Drishti: Do Not Forget Slicing While Designing Last-Level Cache Replacement Policies for Many-Core Systems

Sweta
Indian Institute of Technology Bombay
Mumbai, India
sweta@cse.iitb.ac.in

Prerna Priyadarshini
Indian Institute of Technology Bombay
Mumbai, India
prerna@cse.iitb.ac.in

Biswabandan Panda
Indian Institute of Technology Bombay
Mumbai, India
biswa@cse.iitb.ac.in

Abstract

High-performance Last-level Cache (LLC) replacement policies mitigate off-chip memory access latency by intelligently determining which cache lines to retain in the LLC. State-of-the-art replacement policies significantly outperform policies like LRU. However, the effectiveness of these policies is not evaluated on many-core systems with sliced LLCs, which is common in commercial many-core systems. Recent state-of-the-art LLC replacement policies use two seminal ideas: (i) a sampled cache and (ii) a reuse predictor. In a monolithic LLC, there is a single sampled cache and a single reuse predictor. However, these structures must be created per slice with the sliced LLC. We study the interaction between sliced LLC and state-of-the-art replacement policies, identifying a few unexplored interactions. A per-slice reuse predictor makes myopic decisions based on the accesses made to a particular slice, unaware of the global reuse behavior. A trivial solution to this problem is to design a centralized reuse predictor shared by all the LLC slices. However, this will significantly increase interconnect traffic, requiring more bandwidth to access the centralized reuse predictor. Next, we observe that with a sliced LLC, the LLC sets used for the sampled cache do not receive sufficient LLC misses. As these LLC sets drive the decisions of LLC replacement policies, some of the decisions become suboptimal.

We propose Drishti, which is designed to improve the effectiveness of LLC replacement policies further. We make a case for two enhancements: (i) a per-core and yet global reuse predictor with a local (per-slice) sampled cache, and argue that there is no need for a global sampled cache, and (ii) a per-slice dynamic sampled cache to improve the utility of LLC sets used for the sampled cache. We evaluate two state-of-the-art LLC replacement policies, Hawkeye and Mockingjay, on four, 16, and 32-core systems with eight, 32, and 64MB sliced LLC. On a 32-core system, Drishti enhances the effectiveness of two state-of-the-art replacement policies, Hawkeye and Mockingjay, by improving performance by 5.6% and 13.2%, respectively, compared to the baseline LRU policy. Without Drishti, Hawkeye and Mockingjay improve performance by 3.3% and 6.7%, respectively.

1 Introduction

Last-level cache (LLC) replacement policies are essential in improving overall system performance, as it is a resource shared among multiple cores. High-performance LLC replacement policies proposed in the last two decades [48], [28], [34], [60, 61], [18], [23], [32], [31], [29], [59], [58], [55], [39], [38], [27] and [52] have pushed the performance limits significantly. However, these policies [18, 23, 27–29, 31, 32, 34, 48, 52, 55, 59, 60] are not evaluated for multi-core systems with sliced LLC, which is common in commercial processors [10, 13, 24]. For example, AMD Zen3 [10] has a 32MB L3 cache (eight 4MB slices) that is shared by eight cores. These slices are distributed, leading to non-uniform cache access (NUCA). In this paper, we do not consider machine learning (ML) [55] and reinforcement learning (RL) [38]-based LLC replacement policies, as our goal is not to propose new replacement policies; instead, we focus on identifying unexplored interactions while revisiting some of the fundamental ideas used in LLC replacement policies. We also did not evaluate memory-level parallelism (MLP) and concurrency-aware LLC replacement policies [39], as our goal is to assess the effect of fundamental ideas without being overshadowed by MLP and concurrency awareness.

Our observations. We evaluate state-of-the-art replacement policies on a sliced LLC, and we present the following observations.

Observation I: Myopic predictions. State-of-the-art LLC replacement policies use seminal ideas like set dueling [35] in the form of a sampled cache [18, 27, 52, 60, 61]. The set dueling technique monitors the behavior of a few randomly chosen LLC sets to make predictions for the entire LLC. Most recent policies extend the idea of set dueling into a sampled cache that tracks the reuse behaviors of cache lines belonging to those sampled sets. Also, most of these techniques use a program counter (PC) as the signature to track and predict the reuse of cache lines. With a sliced LLC, each slice uses its own local sampled cache and a local predictor, and different load addresses brought by the same PC get scattered across different slices. This scattering effect causes the predictor of each slice to be trained only on the *myopic* accesses. These accesses are monitored by the local sampled cache of each slice, leading to *suboptimal* decisions, which exacerbates in many-core systems as the likelihood of loads from the same PC being mapped to one slice decreases with increasing core count.

Observation II: Underutilized sampled sets. On a sliced LLC, the LLC sets used for the sampled cache are per slice, and proper LLC hashing [33, 41] ensures uniform distribution of accesses across LLC slices. However, the LLC misses observed at the LLC sets that are part of the sampled cache are non-uniform. There are a few LLC sets that get fewer misses, and few sets get a large fraction of misses [47, 50, 62]. LLC slices with sampled cache having LLC sets with fewer misses do not contribute significantly to the replacement policy decisions. Based on the observations, we find a few unanswered questions regarding LLC replacement policies on many-core systems with sliced LLCs.

MICRO 2025, October 18–22, 2025, Seoul, Korea
2025. ACM ISBN 978-X-XXXX-XXXX-X/XX/XX
https://doi.org/XXXXXXX.XXXXXXX

The pertinent questions. We ask the following fundamental questions: (i) On a sliced LLC, should the reuse predictors used by LLC replacement policies be local to each slice, or should they be global to the entire LLC so that it can mitigate the *myopic* effect? (ii) What are the challenges in designing a global reuse predictor on a many-core system? A global reuse predictor shared across all slices will lead to a bandwidth bottleneck as multiple cores will contend for a single global predictor. The bottleneck will become worse with the increase in core count. (iii) With large sliced LLCs (e.g., 32MB and 64MB with 32 and 64 slices) used in many-core systems, and accesses becoming scattered, is the conventional method of randomly selecting a few LLC sets (e.g., 32 or 64 per slice) for the sampled cache effective? These questions highlight that existing policies may not fully exploit all opportunities for optimization, suggesting scope for further performance improvement.

Our goal is to improve the performance of state-of-the-art LLC replacement policies for a many-core processor with a large LLC that is sliced. To achieve this, we aim to capitalize on unexplored interactions between sliced LLC and LLC replacement policies. We propose Drishti[1], which provides a few fundamental enhancements on top of LLC replacement policies, ensuring the performance delivered by these policies is enhanced in the sliced LLCs shared by multiple cores.

Our approach. We argue for a per-slice sampled cache, local to each LLC slice. However, we use a per-core and yet global reuse predictor, used by all LLC slices. As LLC accesses from a particular core get scattered across multiple LLC slices, we use one reuse predictor per core located closer to the core's LLC slice, which reduces the shared interconnect traffic compared to a centralized global predictor (one global predictor for all slices). Although using a per-core global predictor reduces interconnect traffic compared to a centralized global predictor, there is still additional traffic because each core's predictor can be accessed by any slice. To reduce this traffic, we use a dedicated low-latency interconnect (NOCSTAR) [19] that links all slices to the predictors with a latency of just three cycles. We make a case for a dynamic sampled cache that selects LLC sets for the sampled cache based on LLC misses, rather than selecting them randomly.

Our contributions. To the best of our knowledge, this is the first work that evaluates the state-of-the-art LLC replacement policies on a many-core system with a large sliced LLC. We observe that the effectiveness of these policies can be improved further on the sliced LLC. We motivate the need for fundamental changes in how replacement policies interact with the sliced LLC (Section 3). We propose Drishti, which enhances the effectiveness of state-of-the-art replacement policies through a (i) local per-slice sampled cache and per-core predictor that is global to all the LLC slices and (ii) a dynamic sampled cache. We address the design challenges to make Drishti scalable (Section 4). On a 32-core system with 64MB LLC, when Drishti enhancements are applied to Hawkeye and Mockingjay, it provides performance improvements of 5.6% and 13.2% compared to a baseline with LRU replacement policy. On the other hand, Hawkeye and Mockingjay without Drishti provide 3.3% and 6.7% over LRU for 32-core systems. Drishti's effectiveness comes with storage savings (instead of overhead) of 7.25KB and 2.96KB per core for Hawkeye and Mockingjay, respectively (Section 5).

[1]Drishti is a Sanskrit word meaning focused gaze. We envision Drishti as a focused enhancement of state-of-the-art replacement policies.

2 A quick primer on Hawkeye and Mockingjay

In this Section, we provide an overview of the recent advances in LLC replacement policies. Seminal works, such as the dynamic insertion policy (DIP) [48] and the re-reference interval prediction-based policy (RRIP) [28], pave the way for non-LRU-based policies. The computer architecture community organized two championships on cache replacement policies [1] [5]. Post RRIP, ideas like SHiP [60] make a case for the reuse prediction of cache lines based on coarse-grained signatures like program counter (PC) and memory region. The next big leap in the field of cache replacement policy came in the form of Hawkeye [27], which is also the winner of 2nd cache replacement championship [5].

Hawkeye emulates Belady's policy by looking at the past accesses to a large cache, which is eight times the LLC. To make this approach practical, Hawkeye uses the concept of a sampled cache. The role of the sampled cache is to track the reuse behavior of cache lines based on the program counter (PC). Instead of monitoring all the accesses, the sampled cache only records the accesses seen by the sampled sets (a few LLC sets chosen randomly), and these accesses take part in mimicking Belady's optimal policy.

Hawkeye's reuse predictor is trained based on whether a load results in a cache hit or miss under Belady's optimal policy. When a load is a cache hit, according to Belady's policy, the predictor learns that the PC triggered the load as "cache-friendly." Conversely, if the load is a miss under Belady's policy, the predictor is trained as "cache-averse" for that particular PC. In the future, when a cache fill occurs, the predictor assigns an RRIP (Re-reference Interval Prediction) value based on whether the associated PC is cache-friendly or cache-averse.

Mockingjay extends Hawkeye from a binary classification of cache lines into a multi-class classification problem. It uses Beladay's emulation, as done in Hawkeye, for multi-reuse prediction. Mockingjay predicts a cache line's Estimated Time of Arrival (ETA). However, to make it practical, Mockingjay argues that the relative ordering of Estimated Time Remaining (ETR) is the same as that of ETA. So, Mockingjay maintains an ETA based on the ETR and the current timestamp. These ETRs are used to mimic Belady's optimal policy.

Similar to Hawkeye, Mockingjay also employs a sampled cache and PC-based reuse predictors. The sampled cache monitors the reuse of loads observed by the sampled set and maintains a timestamp for each block. When a sampled cache hit occurs, the block's last timestamp trains the reuse predictor with the observed reuse distance. In contrast, on a sampled cache miss, the PC of the evicted line is trained to indicate that it was not reused, and it is assigned an INFINITE reuse distance. In summary, both replacement policies use the sampled cache, and both policies use predictors that predict the reuse behavior of cache lines using PCs.

Mockingjay with a sliced LLC. In a many-core system with a sliced LLC, each slice has its local sampled cache and predictor. Figure 1 shows the structure of Mockingjay in a 32-core system with 32 slices. Each slice has its per-core predictor and sampled cache. Figure 1 also explains the end-to-end tracking of the load

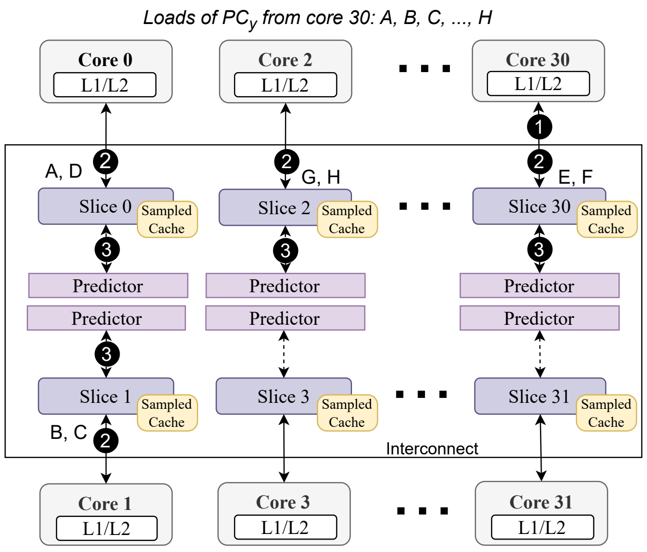


Figure 1: Mockingjay with per-slice per-core reuse predictor on a sliced LLC-based 32-core system. Each slice has a predictor, indexed with a hash of PC and core ID.

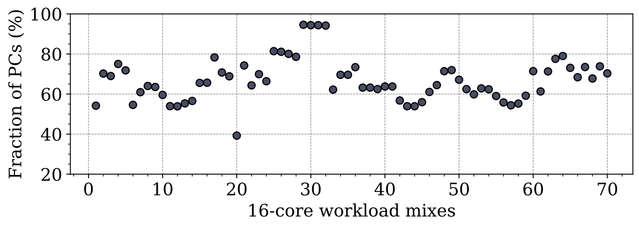


Figure 2: Fraction of PCs per core(excluding those that bring only a single load) mapping demand loads to one LLC slice throughout their execution for 16-core mixes (35 homogenous and 35 heterogenous created from SPEC CPU2017 and GAP). The higher, the better.

by the local sampled cache, learning, and prediction of the local predictors as different steps. As per step ❶, a PC_Y of core 30 brings load sequences like A, B, C,..., H. It can be seen in step ❷ that these loads get scattered into different slices and get recorded by the respective local sampled cache. In step ❸, it can be observed that the local sampled cache of each slice then trains the local predictor of the respective slice based on the loads seen by the slice.

3 Motivation

In this Section, we highlight some key concerns related to the interaction of state-of-the-art LLC replacement policies with a sliced LLC where slices are distributed over an interconnect.

3.1 The myopic behavior

In this section, we provide a comparative view of the reuse predictions from global and myopic views. The global view is the case when the predictor of all slices can see access to every sampled set in LLC. Figure 2 shows the fraction of PCs whose corresponding memory accesses are mapped to only one LLC slice. On average across 35 homogeneous and 35 heterogeneous mixes, load requests brought by 66.2% of PCs (per core) are mapped to one LLC slice. The trend is worrisome for workloads like `xalan`, as around 40% of PCs are mapped to only one LLC slice (workload mix 20 in Figure 2), making the reuse predictor a *myopic* reuse predictor. Please note that the trend illustrated in Figure 2 is unaffected by LLC replacement policies and hardware prefetching techniques.

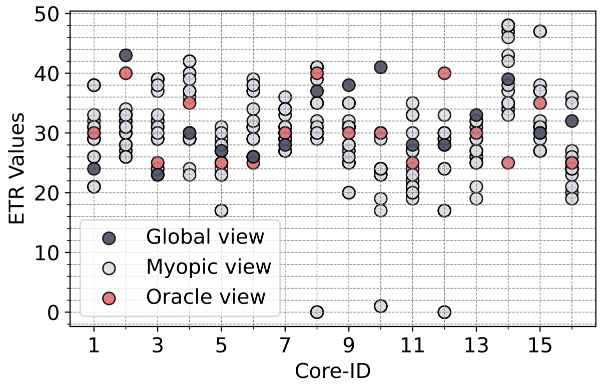


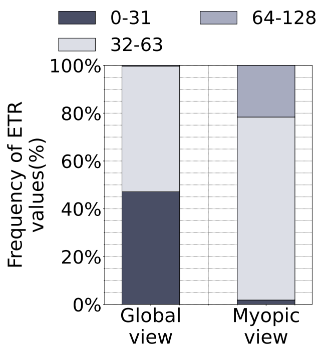
Figure 3: ETR values corresponding to the loads of PC `0x59cdbf` with Mockingjay when the predictor sees global view, myopic view, and the oracle view of access pattern on a 16-core system running a homogeneous mix of `xalan`.

Figure 3 compares the predicted ETR values when the predictor is trained on global accesses versus when trained on myopic accesses, and the oracle ETR (actual reuse distance). To showcase this, we select a PC, `0x59cdbf`, which we refer to as PC_X, for a 16-core workload running `623.xalancbmk_s-202B`. We track the loads issued by PC_X, and as expected, these loads are scattered across different slices. Additionally, we record the prediction values (ETR values of Mockingjay) corresponding to PC_X from the predictor for each core and slice when the predictor observes both myopic and global access patterns. For the myopic predictions, ETR values are shown for each of the 16 slices per core, resulting in 16 dots per core (though overlapping may make it appear fewer than 16). It is clear that myopic prediction values deviate from the corresponding global prediction values and the oracle ETR values, which leads to a drop in prediction accuracy. It is also clear that the global view provides better ETR values as it is closer to the oracle view.

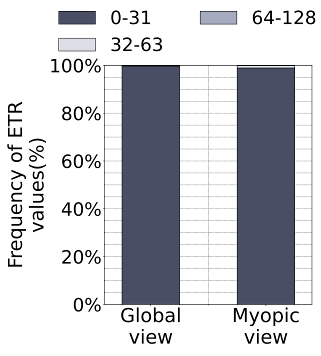
Figure 4 shows how the frequency of predicted reuse values differs between myopic and global views. We use two workloads, `xalan`, with the fewest PCs mapped to the same slice, and `pr`, with the most PCs mapped to the same slice, as case studies to understand reuse predictions based on the scattering effect. The difference in ETR values for Mockingjay in Figure 4a is more significant than in Figure 4b due to the higher number of PCs with loads scattered across different slices. For Hawkeye, the number of LLC lines with predicted RRIP values of zero (cache-friendly) and seven (cache-averse) differs between global and myopic views. Hawkeye's difference in Figure 4c is more significant than in Figure 4d.

3.2 Under-utilized sampled cache

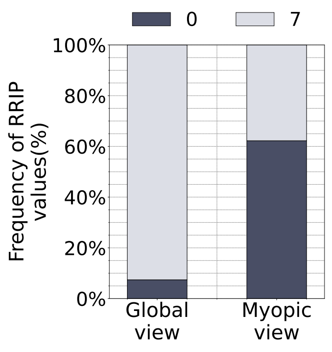
Apart from the myopic behavior and PCs getting scattered across slices, there is a subtle issue with the effectiveness of the sampled



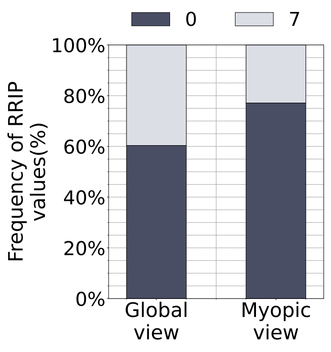
(a) ETR values in xalan with Mockingjay.



(b) ETR values in pr with Mockingjay.

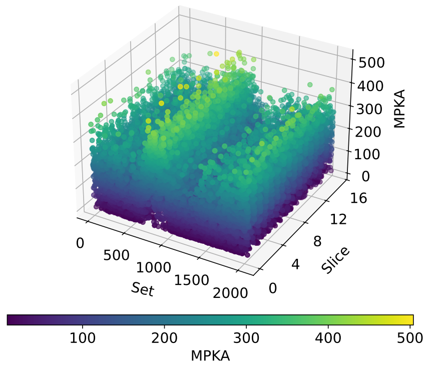


(c) RRIP values in xalan with Hawkeye.

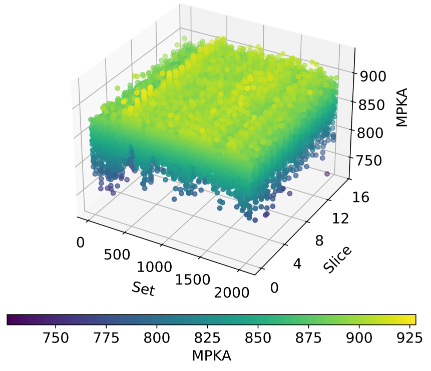


(d) RRIP values in pr with Hawkeye

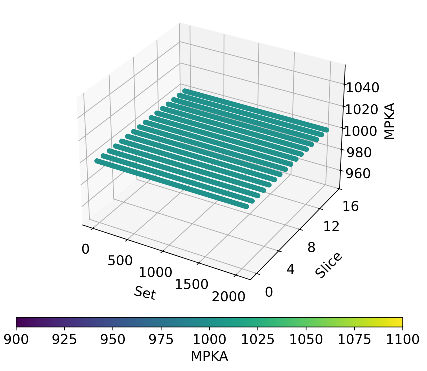
Figure 4: Frequency distribution of ETRs and RRIPs in Mockingjay and Hawkeye for xalan and pr running on a 16-core system.



(a) mcf



(b) gcc



(c) lbm

Figure 5: Miss per kilo accesses (MPKA) per LLC set with three different 16-core homogeneous workloads.

cache that worsens in the sliced LLC as the core count increases. We find that the problem of lower reuse prediction accuracy also arises due to the sampled cache, in which a few LLC sets that are part of the sampled cache do not see enough LLC misses. State-of-the-art replacement policies randomly select a few LLC sets and train their reuse predictor based on accesses in the sampled sets. Some of the sampled LLC sets see few LLC misses, whereas some have high LLC misses. Figure 5 shows miss per kilo access (MPKA) for each LLC set of 16 slices on 16 cores for three homogeneous mixes. For mcf, Figure 5(a) shows that there are many LLC sets that get low MPKA (lower than 100). This trend continues in sampled sets, too. For gcc, Figure 5(b) shows that the trend is better than mcf, and there are a few LLC sets that get MPKA lower than the average MPKA of all sets. Finally, for lbm (Figure 5(c)), which is a streaming workload, all sets get an equal number of accesses with a uniform distribution of MPKAs across all LLC sets.

To investigate the impact of high and low MPKA sets on the reuse predictor, we perform experiments to create three cases. First, we select the top 32 LLC sets with the highest MPKA values and use them for the sampled cache. In the second case, we choose 32 sets with the lowest MPKA values. Third, we sample 16 LLC sets with the highest MPKA values and 16 with the lowest MPKA values. Table 1 presents the results for all cases using a homogeneous mix of mcf.

Table 1: Performance improvements on top of Mockingjay (randomly selected sampled sets) with three cases for selecting the sampled sets based on the MPKAs. We run 16-core homogeneous mix of mcf .

Case	I	II	III
Speedup(%)	16.4	8.3	9.5

In the first case, we observe a performance improvement of 16%. In the second case, the performance improvement is 8.3%, while in the third case, the performance improvement is 9.5%. We see the maximum performance improvement in the first case because the prediction accuracy for high MPKA sets increases due to less noise from low MPKA sets during training. Compared to the second case, in the third case, performance increases further when only half of the sampled sets belong to low MPKA sets. We conclude that the LLC sets with the highest MPKA significantly contribute to the training of the predictor. Based on the observations discussed in this Section, we propose enhancements to state-of-the-art LLC replacement policies, improving their interactions with LLC slices.

3.3 Effect of hardware prefetching

Hardware prefetchers at the L1 and L2 interact with the LLC replacement policies as the prefetched blocks get inserted into the

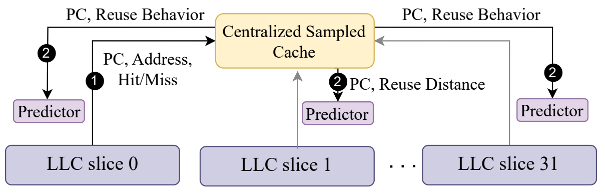


Figure 6: Tracking reuse behavior and training the local predictors with a global (centralized) sampled cache.

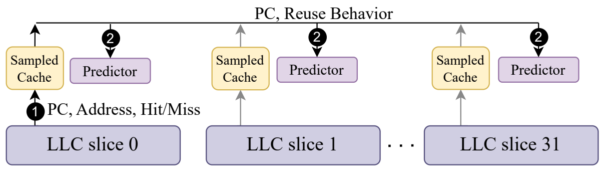


Figure 7: Tracking reuse behavior and training the local predictors with a global (distributed) sampled cache.

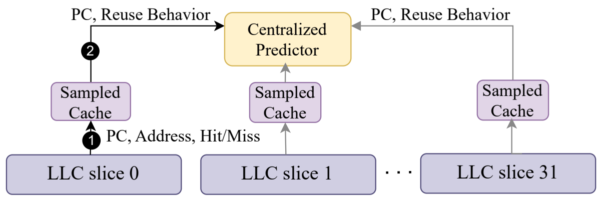


Figure 8: Tracking reuse behavior by a local sampled cache and using it to train the global (centralized) predictor.

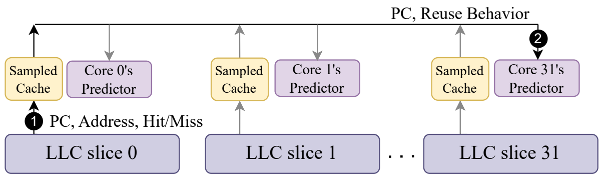


Figure 9: Drishti's enhancement: Tracking reuse behavior and training the per-core and yet global reuse predictor with local (per-slice) sampled caches.

LLC. In our baseline, we use next-line prefetcher at L1D and IP-stride prefetcher at L2. As prefetch requests do not have a PC associated with it, policies like Mockingjay use the PC of the load that triggered the prefetch as the PC of a prefetch request. The predictors use a *prefetch* bit to differentiate a demand load from the same PC. Note that even in the presence of prefetching, the myopic behavior persists both for the demand load PCs and PCs used for prefetch requests. Additionally, the effectiveness of the reuse predictors improves with the accuracy of prefetchers. We find that our observations persist even in the presence of recent prefetchers like SPP+PPF [20],Bingo [16], IPCP [44], Berti [43], and Gaze [21].

4 Drishti Enhancements

In this section, we outline two enhancements that mitigate the effect of the myopic view on reuse predictions and the under-utilization of sampled sets used in the sampled cache.

4.1 Enhancement I: Mitigating myopic predictions

In general, we have four possible design choices: (i) local (per slice) sampled cache and local (per slice) reuse predictor, (ii) global sampled cache and local (per slice) reuse predictor, (iii) local (per slice) sampled cache and global reuse predictor, and (iv) global sampled cache and global reuse predictor. The global sampled cache and the reuse predictor can be designed as a centralized structure, or it can be distributed across LLC slices. Out of these four choices, the first choice leads to myopic decisions affecting the effectiveness of LLC replacement policies. The fourth choice is unnecessary as making one of the structures global will mitigate the myopic predictions. We outline the rest potential design choices that can mitigate the myopic predictions with their pros and cons.

4.1.1 *Global sampled cache and local reuse predictor.*

Centralized sampled cache. In a centralized sampled cache design, a sampled cache is shared by all LLC slices, tracking load accesses across all LLC slices. Figure 6 shows the reuse tracking and training of local predictors by the centralized sampled cache in a 32-core system with 32 slices. In step 1, the centralized sampled cache receives the program counter (PC), block address, and hit/miss status from the LLC slice (slice 0). In step 2, in response, the centralized sampled cache updates all the local predictors spread across all LLC slices as a *broadcast* message with the reuse behavior (e.g., reuse distance in the case of Mockingjay) observed for the given PC. A broadcast is required because the loads associated with a single PC are distributed across multiple slices. As a result, multiple local reuse predictors may hold an entry for the same PC. This holds true even for replacement policies (e.g., Mockingjay) that use a per-slice-per-core reuse predictor (predictor indexed with the hash of PC and core-id), as multiple slices can have the entry for the same hash. Therefore, when the predictor needs to be updated, all corresponding local predictors must be updated simultaneously, necessitating a broadcast.

This centralized structure leads to a bandwidth bottleneck, as an LLC access to a sampled set at any LLC slice will send the PC, block address, and hit/miss information to the global sampled cache. The problem becomes more pronounced as the number of cores increases, further intensifying the bandwidth demand. Additionally, broadcasting from the sampled cache to all the predictors is an additional concern regarding interconnect bandwidth.

Distributed sampled cache. To address the problem with a centralized sampled cache, a distributed sampled cache can be used, which distributes the sampled cache structure across all LLC slices. Figure 7 shows the steps involved in reuse tracking and the training of local predictors by a distributed sampled cache in a 32-core system with 32 slices. In step 1, the sampled cache tracks reuse for its respective slice. However, we call it is global sampled cache because now it can communicate with the local predictors of all the slices. As a result, during the training process, the sampled cache updates all the local predictors based on the tracked reuse information, as shown in step 2. This mitigates the bandwidth issue as it allows concurrent accesses to the distributed sampled cache. However, the *broadcast* message is still a concern, because of which making a sampled cache global, is a costly design choice.

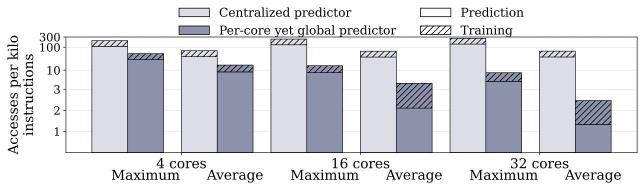


Figure 10: Accesses per kilo instructions to the centralized and per-core global predictors in Mockingjay, averaged across 35 homogeneous and 35 heterogeneous mixes. Each bar shows the training and prediction lookups to the predictor.

4.1.2 *Local sampled cache and global reuse predictor.*

The issue of myopic training and prediction can be effectively addressed by training a single global predictor using the local sampled cache from all slices. This will ensure that, although each local sampled cache tracks reuse behavior specific to its slice, the single predictor is trained on every sampled set access across all LLC slices, resulting in a global training of the predictor.

(a) Centralized reuse predictor. In the case of a centralized reuse predictor, the local sampled caches from all slices compete for the same predictor during updates. Additionally, since the predictor is accessed during every LLC fill, a single predictor must handle all the requests or responses coming from different slices, which can lead to bandwidth and traffic issues that grow with the core count in many-core systems. Figure 8 illustrates the training in the centralized reuse predictor. In step ❶, a sampled set is accessed in LLC slice zero by core 31's request, which updates the sampled cache of slice 0, and in step ❷, the sampled cache updates the centralized reuse predictor.

(b) Distributed (per-core yet global) reuse predictor. In a distributed design, we use a reuse predictor, which is distributed across all slices, with each slice containing a predictor dedicated to a specific core. The predictor for each core is placed closer to the core's nearest LLC slice. To ensure the correct predictor for a given core is accessed, we use a hash of the PC combined with the core ID to retrieve the corresponding entry in the reuse predictor. Figure 9 illustrates the training in per-core yet global predictor, In step ❶, where a sampled set is accessed in LLC slice zero by core 31's request, which updates the sampled cache of slice 0, and in step ❷, sampled cache *only* updates core 31's predictor. *We call this design Drishti's first enhancement to mitigate the myopic reuse predictions.* Table 2 shows the possible design choices for the predictor and sampled cache, along with the associated concerns.

4.1.3 *Reducing traffic with a dedicated interconnect.*

Compared to the baseline design, Drishti's enhancement incurs additional interconnect traffic because of per-core and yet global predictors. Note that without Drishti's enhancements, there is no interconnect traffic between slices and predictors. Figure 10 shows that a centralized reuse predictor, for 32 cores, experiences an average of more than 65 accesses per kilo instruction, with a maximum of 257.76 accesses per kilo instruction (`mcf`). In contrast, the per-core yet global reuse predictor sees only an average of 2.46 accesses per kilo instruction, with a maximum of 8.05 accesses per kilo instructions per core. The accesses include predictor accesses for both training and prediction during an LLC fill.

Table 2: Potential design choices to address myopic predictions, along with their advantages and disadvantages.

Sampled cache	Predictor	Type	Global View?	Bandwidth	Broadcast?
Global	Local	Centralized	Yes	High	Yes
		Distributed	Yes	Low	Yes
Local	**Global**	Centralized	Yes	High	No
		Distributed	Yes	Low	No

Even with a per-core and yet global predictor (each slice contains the predictor for its respective core), the local sampled cache of any slice can access the predictor located in any other slice. However, this introduces additional interconnect latency. This latency grows in many-core systems as the number of slices increases. For a 32-core system, we observe an average interconnect latency of 20 cycles. To overcome this challenge, we use a dedicated and low-latency interconnect (NOCSTAR) [19] that connects all the slices with the predictors, with a three-cycle latency.

4.1.4 *NOCSTAR interconnect.*

NOCSTAR is a side-band, latchless circuit-switched interconnect. With this interconnect, we add latchless switches next to each predictor and slice. A switch is a collection of muxes, where each mux acts like a *repeater*. Compared to a conventional multi-hop Mesh interconnect, the NOCSTAR interconnect uses fewer hops (as few as one if no contention) and provides lower bandwidth as it does not use routers. However, as updates to predictors is not frequent (it is limited to accesses to sampled sets), this low bandwidth and low latency interconnect is sufficient for our need. To ensure concurrent accesses from both request and response (fill) paths, we use two dedicated links.

To ensure communication between a slice and a predictor within a few cycles, NOCSTAR uses separate control wires to acquire all the links in the path between a slice and a predictor. Each LLC slice is connected to an *arbiter* associated with a link through which an LLC slice communicates with the predictor. We use an XY-based routing policy with NOCSTAR.

Static power, area, and dynamic energy. For a 28nm node, with NOCSTAR, the static power consumed by the switch and arbiters is 0.4 mW and two mW, respectively, which is negligible compared to a 2MB LLC slice that consumes static power of 60 mW. In terms of area, the combined area occupied by the switch and the arbiter is 0.005 mm^2, which is negligible to the area occupied by one 2MB LLC slice (1.85 mm^2). The dynamic energy consumed by a slice to predictor communication with the NOCSTAR is negligible, an average of 50pJ (20pJ for link, 10pJ for switch, and 20pJ for control wires) per communication). Figure 11a shows the performance degradation when the reuse predictors use the existing on-chip interconnect compared to a low-latency interconnect[19]. It is interesting to note that the improvement in reuse predictions with Drishti is nullified by the additional interconnect latency. It is visible that if we do not use a low-latency interconnect for slice-to-predictor communication, there will be a performance slowdown compared to the baseline Mockingjay LLC replacement policy. On average, there is a 2.8% performance drop compared to Mockingjay on four cores, which becomes 5.5% for 16 cores and 9% for 32 cores (as high as 40% for `mcf` homogeneous mix) as the interconnect latency increases with the increase in core counts. Figure 11b shows the NOC latency sensitivity on the performance of a 32-core system. Based on Figures 11a and 11b, it is clear that the latency of around 20

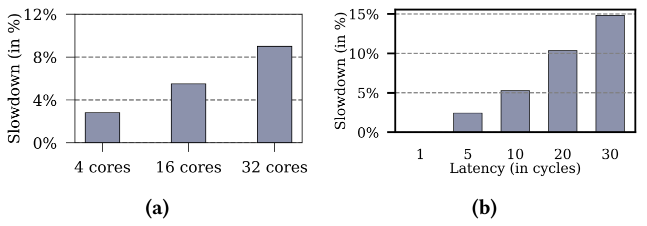


Figure 11: (a) Slowdown in Mockingjay with Drishti without a low-latency interconnect between slices and the predictors. (b) Interconnect latency sensitivity on a 32-core system across 35 homogeneous and 35 heterogeneous mixes.

cycles is the contributor to performance slowdown in the 32-core system. Also, the latency of less than five cycles does not lead to a significant performance slowdown.

Note that with a low-latency, lightweight, and dedicated interconnect like NOCSTAR, we ensure no interference to the existing on-chip interconnect.

4.2 Enhancement-II: Dynamic Sampled Cache

Drishti uses a dynamic sampled cache per slice by sampling the sets with high capacity demands to mitigate the underutilization of sampled cache arising from randomly selected sets. One of the challenges in designing a dynamic sampled cache is identifying and preventing the selection of lower MPKA LLC sets as sampled sets. To overcome this challenge, Drishti identifies the sets with high-capacity demands from each LLC slice.

Identifying LLC sets with high capacity needs. To identify the LLC sets with high capacity demands (higher MPKA), we maintain a k-bit saturating counter that is initially initialized to $\frac{2^k}{2}$ for each set in an LLC slice. To monitor the behavior of LLC sets in terms of MPKA, the counter of each LLC set is incremented on an LLC miss and decremented on an LLC hit. This monitoring is carried out over an interval of L (number of cache lines in an LLC slice) load accesses to an LLC slice. Once the monitoring interval is over, the N sets with the highest saturating counter values are selected as sampled sets in the sampled cache. Here, N represents the number of LLC sets to be sampled per slice. To achieve a balanced trade-off between performance and storage, we empirically determine the optimal number of sampled sets for different replacement policies. Our findings show that Hawkeye and Mockingjay with Drishti require only eight and 16 sampled sets per slice, respectively, whereas previously, they required 64 and 32 sampled sets per slice. This reduction is because Drishti no longer selects sampled sets randomly but does so intelligently. Therefore, by selecting a smaller number of sampled sets, we can still efficiently track the reuse behavior of cache lines. We use k as eight and L as 32K to accurately determine sets with high capacity demands, ensuring each cache line has an equal probability of being accessed.

Phase Change and count reset. To adapt to application phase changes, we identify high-MPKA sets after every 128K load access to an LLC slice (equivalent to four times the number of cache lines in a slice) as it achieves the sweet spot in performance for our workloads. We achieve this by resetting the saturating counter to its initial value and selecting new sampled sets with high MPKA. As some workloads exhibit uniform capacity demand across all LLC sets, such as `lbm`, as shown in Figure 5c, we detect such workloads and compare the highest and lowest MPKA values recorded by the saturating counters. If the difference between these values is less than 100 (average difference across all outlier workloads), we classify the workload as having uniform capacity demand across all sets in an LLC slice. When Drishti encounters a workload with this uniform MPKA behavior, it turns off the dynamic sampled cache and switches to randomly selecting sampled sets (similar to prior approaches) across the LLC slice. Note that predictors are accessed by the sampled cache only when there is an access to a sampled set in the LLC. In our dynamic sampled cache enhancement, we specifically choose sampled sets that experience high capacity demands. This further makes the global sampled cache(Section 4.1.1) an unfavorable design.

4.3 Drishti in action

Figure 12 illustrates the events of interest, keeping Drishti's enhancement in mind. In step ❶, a PC_Y of core 30 generates load requests (A, B, C,..., H) to LLC, due to L2 misses. In step ❷, the load request gets scattered to different slices according to its address, where load addresses: A and D map to slice 0; G and H map to slice 3; E and F map to slice 30; and B and C map to slice 1. In step ❸, all slices update their sampled cache if the access maps to any of their sampled sets, including details such as the PC, whether it was a hit or miss in the LLC set, and the block address of the access request. In step ❹, the sampled cache of different slices (0, 1, 3, 30) trains the core 30's predictor for PC_Y based on accesses in their slices. In step ❺, the predictor returns a predicted value for the PC to install a block in the cache with an ETA counter per cache line. Step 3.1 is triggered for the dynamic sampled cache, where hits/misses per set are monitored for the next 32K accesses. Then, we choose the sets with the highest Misses per Kilo Access (MPKA) (step 3.2). The sampled cache monitors these sets for the next 128K accesses and trains the predictor based on hits or misses for the core's PC. We repeat step 3.1 to step 3.2 for 32K accesses after every 128K load accesses to the LLC slice.

Table 3 illustrates Drishti's per-core hardware budget for Hawkeye and Mockingjay. With Drishti, the number of sampled sets is reduced, leading to a smaller sampled cache size. Specifically, the sampled cache decreases from 12KB to 3KB per core for Hawkeye and from 9.41KB to 4.7KB per core for Mockingjay. Drishti uses saturating counters, which incur a storage overhead of 1.7KB per core. However, the storage savings from the sampled cache outweigh this additional overhead, resulting in a net reduction in per-core storage. As a result, Drishti uses 20.75KB per core with Hawkeye and 28.95KB per core with Mockingjay, which are lower than the previous storage requirements.

5 Evaluation

5.1 Methodology

We use ChampSim [11], a trace-driven simulator used for the 2nd and 3rd Data Prefetching Championships (DPC-2 [2] and DPC-3 [8]). Recent prefetching and cache management proposals are also coded and evaluated on ChampSim [16, 20, 43, 44, 53]. The version employed in DPC-3 has been extended with a decoupled

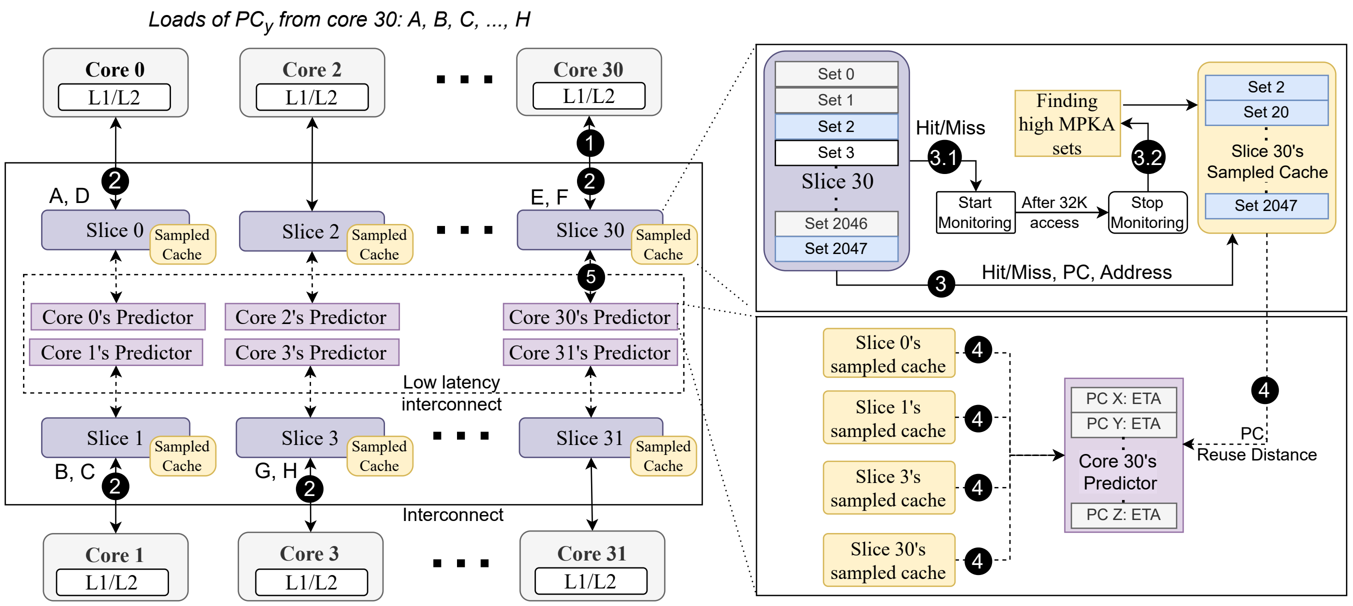


Figure 12: Mockingjay with per-core yet global reuse predictor on a sliced LLC-based 32-core system.

Table 3: Per-core hardware budget with and without Drishti for a 16-way 2MB LLC slice.

Replacement Policy	Components	Without Drishti	With Drishti
Hawkeye	Sampled Cache	12 KB	3 KB
	Occupancy Vector	1 KB	1 KB
	Predictor	3 KB	3 KB
	RRIP counters	12 KB	12 KB
	Saturating counters (2048 entries × 1B)	NA	1.75 KB
	Total	**28 KB**	**20.75 KB**
Mockingjay	Sampled Cache	9.41 KB	4.7 KB
	Predictor	1.75 KB	1.75 KB
	ETR counters	20.75 KB	20.75 KB
	Saturating counters (2048 entries × 1B)	NA	1.75 KB
	Total	**31.91 KB**	**28.95 KB**

Table 4: Simulation parameters of the baseline system.

Core	Out-of-order, hashed-perceptron branch predictor [30], 4 GHz with 6-issue width, 4-retire width, 352-entry ROB
TLBs	L1 iTLB/dTLB: 64 entries, 4-way, 1 cycle STLB: 1536 entries, 12-way, 8 cycles
L1I	32 KB, 8-way, 4 cycles, 8 MSHRs, LRU
L1D	48 KB, 12-way, 5 cycles, 16 MSHRs, LRU, next-line prefetcher
L2	512 KB, 8-way, 15 cycles, 32 MSHRs, SRRIP, non-inclusive, IP-stride prefetcher
LLC	1 slice per core, address to slice mapping as per [33]: 2 MB, 16-way, 20 cycles, 64 MSHRs, LRU, non-inclusive
Network Router	2-stage wormhole, six virtual channels per port, five flit buffer depth, eight flits per data packet, and one flit per address packet
Network Topology	Mesh, each node has a router, processor, private L1 cache, L2 cache, and an LLC slice
DRAM	Controller: One channel/4-cores, 6400 MTPS [22], FR-FCFS, write watermark: 7/8th, Chip: 4 KB row-buffer, open page, t_{RP}: 12.5 ns, t_{RCD}: 12.5 ns, t_{CAS}: 12.5 ns

front-end [49], a detailed memory hierarchy support for address translation, and a faithful DRAM model. We calculate the dynamic energy consumption of the memory hierarchy (caches and DRAM) with CACTI-P [37] and the Micron DRAM [3] power calculator on 7 nm process technology. For interconnect power calculation, we use McPAT [36]. Table 4 details our baseline system configuration, which is similar to an Intel Sunny Cove microarchitecture [6, 7, 26].

We employ publicly available traces [9, 12] from the SPEC CPU 2017 [57] and single-threaded GAP [17] benchmark suites. We use 23 SPEC CPU2017 and 12 GAP benchmarks. SPEC CPU2017 traces were generated with the reference inputs. Both real (Twitter, Web, Road) and synthetic (Kron, Urand) graphs were used as input for the GAP benchmarks.

We limit our study to the memory-intensive traces from SPEC CPU 2017 and all from GAP that exhibit at least one miss per kilo-instruction (MPKI) at the LLC in our baseline system. We run 4-core, 16-core, and 32-core simulations. We collect statistics for 200M sim-point instructions after a 50M-instruction warm-up [54] per core. We simulate 70 mixes (35 homogeneous and 35 heterogeneous), created from SPEC CPU2017 and GAP traces, and report normalized weighted speedup. We use the LRU policy as the baseline LLC replacement policy. For, multi-programmed homogeneous workloads, we run different sim-points of the same benchmark across cores. However, as the number of sim-points [9] is less than the number of cores, especially for 16 and 32 cores, some of the cores run the same trace. For heterogeneous mixes, we create random mixes, similar to Mockingjay [52]. A workload terminates when the slowest core completes simulation of 200M instructions.

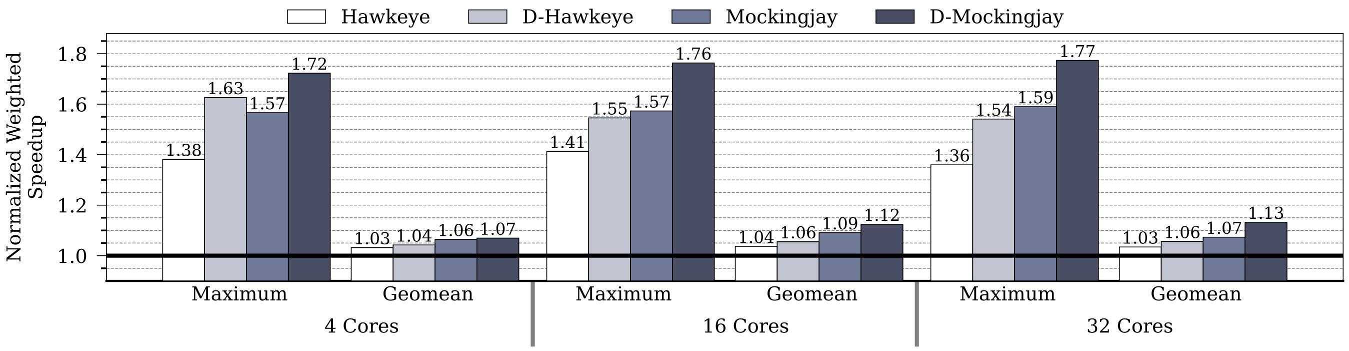


Figure 13: Performance improvement with state-of-the-art LLC replacement policies normalized to LRU on 4-core, 16-core, 32-core systems with 8MB, 32MB, and 64MB sliced LLC across 70 mixes (35 homogeneous and 35 heterogeneous).

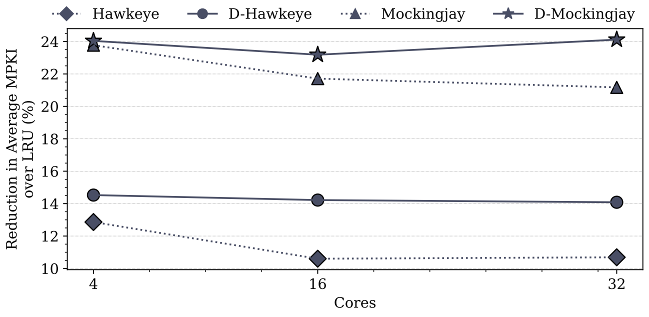


Figure 14: Miss reduction over LRU on 4, 16, and 32 cores averaged across 70 (35 homo. and 35 hetero) mixes.

5.2 Performance analysis

We conduct a thorough performance evaluation of Drishti to assess its effectiveness, focusing on key metrics such as weighted speedup (WS) [56], harmonic mean of speedups(HS) [40], maximum individual slowdown(MIS), and unfairness [42]. These metrics are crucial in many-core systems. The equations below define these metrics.

$$IS_i = \frac{IPC_i^{together}}{IPC_i^{alone}}, \qquad WS = \sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}},$$

$$HS = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}, \quad Unfairness = \frac{\max\{IS_0, IS_1, \dots, IS_{N-1}\}}{\min\{IS_0, IS_1, \dots, IS_{N-1}\}}$$

$$MIS = MAX\{IS_0, IS_1, \dots, IS_{N-1}\},$$

Here, IPC_i^{together} is the IPC of core i when it runs along with other $N-1$ applications on a multi-core system of N cores. IPC_i^{alone} is the IPC of core i when it runs alone on a multi-core system of N cores. The rest of the $N-1$ cores are idle. IS_i corresponds to individual slowdown of core i.

We refer to Drishti's enhancement by adding the prefix 'D-' to the name of the original replacement policy. For example, Mockingjay enhanced with Drishti is referred to as D-Mockingjay, and Hawkeye enhanced with Drishti is referred to as D-Hawkeye.

Performance. Figure 13 shows the performance improvement of D-Hawkeye and D-Mockingjay on 4,16 and 32 cores. On a 4-core system, Hawkeye and Mockingjay achieve performance improvements of 3.1% and 6.4%, respectively. With Drishti's enhancement, D-Hawkeye and D-Mockingjay further deliver performance improvements of 4.2% and 6.9%, demonstrating the marginal utility of Drishti. However, as the number of cores increases, the utility of Drishti becomes more pronounced. On a 32-core system, Hawkeye and Mockingjay achieve performance improvements of 3.3% and 6.7%, respectively, while D-Hawkeye and D-Mockingjay deliver significantly higher gains of 5.6% and 13.2%. Notably, for the `mcf_1554B` workload on a 32-core system, D-Mockingjay achieves a maximum improvement of 77%, compared to 59% with Mockingjay.

Table 5: Average LLC WPKI across 35 homogeneous mixes and 35 heterogeneous mixes.

Cores	LRU	Hawkeye	D-Hawkeye	Mockingjay	D-Mockingjay
4	0.18	1.48	2.63	7.64	6.20
16	0.18	1.15	2.63	7.16	7.02
32	0.17	1.23	2.60	7.26	6.98

Reduction in LLC MPKI. Drishti's performance improvements come because of a reduction in LLC MPKIs. Figure 14 shows the reduction in average LLC Misses across all slices for Hawkeye, D-Hawkeye, Mockingjay, and D-Mockingjay as compared to LRU. On four cores, Hawkeye reduces the average MPKI by 12.9%, whereas D-Hawkeye reduces the average MPKI by 14.5%. Mockingjay reduces the average MPKI by 23.8%, whereas D-Mockingjay reduces the average MPKI by 24%. On 32 cores, Hawkeye reduces the average MPKI by 10.6%, whereas D-Hawkeye reduces the average MPKI by 14.1%, and Mockingjay reduces the average MPKI by 21.2%. In contrast, D-Mockingjay reduces the average MPKI by 24.1%. The effectiveness of enhancements improves with an increase in core count, as evident in the reduction in MPKI and slight increase in LLC miss latency.

Reduction in LLC writebacks. Table 5 shows LLC WPKI (Writebacks per kilo instruction) values for LRU, Hawkeye, D-Hawkeye, Mockingjay, and D-Mockingjay. The number of writebacks from LLC increases in Hawkeye and Mockingjay because all these policies assign the lowest priority to dirty lines, thereby increasing the rate at which writebacks occur between LLC and DRAM.

Energy Consumption. Figure 15 shows a reduction of dynamic energy consumption in LLC, NOC, and DRAM across 16-core and 32-core systems. On 32 cores, Hawkeye and Mockingjay reduce

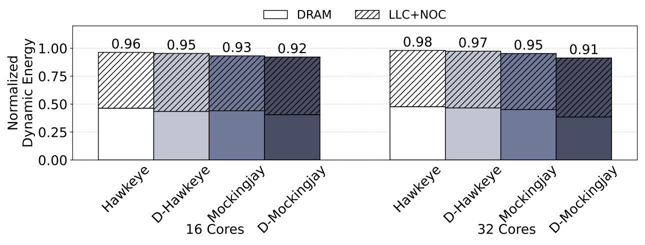


Figure 15: Uncore (LLC, NOC, and DRAM) energy consumption normalized to LRU across 70 mixes (lower the better).

Table 6: Drishti on a 32-core system with 64MB LLC across 35 homogeneous and 35 heterogeneous mixes.

Metrics	Hawkeye	D-Hawkeye	Mockingjay	D-Mockingjay
WS(%)	3.3	5.6	6.7	13.3
HS(%)	3.4	5	4.5	12.8
Unfairness	1.2	1.2	1.30	1.28
MIS(%)	41.4	40	37	34.2

uncore energy consumption by 2% and 5%, respectively, whereas D-Hawkeye and D-Mockingjay reduce uncore energy consumption by 3% and 9%, respectively. Note that for D-Hawkeye and D-Mockingjay, NOC includes the energy of NOCSTAR, too. The uncore energy savings in D-Mockingjay come from reduced DRAM reads and LLC writebacks.

Other metrics of interest. Table 6 shows the effectiveness of Drishti enhancements for various performance and fairness metrics. In terms of weighted speedup, Hawkeye achieves a 3.3% performance improvement, which becomes 5.6% with D-Hawkeye. Similarly, Mockingjay achieves 6.7% performance improvement, which becomes 13.3% with D-Mockingjay. In terms of the harmonic mean of speedups, Hawkeye achieves 3.4% performance improvement, which becomes 5% with D-Hawkeye. Similarly, Mockingjay achieves 4.5% performance improvement, which becomes 12.8% with D-Mockingjay, a significant improvement. In the case of unfairness and maximum individual slowdown, Drishti enhancements do not contribute much and are similar to Hawkeye and Mockingjay.

5.3 Drishti with Mockingjay: A detailed analysis

In this section, we provide a detailed analysis of D-Mockingjay, highlighting the benefits of our two main contributions: the per-core yet global reuse predictor and the dynamic sampled cache. Figure 16 compares the performance of Mockingjay and D-Mockingjay across 70 32-core workload mixes using the normalized weighted speedup metric. D-Mockingjay consistently outperforms Mockingjay across all workload mixes. For a 32-core homogeneous mix of `mcf` with a large fraction of `605.mcf_s-1554B`, D-Mockingjay achieves 77% performance improvement over LRU, which was only 59% with mockingjay. This boost is primarily driven by Drishti's dynamic sampling approach and a marginal effect from the per-core global predictor, as `605.mcf_s-1554B` shows non-uniformity in sampled set access, with some sets having lower MPKA and others with higher MPKA, and only 60% of PCs are mapped to one slice. For the

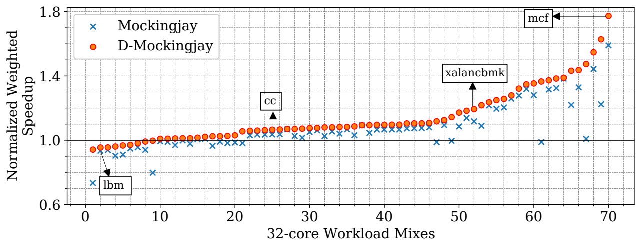


Figure 16: Performance of Mockingjay and D-Mockingjay on 32-core systems across 70 mixes. The mixes are shown in sorted order (as per performance improvement), and their indices do not correspond to those in Figure 2.

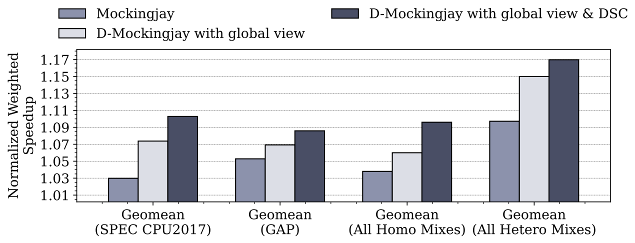


Figure 17: Performance normalized to LRU with only global view and D-Mockingjay with global view & DSC across 32-core 35 homogeneous and heterogeneous mixes.

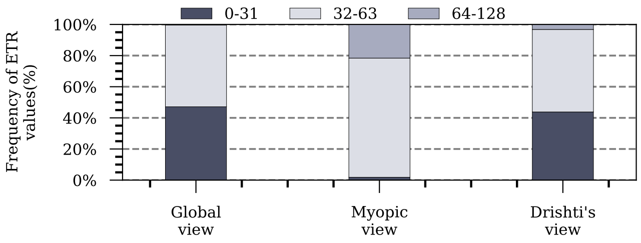


Figure 18: ETR values with Drishti in Mockingjay for `xalan` running on a 16-core system.

homogeneous mix dominated by `xalancbmk-202B`, D-Mockingjay enhances the performance by 26%, which is 20% with Mockingjay. For `xalancbmk`, the primary contributor to the performance is the conversion from myopic to global view, as it is one of the mixes that is significantly affected by the myopic view, as shown in Figure 2.

Utility of each enhancement. Figure 17 shows the utility of each enhancement. Mockingjay delivers an average performance improvement of 3.8% and 9.7% over LRU across SPEC and GAP homogeneous and heterogeneous respectively. When the per-core global predictor (D-Mockingjay with a global view) is introduced, the average speedup across SPEC and GAP increases to 6% and 15%, with a 7.4% improvement for SPEC and a 6.9% improvement for GAP. On top of this, performance improvement occurs when the dynamic sampled cache (DSC) is integrated with the global predictor (D-Mockingjay with a global view and DSC), resulting in an average speedup of 9.7% and 16.9% across both SPEC and GAP homogeneous and heterogeneous, respectively, with a 10.2% improvement for SPEC and an 8.5% improvement for GAP.

Figure 18 shows the utility of Drishti's enhancement as it provides a view closer to the global view. In the case of Drishti's view,

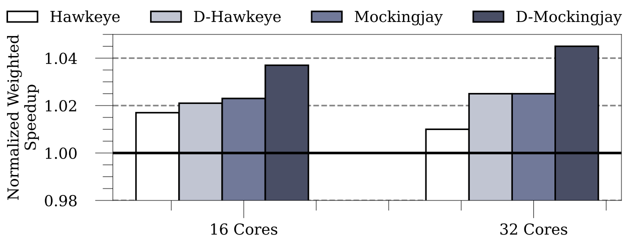


Figure 19: Performance of Hawkeye, D-Hawkeye, Mockingjay, D-Mockingjay on 50 16-core and 50 32-core mixes created from CVP1 [46], Google datacenter traces [15, 51], CloudSuite [4], and XSBench [14].

the ETR predictions are closer to the global view, leading to high accuracy in ETR prediction. In the case of a homogeneous mix dominated by `619.lbm_s-2676B`, Mockingjay incurs a performance slowdown of about 7%, with Drishti this degradation becomes 4%. In general, Mockingjay does not perform well for streaming workloads, and Drishti's enhancement improves it slightly as it takes care of the myopic view.

Scalability. So far, we have evaluated Drishti with 4, 16, and 32 cores. We also evaluate D-Mockingjay for 64 and 128 cores with 128MB and 256MB L3 cache with 64 and 128 2MB slices, respectively. We observe similar average performance improvements with D-Mockingjay as compared to Mockingjay for 64 and 128 cores. In general, we see a positive trend as we increase the core count, and D-Mockingjay remains effective with 64 and 128 cores (additional performance improvement of 1% as compared to 32 cores).

Drishti with other workloads. Figure 19 shows the effect of Drishti enhancements on CVP1 [46], CloudSuite [4], Google datacenter [15], and XSBench [14] traces. For CVP1, we use the updated traces that appeared in IISWC 2023 [25], and for Google datacenter traces, we use the artifact of the MICRO'24 paper [51]. We create 50 random mixes for 16 and 32-core systems. Compared to SPEC-CPU2017 and GAP workloads, the effectiveness of replacement policies like Hawkeye and Mockingjay is in the range of 2 to 3%, with a maximum improvement of 13%. Drishti's enhancement improves the effectiveness of Hawkeye and MockingJay by an additional 2%, on average, when compared to a baseline LRU.

5.4 Sensitivity Studies

In this Section, we perform a sensitivity study running homogeneous mixes on a 16-core simulated system.

Drishti with different LLC sizes. Figure 20 shows the performance of Drishti on varying the size of the LLC slice on 16 cores. We observe that the effectiveness of Drishti on Hawkeye and Mockingjay remains the same regardless of changes in LLC slice size. In our baseline, we use 2MB LLC/core. Next, we increase the LLC size from 1MB/core to 4MB/core, keeping the number of sampled sets fixed as per 2MB LLC. We observe that Drishti performs best on an LLC slice of 2MB because 1MB/core demands a smaller number of sampled sets in the sampled cache, whereas 4MB/core demands a larger number of sets in the sampled cache for training.

Drishti with different L2 sizes. Figure 21 shows the performance of Drishti on varying the size of the L2 cache on 16 cores. Drishti

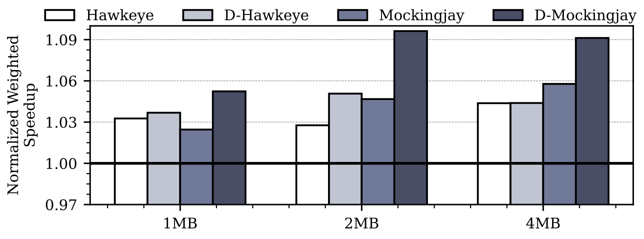


Figure 20: Performance normalized to LRU on a 16-core system with different sizes of sliced LLC.

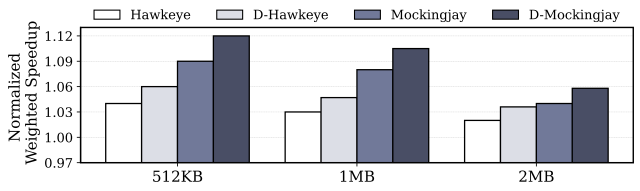


Figure 21: Performance normalized to LRU on a 16-core system with different sizes of L2.

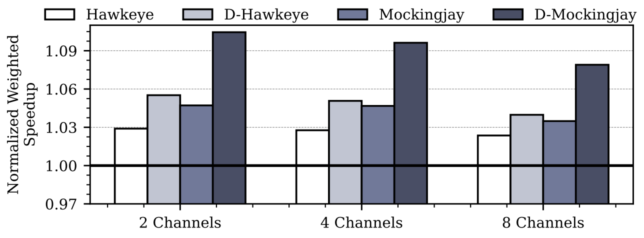


Figure 22: Performance normalized to LRU on a 16-core system with different DRAM channels.

enhances the effectiveness of Hawkeye and Mockingjay across different L2 sizes. However, with a larger L2, such as 2MB, the rate of performance improvement with Hawkeye and Mockingjay drops as more applications' working sets start fitting in L2, resulting in LLC MPKI < 1 in the baseline itself.

Drishti with different DRAM channels. Figure 22 shows the performance of Drishti as we increase the number of DRAM channels. In our baseline, we use four channels for 16 cores. When we move to two channels for 16 cores, Hawkeye provides only 2.3% performance improvement, and Drishti enhances this performance improvement to 5.5% with D-Hawkeye. Mockingjay provides 4.7% performance improvement, and Drishti enhances this performance improvement to 10.4% with D-Mockingjay. In the case of eight channels for 16 cores, the effectiveness of replacement policies goes down as LLC miss latency improves.

Drishti with hardware prefetchers. So far, we use next-line at L1 and IP-stride at L2. Now, we evaluate Drishti with five state-of-the-art L1 and L2 prefetchers: SPP+PPF [20], Bingo [16], IPCP [44], Berti [43], and Gaze [21]. Figure 23 shows the performance improvements with Drishti normalized to five different baselines with five different prefetchers. In general, we see synergy between DRSIHTI's enhancements and hardware prefetching techniques. We observe that highly accurate prefetchers, such as SPP+PPF and Berti, improve the baseline itself, and the scope for improvement

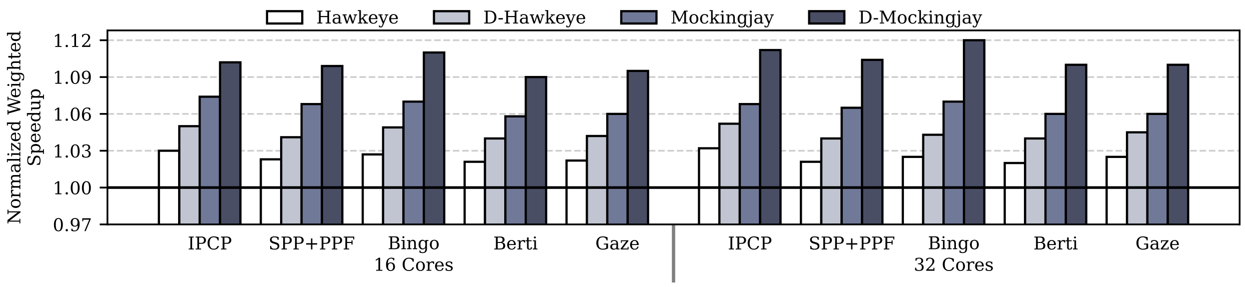


Figure 23: Performance normalized to LRU with different hardware prefetchers averaged across 16 and 32-core mixes.

Table 7: Applicability across LLC replacement policies.

Type	Replacement Policies	Per-core yet Global Predictor	Dynamic Sampled Cache
Memoryless Policies	DIP, RRIP, IPVs	×	✓
Prediction-based Policies	SDBP, SHiP, SHiP++,Leeway, Glider, MPPPB, Percepton learning, MDPP, CARE, CHROME	✓	✓
	EVA	×	×

is marginally lower. In DRAM bandwidth-constrained scenarios, CLIP [45] can be used alongside Drishti.

6 Applicability

Memoryless policies. Rereference Interval Prediction (RRIP), Dynamic Insertion Policy (DIP), and Insertion and Promotion Vector (IPV) [28, 32, 48] employ set-dueling to choose between heuristics. Still, the random selection of sampled sets can lead to suboptimal outcomes. Drishti's dynamic sampled set selection can enhance the effectiveness of the selection.

Prediction based policies. Replacement policies in this category use a prediction-based approach to making eviction decisions. They use common structures like sampled cache and reuse predictors. Sampling Dead Block Prediction (SDBP) [34] determines whether the loads brought by a particular PC are dead lines. To do this, it tracks the reuse patterns of loads associated with that PC. Signature-based Hit Predictor (SHiP++) [60, 61] learns reuse patterns using sampled sets that store the most recent signature per line and update the Signature-based Hit Table (SHCT). Economic Value Added (EVA) [18] prioritizes cache lines based on cache hit distribution. Leeway [23] improves dead block prediction by using reuse distance and decides eviction based on access patterns. Leeway is one of the prior works that tries to optimize the latency and energy requirements for accessing the prediction tables. Instead of accessing the prediction table on every LLC hit and miss, Leeway designs a predictor that is accessed only on LLC misses. However, it still suffers from myopic behavior and underutilized sample sets. Perceptron learning [59] and Multiperspective Placement, Promotion, and Bypass (MPPPB) [31] uses different features along with PC to predict the reuse behavior of the cache block using sampled sets from LLC. Minimal Disturbance Placement and Promotion (MDPP) [58] focuses on advancing reused blocks that have lower protection levels, ensuring minimal disruption to the remaining blocks in the set using a PC trained on sampled sets. Glider [55] is a deep learning-based policy that uses a PC History Register (PCHR) as a sampled cache and an Integer SVM (ISVM) as a predictor to estimate cache line reuse behavior. CARE [39] is a concurrency-aware replacement policy that considers both cache locality and concurrency using the metric of pure miss cycles to decide the caching behavior of the PC. It also uses a sampled cache and predictor indexed by PC per slice. CHROME [38] is an RL-based replacement policy that utilizes the SARSA algorithm to determine caching actions at the levels of PC and DRAM page.

Table 8: Drishti with SHiP++, CHROME, and Glider policies for 16-core systems.

SHiP++	D-SHiP++	CHROME	D-CHROME	Glider	D-Glider
1.03	1.08	1.06	1.13	1.03	1.06

Table 7 summarizes Drishti's applicability across various policies based on its two enhancements. Table 8 shows the effectiveness of Drishti with three more replacement policies for a 16-core system. We select SHiP++ as SHiP is one of the early RRIP-based policies that uses a predictor. Next, we choose CHROME, an RL-based replacement policy, and Glider, a deep learning based replacement policy. Drishti's enhancements are effective across all these policies. SHiP++'s 3% gain over LRU increases to 8% with D-SHiP++. CHROME improves performance by 6% over LRU, which increases to 13% with D-CHROME and Glider's 3% gain became 6% with D-Glider.

7 Conclusion

We proposed a few fundamental enhancements (Drishti enhancements) to improve the effectiveness of last-level cache replacement policies for a sliced LLC with NUCA, which is shared by many cores. We observed that the reuse predictions made by state-of-the-art replacement policies are myopic. To mitigate the myopic predictions, we made a case for a local per-slice sampled cache and per-core yet global reuse predictor, which is connected to an LLC slice through a dedicated interconnect. We also proposed a dynamic sampled cache to mitigate the issue of underutilized sampled sets across LLC slices. Our enhancements improve the effectiveness of the state-of-the-art LLC replacement policies, especially for many-core systems.

Acknowledgments

We would like to thank members of the CASPER research group and Anuj for the implementation of sliced LLC in champsim. This work is supported by the Trust Lab Grant 2023 and an AMD research gift.

References

[1] 2010. The 1^{st} Cache Replacement Championship (CRC-1, JWAC-1) @ ISCA. https://jilp.org/jwac-1/.

[2] 2015. The 2nd Data Prefetching Championship (DPC-2). https://comparch-conf.gatech.edu/dpc2/

[3] 2015. Micron DRAM Power Calculator. https://www.micron.com/-/media/client/global/documents/products/technical-note/dram/tn4007_ddr4_power_calculation.pdf.

[4] 2017. CloudSuite traces for ChampSim. https://www.dropbox.com/sh/pgmnzfr3hurlutq/AACciuebRwSAOzhJkmj5SEXBa/CRC2_trace?dl=0&subfolder_nav_tracking=1.

[5] 2017. The 2^{nd} Cache Replacement Championship (CRC-2) @ ISCA. https://crc2.ece.tamu.edu/.

[6] 2018. SunnyCove microarhcitecture. https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove.

[7] 2018. SunnyCove microarhcitecture latency. https://www.7-cpu.com/cpu/Ice_Lake.html.

[8] 2019. The 3rd Data Prefetching Championship (DPC-3). https://dpc3.compas.cs.stonybrook.edu/

[9] 2019. SPEC CPU 2017 traces for ChampSim. https://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/.

[10] 2020. AMD Zen3 L3 cache. https://www.anandtech.com/show/16214/amd-zen-3-ryzen-deep-dive-review-5950x-5900x-5800x-and-5700x-tested/4.

[11] 2020. ChampSim simulator. http://github.com/ChampSim/ChampSim.

[12] 2021. GAP traces for ChampSim. https://utexas.app.box.com/s/2k54kp8zvrqdfaa8cdhfquvcxwh7yn85/folder/132804598561.

[13] 2021. Intel xeon L3 cache. https://sites.utexas.edu/jdm4372/2021/09/10/mapping-addresses-to-l3-cha-slices-in-intel-processors/.

[14] 2021. XSBench traces for ChampSim. âĂœXSBench,âĂİhttps://github.com/ANL-CESAR/XSBench.

[15] 2024. Google datacenter traces for ChampSim. https://dynamorio.org/google_workload_traces.html.

[16] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 399–411. https://api.semanticscholar.org/CorpusID:89617535

[17] Scott Beamer, Krste Asanović, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (Aug. 2015). https://doi.org/10.48550/arXiv.1508.03619

[18] Nathan Beckmann and Daniel Sánchez. 2017. Maximizing Cache Performance Under Uncertainty. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), 109–120. https://api.semanticscholar.org/CorpusID:6441937

[19] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. 2018. Scalable Distributed Last-Level TLBs Using Low-Latency Interconnects. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 271–284. https://doi.org/10.1109/MICRO.2018.00030

[20] Eshan Bhatia, Gino Chacon, Seth H. Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-based prefetch filtering. In *46th Int'l Symp. on Computer Architecture (ISCA)*. 1–13. https://doi.org/10.1145/3307650.3322207

[21] Zixiao Chen, Chentao Wu, Yunfei Gu, Ranhao Jia, Jie Li, and Minyi Guo. 2025. Gaze into the Pattern: Characterizing Spatial Patterns with Internal Temporal Correlations for Hardware Prefetching. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025*. IEEE, 173–187. https://doi.org/10.1109/HPCA61900.2025.00024

[22] DDR. 2024. DDR standards. https://en.wikipedia.org/wiki/Double_data_rate. https://en.wikipedia.org/wiki/Double_data_rate

[23] Priyank Faldu and Boris Grot. 2017. Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 180–193. https://doi.org/10.1109/PACT.2017.32

[24] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3302424.3303977

[25] Josué Feliu, Arthur Perais, Daniel A. Jiménez, and Alberto Ros. 2023. Rebasing Microarchitectural Research with Industry Traces. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 100–114. https://doi.org/10.1109/IISWC59245.2023.00027

[26] Agner Fog. 2020. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Available at https://www.agner.org/optimize/microarchitecture.pdf.

[27] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 78–89. https://doi.org/10.1109/ISCA.2016.17

[28] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[29] Daniel A. Jiménez. 2013. Insertion and promotion for tree-based PseudoLRU last-level caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 284–296. https://doi.org/10.1145/2540708.2540733

[30] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 197–206. https://doi.org/10.1145/859618.859655

[31] Daniel A. Jiménez and Elvira Teran. 2017. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 436–448. https://doi.org/10.1145/3123939.3123942

[32] Daniel A. Jiménez, Elvira Teran, and Paul V. Gratz. 2023. Last-Level Cache Insertion and Promotion Policy in the Presence of Aggressive Prefetching. *IEEE Computer Architecture Letters* 22, 1 (2023), 17–20. https://doi.org/10.1109/LCA.2023.3242178

[33] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. 2016. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 72, 6 pages. https://doi.org/10.1145/2897937.2897962

[34] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 175–186. https://doi.org/10.1109/MICRO.2010.24

[35] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. 2007. Comparing memory systems for chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. Association for Computing Machinery, New York, NY, USA, 358–368. https://doi.org/10.1145/1250662.1250707

[36] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[37] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*. 694–701. http://dx.doi.org/10.1109/ICCAD.2011.6105405

[38] Xiaoyang Lu, Hamed Najafi, Jason Liu, and Xian-He Sun. 2024. CHROME: Concurrency-Aware Holistic Cache Management Framework with Online Reinforcement Learning. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 1154–1167. https://doi.org/10.1109/HPCA57654.2024.00090

[39] Xiaoyang Lu, Rujia Wang, and Xian-He Sun. 2023. CARE: A Concurrency-Aware Enhanced Lightweight Cache Management Framework. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 1208–1220. https://doi.org/10.1109/HPCA56546.2023.10071125

[40] Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. 164–171. https://doi.org/10.1109/ISPASS.2001.990695

[41] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404* (Kyoto, Japan) *(RAID 2015)*. Springer-Verlag, Berlin, Heidelberg, 48–65. https://doi.org/10.1007/978-3-319-26362-5_3

[42] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 146–160. https://doi.org/10.1109/MICRO.2007.21

[43] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibánez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an Accurate Local-Delta Data Prefetcher. In *55thInt'l Symp. on Microarchitecture (MICRO)*. 975–991. http://dx.doi.org/10.1109/MICRO56248.2022.00072

[44] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *47th Int'l Symp. on Computer Architecture (ISCA)*. 118–131. http://dx.doi.org/10.1109/

ISCA45697.2020.00021

[45] Biswabandan Panda. 2023. CLIP: Load Criticality based Data Prefetching for Bandwidth-constrained Many-core Systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 714–727. https://doi.org/10.1145/3613424.3614245

[46] Arthur Perais. 2018. 1^{st} Championship Value Prediction Public Traces. https://doi.org/10.18709/perscido.2023.02.ds382.

[47] M.K. Qureshi, D. Thompson, and Y.N. Patt. 2005. The V-Way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 544–555. https://doi.org/10.1109/ISCA.2005.52

[48] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 381–391. https://doi.org/10.1145/1273440.1250709

[49] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch Directed Instruction Prefetching. In *32nd Int'l Symp. on Microarchitecture (MICRO)*. 16–27. http://dx.doi.org/10.1109/MICRO.1999.809439

[50] Dyer Rolán, Basilio B. Fraguela, and Ramón Doallo. 2009. Adaptive line placement with the set balancing cache. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/1669112.1669178

[51] David Schall, Andreas Sandberg, and Boris Grot. 2024. The Last-Level Branch Predictor. In *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*. IEEE, 464–479. https://doi.org/10.1109/MICRO61859.2024.00042

[52] Ishan Shah, Akanksha Jain, and Calvin Lin. 2022. Effective Mimicry of Belady's MIN Policy. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 558–572. https://doi.org/10.1109/HPCA53966.2022.00048

[53] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Multi-Lookahead Offset Prefetching. In *The 3rd Data Prefetching Championship*. https://api.semanticscholar.org/CorpusID:199570386

[54] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 45–57. https://doi.org/10.1145/635508.605403

[55] 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3352460.3358319

[56] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. Association for Computing Machinery, New York, NY, USA, 234–244. https://doi.org/10.1145/378993.379244

[57] Standard Performance Evaluation Corporation. 2017. SPEC CPU2017. http://www.spec.org/cpu2017

[58] Elvira Teran, Yingying Tian, Zhe Wang, and Daniel A. Jiménez. 2016. Minimal disturbance placement and promotion. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 201–211. https://doi.org/10.1109/HPCA.2016.7446065

[59] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. 2016. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783705

[60] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. 2011. SHiP: Signature-based Hit Predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 430–441. https://doi.org/10.1145/2155620.2155671

[61] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moin Qureshi. 2017. SHiP + + : Enhancing Signature-Based Hit Predictor for Improved Cache Performance. https://api.semanticscholar.org/CorpusID:43689204

[62] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. 2010. STEM: Spatiotemporal Management of Capacity for Intra-core Last Level Caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, USA, 163–174. https://doi.org/10.1109/MICRO.2010.31