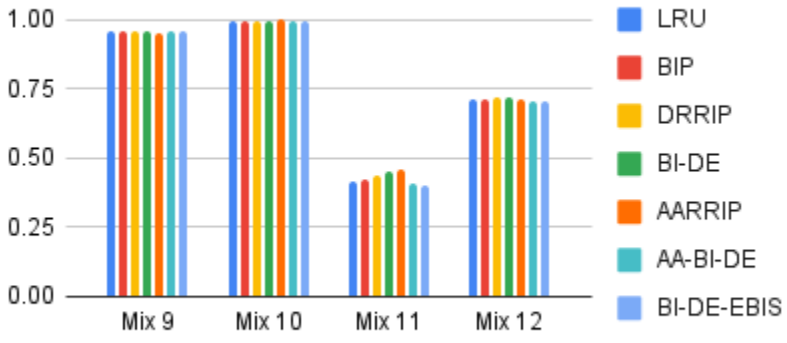


Weighted Speedup

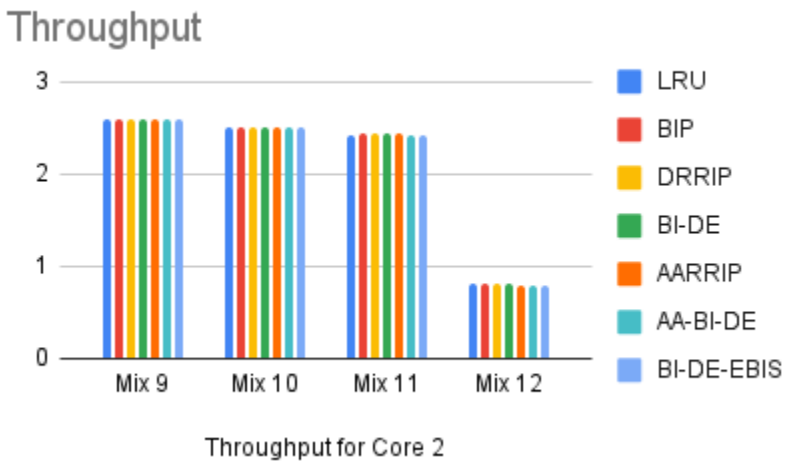


WS for Cores 2

Harmonic Mean Fairness

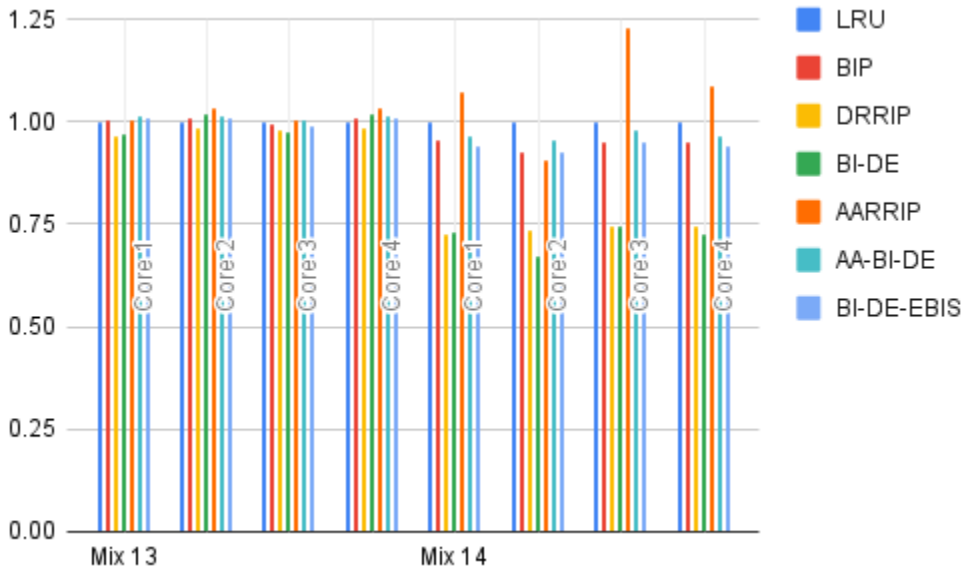


HMF for Cores 2

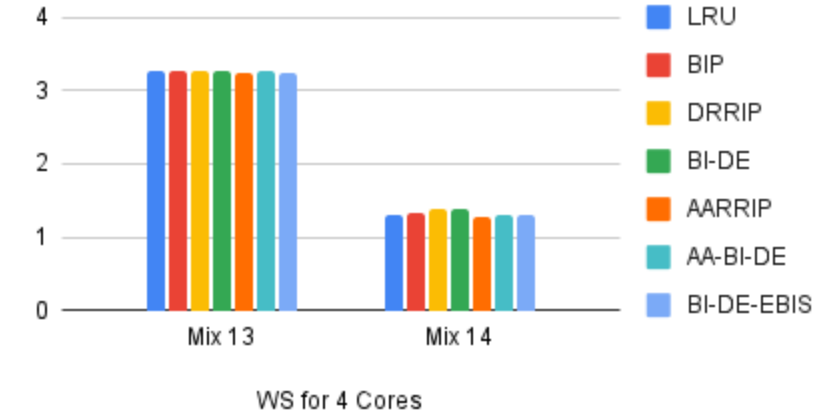


Core 4 (Mix 13 to Mix 14)

The given below is cache miss latency relative to the LRU cache.

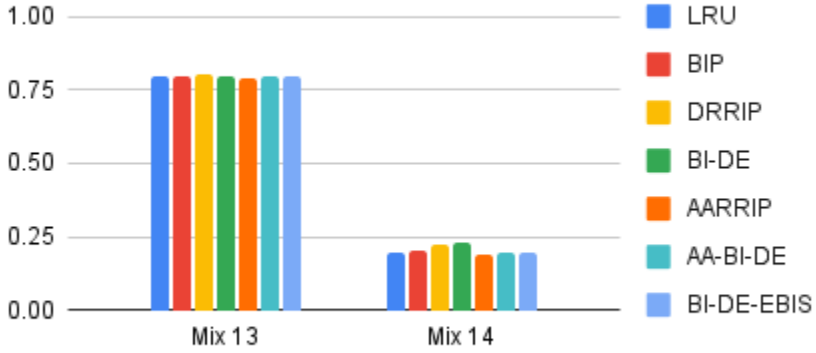


Weighted Speedup



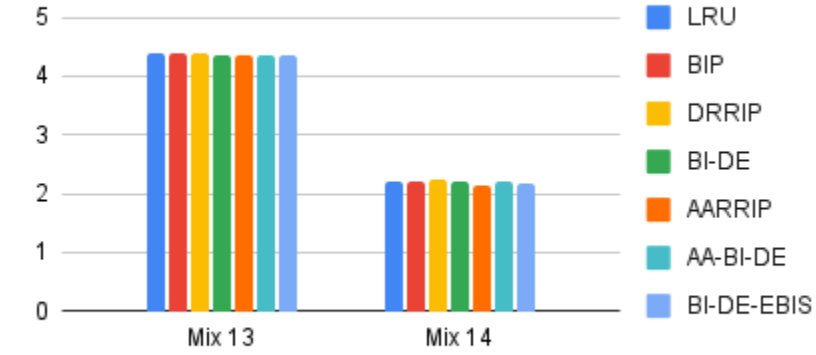
WS for 4 Cores

Harmonic Mean Fairness



HMF for 4 cores

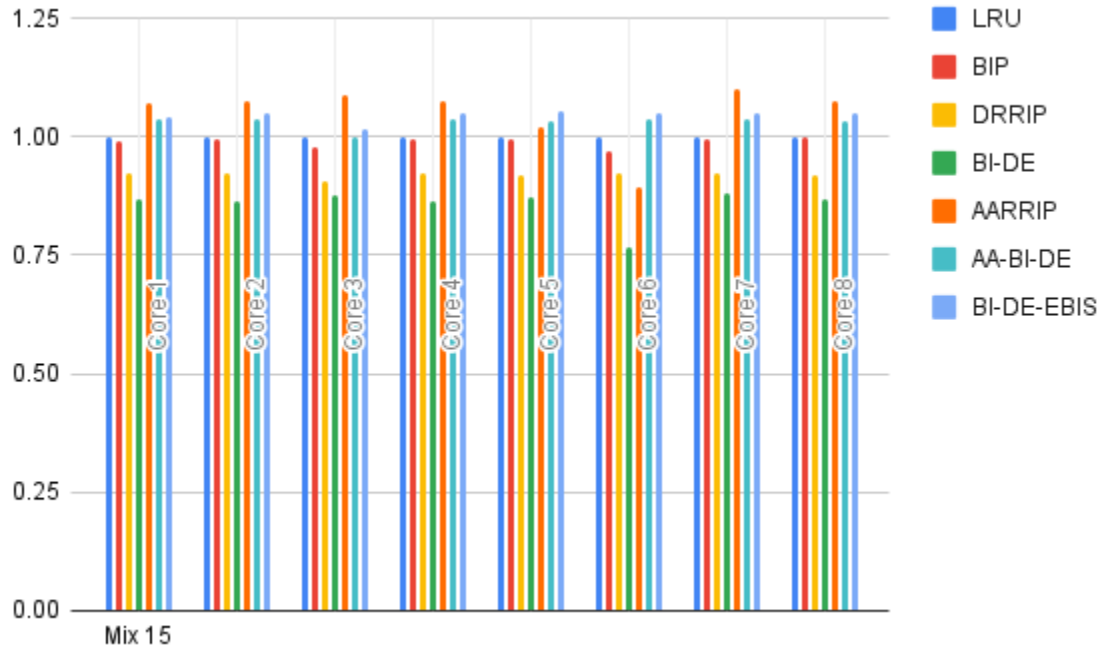
Throughput

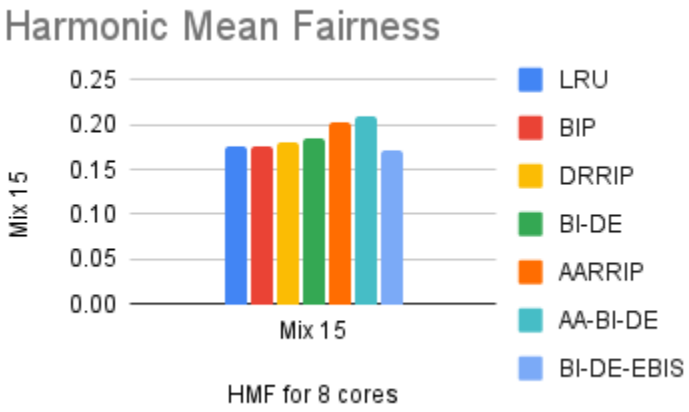


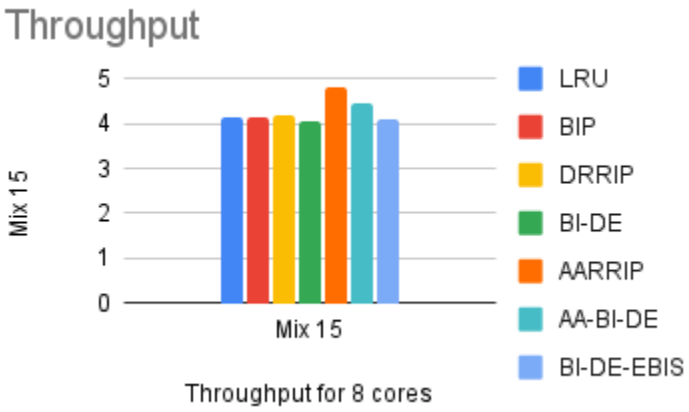
Throughput for Core 4

Core 8 (Mix 15)

The given below is cache miss latency relative to the LRU cache.







From the analysis from the all the experiments above with different core settings, we analyze that

1. Our proposed BI-DE replacement policy outperforms the aforementioned policies in the average latency metric under all workloads. The mix of traces tried to keep a healthy mixture of both cache-friendly and cache-averse applications, meaning that the decision to switch between DDRRIP and BIP globally did make sense, even in a shared workload environment.
2. Although adding application awareness to BI-DE did cause a drop in throughput, it provided a better weighted speedup. It beat the regular AARRIP in terms of fairness as well, in the eight-core case. Curiously, with fewer cores, it did not provide better awareness. We suspec this is due to the presence of the EbIS, which causes recently evicted blocks of an application to be re-inserted, leading to lower IPC.

3. As expected, both application-aware policies performed really well in terms of fairness as compared to the other non application-aware policies, especially in the eight-core setups. This is due to them first evicting their own blocks, before evicting blocks of another application.
4. Unexpectedly, adding an EbIS to BI-DE did not help. All metrics fell considerably under the presence of an EbIS. We suspect that this is the case since in shared workloads with a single, application un-aware EbIS, too many blocks are being evicted. Thus, it may be causing thrashing between prematurely evicted blocks. We suspect that this could be mitigated by using a larger EbIS, which may be impractical since it is on-chip.

Conclusions:

1. Dynamically switching between a policy that predicts blocks to be reused immediately (LRU) v/s a policy that predicts blocks to be reused in the distant future (RRIP) helps mitigate the shortcomings of both policies.
2. Depending on the use-case, application awareness must be accounted for in order to maximise fairness in a multi-programming environment with shared cache with higher number of processors.
3. Doing this, however, does lead to a drop in throughput and weighted speedup.
4. Keeping track of prematurely evicted blocks does not help as much as expected, mainly because, in the case of shared workloads, too many blocks are evicted.

References:

1. *Díaz, J., Ibáñez, P., Monreal, T. et al. Near-optimal replacement policies for shared caches in multicore processors. J Supercomput* ***77****, 11756–11785 (2021). https://doi.org/10.1007/s11227-021-03736-1*
2. *Y. Tang, et al., "Cache Management with Partitioning-Aware Eviction and Thread-Aware Insertion/Promotion Policy," in International Symposium on Parallel and Distributed Processing with Applications, Taipei, Taiwan, 2010 pp. 374-381.doi: 10.1109/ISPA.2010.45*
3. *Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). SIGARCH Comput. Archit. News 38, 3 (June 2010), 60–71. https://doi.org/10.1145/1816038.1815971*
4. *Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3352460.3358319*
5. *F. Juan and L. Chengyan, "An Improved Multi-core Shared Cache Replacement Algorithm," 2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science, Guilin, China, 2012, pp. 13-17, doi:10.1109/DCABES.2012.39.*
6. *A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely and J. Emer, "Adaptive insertion policies for managing shared caches," 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT), Toronto, ON, Canada, 2008, pp. 208-219.*
7. *Fang, Juan & Kong, Han & Yang, Huijing & Xu, Yixiang & Cai, Min. (2022). A Heterogeneity-Aware Replacement Policy for the Partitioned Cache on Asymmetric Multi-Core Architectures. Micromachines. 13. 2014. 10.3390/mi13112014.*
8. *Ghosh, Soma Niloy, Vineet Sahula, and Lava Bhargava. "Reuse-Aware Cache Partitioning Framework for Data-Sharing Multicore Systems." 2021 IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS). IEEE, 2021.*
9. *Gober, N., Chacon, G., Wang, L., Gratz, P. V., Jimenez, D. A., Teran, E., Pugsley, S., & Kim, J. (2022). The Championship Simulator: Architectural Simulation for Education and Competition. https://doi.org/10.48550/arXiv.2210.14324*
10. *K. K. Dutta, P. N. Tanksale and S. Das, "A Fairness Conscious Cache Replacement Policy for Last Level Cache," 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2021, pp. 695-700, doi: 10.23919/DATE51398.2021.9474096.*
11. *L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," in IBM Systems Journal, vol. 5, no. 2, pp. 78-101, 1966, doi: 10.1147/sj.52.0078.*
12. *M. K. Qureshi, D. N. Lynch, O. Mutlu and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," 33rd International Symposium on Computer Architecture (ISCA'06), Boston, MA, USA, 2006, pp. 167-178, doi: 10.1109/ISCA.2006.5.*