Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy

Jinchun Kim
Texas A&M University
cienlux@tamu.edu

Elvira Teran
Texas A&M University
eteran@tamu.edu

Paul V. Gratz
Texas A&M University
pgratz@gratz1.com

Daniel A. Jiménez
Texas A&M University
djimenez@cse.tamu.edu

Seth H. Pugsley
Intel Labs
seth.h.pugsley@intel.com

Chris Wilkerson
Intel Labs
chris.wilkerson@intel.com

Abstract

Data prefetching and cache replacement algorithms have been intensively studied in the design of high performance microprocessors. Typically, the data prefetcher operates in the private caches and does not interact with the replacement policy in the shared Last-Level Cache (LLC). Similarly, most replacement policies do not consider demand and prefetch requests as different types of requests. In particular, program counter (PC)-based replacement policies cannot learn from prefetch requests since the data prefetcher does not generate a PC value. PC-based policies can also be negatively affected by compiler optimizations. In this paper, we propose a holistic cache management technique called Kill-the-PC (KPC) that overcomes the weaknesses of traditional prefetching and replacement policy algorithms. KPC cache management has three novel contributions. First, a prefetcher which approximates the future use distance of prefetch requests based on its prediction confidence. Second, a simple replacement policy provides similar or better performance than current state-of-the-art PC-based prediction using global hysteresis. Third, KPC integrates prefetching and replacement policy into a whole system which is greater than the sum of its parts. Information from the prefetcher is used to improve the performance of the replacement policy and vice-versa. Finally, KPC removes the need to propagate the PC through entire on-chip cache hierarchy while providing a holistic cache management approach with better performance than state-of-the-art PC-, and non-PC-based schemes. Our evaluation shows that KPC provides 8% better performance than the best combination of existing prefetcher and replacement policy for multi-core workloads.

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Cache memories

Keywords Memory Hierarchy, Data Prefetching, Cache Replacement Policy

1. Introduction

Due to the combined pressures of increasing application working sets [7, 21], the persistence of the Memory Wall [38], and the breakdown of Dennard scaling [5], processor memory system hierarchies have continued to grow in complexity, size, and performance criticality. In recent architectures, caches consume as much as 40% of the total die area and 20% of energy consumption on chip [35]. With so much of the available on-die resources invested in the cache hierarchy, an efficient, high performance design requires intelligent cache management techniques. While many cache management and speculation techniques such as alternate replacement policies [6, 14, 15, 18, 27], deadblock/hit prediction [17, 20, 28, 33, 36], and prefetching techniques [2, 10, 16, 19, 19, 23, 26, 31, 32] have been extensively explored, many of these are piecemeal, one-off solutions that often interact poorly when implemented together and typically only address one level of the memory-system hierarchy. There has been little work exploring the interactions between these policies across multiple levels of the memory hierarchy and examining the information needed across boundaries in the system from software to the core, to the last level cache. This paper proposes a holistic, speculative, multi-level cache management system that effectively reconstructs program behavior in the processor memory hierarchy to prefetch and manage placement of data across the cache hierarchy.

Without coordination between cache management and speculation techniques at different levels in the cache hierarchy, schemes such as data prefetching and replacement

ASPLOS '17, April 08-12, 2017, Xi'an, China

DOI: http://dx.doi.org/10.1145/3037697.3037701

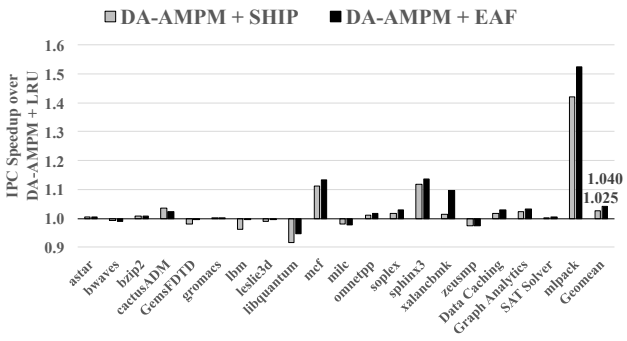


Figure 1: Performance of SHiP [36] (PC-based replacement policy) and EAF [28] (non-PC-based replacement policy) with the DA-AMPM prefetcher, normalized against DA-AMPM with LRU.

often work at cross-purposes. Data prefetching is a well known mechanism that significantly improves performance by preloading future memory accesses into some level of the cache ahead of its actual use. Previous work suggests advanced prefetching algorithms [2, 10, 16, 19, 23, 26, 31, 32] to reduce the gap between processor speed and memory latency. Most data prefetchers are trained by private L1 or L2 cache accesses to make timely prefetches far ahead of demand requests. Often, however, the appropriate placement within the shared LLC for these prefetched blocks is unclear. A sophisticated cache replacement policy is the right tool to solve this problem. Previous work [14, 17, 18, 20, 27, 28, 33, 36] shows a substantial gain can be achieved by placing blocks predicted to be dead [20] at the vulnerable position in the LRU stack. However, with data prefetches, the incremental benefit of replacement policy often becomes marginal or sometimes even negative [11, 29, 37].

In particular, replacement policies which use the PC of missing load to predict reuse [17, 18, 20, 33, 36] experience substantial interference from prefetched blocks, which by definition do not carry demand fetch load PC values. Figure 1 compares the IPC speedup of a top performing, PC-based replacement policy (SHiP [36]), and a recently proposed non-PC-based replacement policy (EAF [28]) when a high performance data prefetcher, DA-AMPM[1] [11] is being used. In this figure, the performance is normalized to DA-AMPM with the baseline LRU replacement policy. Although SHiP typically shows better performance than EAF when running without prefetching, we see that here EAF outperforms SHiP across most applications. This is largely because PCs are simply not available for prefetches, forcing SHiP to use a static prediction (always dead or always live) for prefetched blocks. On the other hand, EAF tracks the physical addresses of recently evicted blocks in a bloom filter to make a dead block prediction. When there is a cache miss and the missing block is found in the victim filter, that block is inserted with higher priority. The baseline assumption is that if a block with high reuse is prematurely evicted from the cache, it will be accessed soon after eviction [28]. Thus, EAF can make a per-block-based prediction for both demand and prefetch that yields higher performance than SHiP without relying on PCs.

To alleviate the harmful interference between prefetching and replacement policy, several works [11, 29, 37] propose to selectively prioritize the prefetch request over demand request or vice versa. While these approaches show some gains, there remains little integration between the techniques, leaving critical program behavior information known by the prefetcher out of the replacement/placement decision, thus leaving performance on the table.

In this paper, we present a novel cache management mechanism called Kill-the-PC (KPC) that integrates data prefetching and replacement policy across multiple levels of the cache hierarchy into a single system. KPC consists of two main components. First, we develop a prefetcher (KPC-P) that produces a proxy of future use distance based on its prediction confidence. KPC-P generates a confidence value which is used to determine which cache level to insert the prefetched blocks. Second, we propose a replacement policy (KPC-R) that quickly adapts to the dynamic program phase using two small global counters. Each counter is exclusively updated by demand or prefetch so that KPC-R can predict useless cache blocks for both memory requests.

Additionally, KPC-R and KPC-P are integrated to share information. For example KPC-P learns from KPC-R to dynamically adjust a threshold for the prefetching fill level. KPC-R monitors a sample of prefetched blocks in the LLC to check if they could have been timely prefetches at the L2 cache. If there are enough timely prefetches detected by KPC-R, the fill level threshold becomes lower allowing prefetches go all the way up to the L2 cache. On the other hand, if the L2 cache is being polluted by a low fill level threshold, KPC-P automatically increases the threshold. Further, when KPC-P sends prefetch requests from the L2 to hit in the LLC with a given confidence, that confidence is used to update placement information withing the LLC's replacement stack. Critically, neither component depends on load PCs, eliminating the hardware complexity of PC propagation through the entire on-chip cache hierarchy. Our analysis shows that KPC outperforms a prior unified memory architecture [11] by 8.1% and best-of-class prefetch aware replacement algorithms [29, 37] by 5% on single core and 8% on multi-core workloads.

The remaining sections are organized as follows. Section 2 discusses the motivation for an integrated cache management without PCs. Section 3 describes the detailed design of KPC system. A performance evaluation is presented in Section 4 and related works are presented in Section 5. Finally, we conclude the paper in Section 6.

2. Motivation

In this section, we discuss the need for holistic cache management and explain why the PC is an inadequate input fea-

[1] Note: DA-AMPM is an extended version of AMPM [10], the 1st Data Prefetching Championship winner.

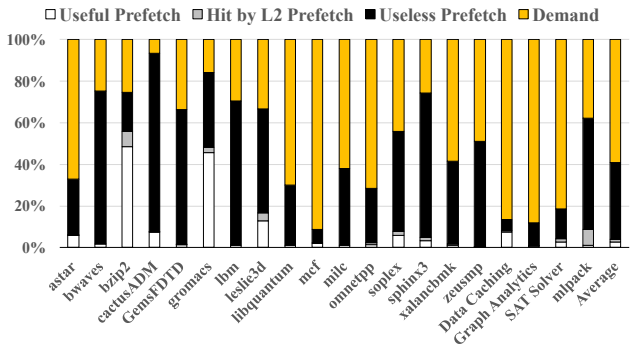


Figure 2: LLC allocation breakdown with DA-AMPM prefetcher.

ture for holistic cache design, especially under the effect of prefetching.

2.1 Why do we need a holistic cache management?

Previous studies [29, 32, 37] show that a large fraction of prefetches are dead in the LLC. Figure 2 analyzes the types of allocations within the LLC when the DA-AMPM prefetcher is in use. More than 40% of LLC allocations are filled by prefetching and approximately 90% of these prefetches are useless, *i.e.*, they will have no accesses in the LLC (they are also pulled into the L2 and all hits to them occur there). Ideally, an intelligent cache replacement policy should detect these dead prefetches and evict them as soon as possible from the LLC. However, as we noted in Figure 1, both PC- and non-PC-based replacement algorithms often yield negative impact when they are combined with prefetching. If we further break down the usage of prefetched blocks, it is even harder for a replacement policy to predict correct reuse behavior by itself. Some prefetched blocks are constantly reused by demand (white) while some blocks are only hit by another prefetch request from the L2 (grey).

In order to minimize the interference from prefetching, Wu *et al.* propose PACMan [37], a prefetch-aware cache management policy. PACMan dedicates a few sets of the LLC to each of the three competing policies that treat demand and prefetch requests differently and uses the policy that shows the lowest number of cache misses. Competition between three different policies, however, increases the overhead of set dueling [27], especially in a multicore environment. Similarly, Seshadri *et al.* propose ICP [29]. ICP is also designed as a comprehensive mechanism to mitigate prefetching interference. ICP simply demotes a prefetched block to the lowest priority on a demand hit based on the observation that most prefetches are dead after their first hit. To address prefetcher-caused cache pollution, it also uses a variation of EAF [28] to track prefetching accuracy and inserts only an accurate prefetch to the higher priority position in the LRU stack. ICP assumes, however, that all prefetches are inserted only into the LLC, which restricts the maximum benefit of prefetching. Additionally, with sophisticated, high-performance data prefetchers [11, 16, 19, 23, 26], demoting a prefetched block on the first hit actually degrades the overall performance. In fact, we found that EAF [28] shows better performance than ICP with lower hardware complexity and storage overhead when it is combined with the DA-AMPM prefetcher. Critically, both PACMan and ICP consider the data prefetcher as an independent component that disturbs the LLC replacement policy and attempts to isolate that disturbance. Neither technique attempts to leverage information from the replacement policy to produce better prefetching algorithm.

Without a holistic approach that identifies how prefetched blocks are used in the L2 and LLC, we cannot optimize the efficiency of the precious on-chip cache resource. A Unified Memory Optimizing (UMO) architecture [11] is the most recent work to attempt holistic cache design. UMO's main idea is to design a data prefetcher that increases the DRAM row buffer locality (DA-AMPM) and a replacement policy that refers to the data prefetcher for better prediction accuracy. However, UMO needs to access the L2 prefetcher on every LLC access since its replacement policy depends on the status map of DA-AMPM. Further, its prefetching algorithm is still separated from the LLC replacement policy and operates as a stand alone module. More importantly, UMO assigns equal priority to a stream of prefetches whose future use distance can be different from each other. In fact, we find that PACMan and EAF achieve higher performance than UMO when they are combined with DA-AMPM, which necessitates a better approach for holistic cache design.

2.2 Why is a PC-based policy insufficient?

One way to implement a holistic approach is to use a PC from the core pipeline for both prefetching and replacement policy. Passing PCs throughout the load-store queue and the all levels of cache hierarchy, however, requires extra logic, wire, and energy consumption. Additionally, there is a significant organizational cost of extra communication between front-end, mid-pipe, cache design, and verification teams as new interfaces are defined, implemented, and tested. When time-to-market is considered, incorporating the PC into prefetching and replacement may be considered too costly by industrial microarchitects. Moreover, modern data prefetchers [11, 23, 26, 30] do not associate prefetches with a particular PC. Thus, when the LLC allocates a cache line brought by a prefetch request, there is no PC value that can be used for reuse distance prediction. Even for demand requests, the PC does not always correlate with reuse behavior.

The baseline assumption of PC-based replacement algorithms is that a given memory instruction will exhibit certain memory use behavior over the program execution, regardless of which particular data location that instruction references. This is not always true, however, since a single load or store instruction might show variable cache line reuse behavior. For instance, if there is a load instruction located in a nested loop, data brought by that load may or may not be reused depending on the result of prior branches. In this sce-

```
for (i=0; i<n; i++) {
  X = load(arr[i]);  // PC Y loads arr[i]
  Z = reuse(X);      // PC Y is reused
}
```

(a) Original loop

```
X1 = load(arr[0]);     // PC Y1 loads arr[0]
...
Xn = load(arr[n-1]);   // PC Yn loads arr[n-1]

Z1 = reuse(X1);        // PC Y1 is reused once
...
Zn = reuse(Xn);        // PC Yn is reused once
```

(b) Unrolled loop

Figure 3: Loop unrolling example.

nario, using a single PC cannot provide a robust prediction. Instead of using a single instruction address, it is possible to accumulate a history of PCs. Although Lai *et al.* [20] introduced and Liu *et al.* [22] later refine a dead block predictor that collects a trace of PCs, using multiple PCs does not improve accuracy in the LLC since most memory accesses are filtered by upper-level caches, causing many important PCs to be missed [17].

2.3 Impact of Compiler Optimizations

Compiler optimizations, such as loop unrolling, also affect the performance of PC-based replacement algorithms. Contrary to the prior case where a single PC exhibits multiple reuse behaviors, loop unrolling generates multiple PCs that often show the same reuse behavior, necessitating a larger prediction table to correlate PCs and reuse. Figure 3a shows a typical example of a loop structure that loads data from an array and reuses it in the same loop iteration. Without loop unrolling, the original loop code will be repeated n times. As a result, a single load instruction (PC Y) will be executed n times and the resulting data (X) will each be reused once. In other words, PC Y is responsible for n data reuses. Prior PC-based replacement algorithms are designed to observe these n instances of data reuse and update a prediction table by increasing the reuse counter associated with PC Y.

With compiler loop unrolling, the actual correlation between PCs and data reuse transforms as shown in Figure 3b. The compiler generates n consecutive load instructions and places them ahead of the reuse function. In doing so, we can reduce the number of branch instructions and achieve more memory level parallelism. Unlike the original code, the unrolled loop contains n load PCs and each PC is associated with a single reuse rather than the n reuses without unrolling. To capture this reuse correlation without any conflicts between PCs, the PC-based replacement algorithm requires at least n entries in its prediction table. Furthermore, training n entries will take n times more iterations through the code as a non-unrolled version. Since loop unrolling is a common optimization technique, PC-based algorithms will suffer from hardware overheads and less prediction accuracy.

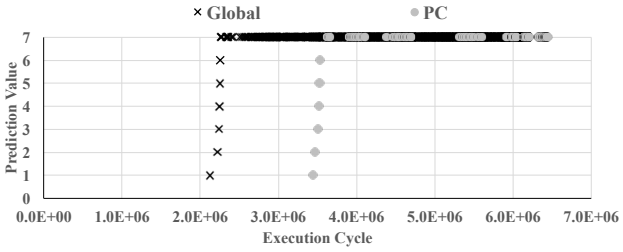
Note that, in the unrolled example, not every PC will be used to access the predictor on every unrolled iteration, since an initial demand read will load multiple words causing subsequent iterations to hit in the L1. Because alignment of data structures is typically not required to be on LLC block boundaries, however, the particular PC that causes a miss will vary from one entry to the unrolled loop to the next.

2.4 The PC can be replaced

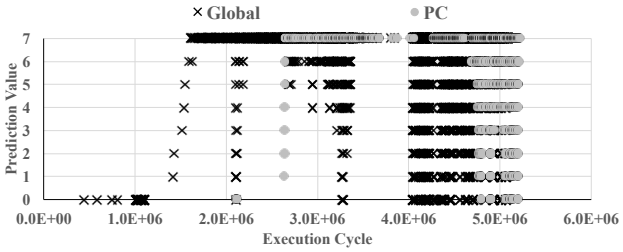
Instead of extracting PCs from the front-end of core pipeline, we propose using a simple global hysteresis mechanism that quickly adapts to the dynamic program phase and provides similar or higher prediction accuracy than PC-based prediction. Figure 4 shows how the prediction counter value changes over the program execution for PC-based prediction and global hysteresis prediction. For PC-based prediction, we profile the most frequently used PC in two SPEC CPU 2006 benchmarks and track a prediction counter value correlated to that PC. In this experiment, we use the prediction mechanism proposed in SHiP [36]. If a cache block is evicted without being reused, the PC that allocated this block increases its prediction counter value. When the counter reaches the maximum value of 7, cache blocks brought by this PC are considered to be dead. Otherwise, if a cache block is hit, the corresponding PC prediction counter decreases. For the global hysteresis experiment, we use a single 3-bit counter that is updated by every LLC hit and miss.

Figure 4a shows the training of the prediction counter in cactusADM. This benchmark has a long streaming access pattern in which all references are effectively dead on arrival. Both prediction techniques eventually saturate their prediction counters and predict most incoming cache blocks to be dead. Some cache blocks with more temporal locality may be preserved to be reused in the LLC. In the figure, we see that the global hysteresis approach learns faster than the PC-based because it is updated on every LLC access while only a subset of references touches any given entry in the PC-based. Figure 4b shows that the prediction counter changes frequently in sphinx3 because the working set size is slightly larger than the size of LLC and there are some LRU friendly blocks. Still, the global hysteresis adapts to the program phase much faster than the PC-based prediction for this workload.

The advantage of fast learning and dynamic adaptation results in better prediction accuracy. Figure 5 shows the prediction accuracy for PC and global hysteresis predictions. As we expected, cactusADM shows similar prediction accuracy for both PC and global hysteresis. For this workload even the slower training of PC-based is sufficient because the streaming access pattern does not change. Alternately, the global hysteresis prediction produces much higher prediction accu-



(a) cactusADM training time



(b) sphinx3 training time

Figure 4: Global hysteresis quickly trains and adapts to program phases.

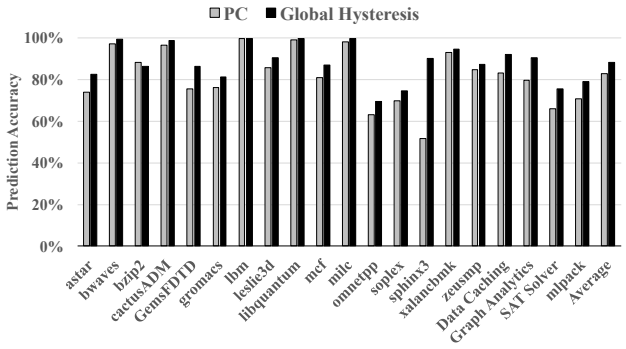


Figure 5: Dead block prediction accuracy for PC and global hysteresis.

racy for sphinx3. As described above, the global mechanism more quickly adapts to the program behavior changes seen in this application.

Why does the simple global hysteresis prediction work? There are two main reasons behind this. First, the global hysteresis makes a dead block prediction only when the counter is saturated. A single cache hit can change the direction of prediction. Thus, there is a low risk of discarding useful cache blocks. Second, data structures with similar temporal locality tend to be accessed in a similar time frame. Typically, when a programmer writes an application, he or she works at the data structure level (*i.e.,* map, list, queue) rather than hardware cache block level. As a result, when a function call accesses a data structure, the cache blocks in that specific structure tend to be accessed at the same time. Thus,

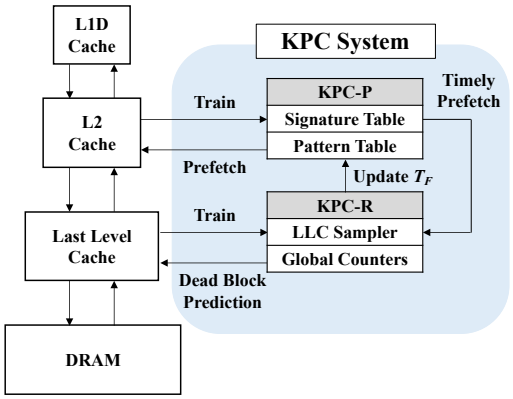


Figure 6: Design overview of the KPC system.

a small global counter can track the reuse behavior of the LLC without complex hardware. In Section 4, we explore several aspects of the global hysteresis prediction and their impact on performance.

3. Design

Here we examine the design of our proposed holistic cache management algorithm, KPC. KPC has two primary components: KPC-P the prefetching component, and KPC-R the cache replacement algorithm. These components are co-designed and integrated to achieve high-performance through the reconstruction of program behavior in the cache hierarchy.

Figure 6 provides a high level design overview of KPC. KPC-P is trained by L2 demand accesses and issues each prefetch with a calculated prediction confidence. The confidence value is used to control prefetch throttling, prefetching level within the cache hierarchy (*i.e.*, L2 or LLC), and prefetch promotion. A prefetch is issued only if its confidence is higher than a set prefetching threshold constant (T_P)[2]. Typically, a prefetch whose predicted use is far in the future is given a low confidence. KPC-R is trained by both prefetch and demand LLC accesses. A small fraction of LLC references are sampled to update the LLC sampler and predictor. KPC-R also dynamically updates the fill level threshold (T_F) based on feedback about prefetch timeliness from KPC-P. Thus, the entire KPC module provides a holistic cache management scheme.

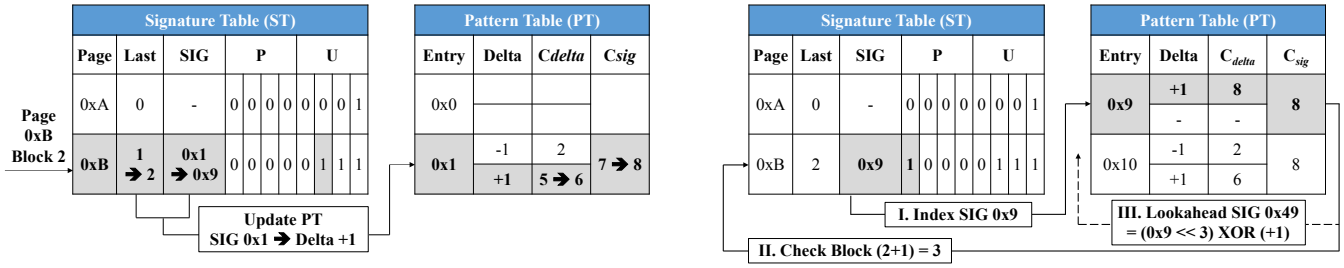
3.1 KPC-P: Confidence-based Prefeching

Next we describe in detail the design and mechanisms of the KPC-P prefetching algorithm.

3.1.1 KPC-P Overview

KPC-P is designed to produce a per-prefetch confidence value that controls the aggressiveness of prefetching. Inspired by prior work on lookahead prefetching [16, 19, 30], KPC-P monitors the pattern of cache block accesses in a

[2] We empirically determined 25% for this threshold was optimal.



(a) Update the signature and delta pattern

(b) Confidence-based prefetching

Figure 7: KPC-P training and prefetching.

physical page, and then recursively prefetches future cache blocks in that page until its prediction confidence falls below a threshold, T_P. To achieve this goal, the Signature Table (ST), shown in Figure 6, records a compressed history of past L1 cache block misses as a history of deltas (*i.e.*, differences) between consecutive memory addresses to the current physical page. This history, including the delta between the last reference in the current page and the current reference in the current page, is used as a signature to index into the Pattern Table (PT) to look up the predicted next delta in the current page. Once this prefetching prediction is made, the PT also generates a "lookahead" signature, corresponding to the predicted next reference in the current page. This lookahead signature is used to re-reference the PT and produce another predicted next delta, and in turn to produce another lookahead signature, and so on.

Figure 7 illustrates the prefetching mechanism of KPC-P when an L2 access occurs to physical page 0xB with an offset of 2 in that page. In the next two subsections we describe the operation of KPC-P using this example.

3.1.2 KPC-P Training

When the L2 cache accesses a block of offset 2 in page 0xB, KPC-P begins by searching for a matching page entry in the ST. Figure 7a shows that the page 0xB was recently accessed, is currently being tracked by the ST, and that the last block offset accessed in this page was 1. Further, we see that the most recent delta history signature in this page (SIG) is 0x1. Thus, the PT is indexed with the signature 0x1. We see that the PT currently has two valid deltas stored at index 0x1, -1 and +1. Because the current reference delta of +1 matches one of these, the +1 delta entry's occurrence counter (C_{delta}) is incremented, adding confidence to the prediction of a signature of 0x1 leading to a delta of +1. The signature occurrence counter (C_{sig}) is also incremented. We will discuss in Section 3.1.3 how these two counters are used to estimate prefetch confidence. Each physical page has its own ST entry, but all pages contribute to building up predictions in a single PT, which is shared by all pages. This accelerates delta pattern learning times.

Based on the current reference, the last block offset and history signature in the ST must also be updated. A new history signature 0x9 is constructed by shifting the prior signature (0x1) to the left 3-bits and XORing it with the current delta (+1). This new value is stored as the new current signature. The last offset into the page (Last) is also updated to 2.

The ST also maintains two bit-vectors to track the status of each cache block within each page. The prefetched (P), and used (U) vectors work together to ensure cache blocks are not redundantly prefetched, and enable the calculation of a general prefetching effectiveness metric, which is used to throttle future prefetches. The used bit is now set, which prevents block 2 from being prefetched again by any future predictions. Finally, we check to see if the prefetch bit corresponding to block offset 2 was previously set. If it was, then prefetching for that block is considered to be timely. The ST resets both prefetch and used bits when a block is evicted from the L2 cache.

3.1.3 KPC-P Prefetching

After updating both tables, KPC-P begins confidence-based prefetching, as illustrated in Figure 7b. First, (**I** on the figure) the PT is indexed by the updated signature 0x9. Any of the deltas in a PT entry can be prefetched, as long as their corresponding confidence, calculated as $C_0 = C_{delta}/C_{sig}$, is above the prefetching threshold T_P. In this example, 0x9 was seen 8 times and each time it was always followed by a +1 delta, giving a confidence of $8/8 = 1$ or 100%. Based on this 100% confidence, KPC-P will plan to prefetch block offset $(2 + 1 = 3)$ within the physical page 0xB.

However, before the actual prefetch request is issued, KPC-P must check the status bit-vectors in the ST to prevent redundant prefetching. If either the corresponding used bit or prefetch bit is already set, KPC-P simply drops the prefetch request. Otherwise, the prefetch is issued and the prefetch bit in the ST is marked to prevent future redundant prefetch requests to that line. Next, KPC-P generates a new, speculative signature based on the demand signature (0x9) and the predicted next cache block delta (+1) (**III** in the figure), creating the first speculative lookahead signature 0x49. KPC-P then continues to recursively index the PT. The lookahead mechanism described here is similar to signature lookahead prefetching [19].

Without a proper throttling mechanism, KPC-P can be too aggressive and pollute the cache through infinite recursion

when individual deltas in the PT have 100% confidence. To prevent this from happening, KPC-P calculates a path confidence according to the following formula:

$$C_n = \alpha * C_{n-1} * C_{delta_n}/C_{sig_n} \quad (1)$$

where n is the current iteration depth and C_{n-1} was the path confidence of the previous iteration. Here we use the global pefetching accuracy α as a scaling factor, preventing infinite recursion on 100% confident prefetches. Note that α is easily calculated by dividing the number of timely prefetches observed at the ST by the number of prefetch requests at the PT. KPC-P uses two 10-bit counters to track these numbers and calculate α. KPC-P also has a small global history register that records a prefetch request that goes beyond the 4KB physical page boundary. Thus, the global history register is able to provide a signature when there is a new page that is not tracked by the ST. For simplicity, the global history register is not shown in Figure 7.

The main advantage of KPC-P is that each prefetch request has a proxy of its future use distance in the form of its path confidence C_n. In general, as the depth of lookahead prefetching increases, the confidence decreases. KPC-P exploits this confidence to support the replacement policy in the LLC. Only when the confidence is higher than the fill level threshold T_F, is a prefetched block also inserted into the L2 (T_F is dynamically adapted by KPC-R as described in Section 3.2.1). Low confidence prefetches with long predicted use distances stay in the LLC waiting for a prefetch request with higher confidence to pull them into the L2. Second, if a prefetch request is a hit in the LLC, the cache block is promoted within the replacement stack of the LLC only when the prefetch confidence is higher than T_F. Otherwise, a prefetch request with low confidence does not change the replacement state. Low confidence indicates two possible scenarios: a long use distance, or an inaccurate prefetch request. In either case, it is best to avoid cache pollution by not filling a low confidence prefetch into the L2. Furthermore, not promoting on a prefetch hit with low confidence ensures that the LLC can evict these blocks earlier than other blocks with higher priority. Thus, KPC-P uses prefetch confidence to integrate prefetching with the replacement policy, and provides a tool for holistic cache design.

3.2 KPC-R: Global Hysteresis Replacement

Here we examine the design and implementation of KPC's global hysteresis replacement algorithm, KPC-R.

3.2.1 KPC-R Overview

KPC-R is a low overhead replacement policy that uses a global hysteresis to predict dead blocks by tracking global reuse behavior. Similar to prior work [28, 36], KPC-R associates the cache recency stack with 2-bit re-reference prediction value (RRPV) counters [14] that represent eviction priority. A cache block with a small RRPV counter has low eviction priority whereas a cache block with maximum RRPV may become the next victim in its set. To train the global hysteresis, KPC-R randomly samples 64 sets in the LLC, and duplicates their tags in a separate structure called the LLC sampler. The sampler is managed using true LRU replacement (unlike the real cache, which uses RRPV counters). When an LLC access is a hit in the sampler, the global hysteresis decreases, indicating that references during this program phase are more likely to be used again. Because the reuse behavior can be different between demand requests and prefetches, KPC-R has two global hysteresis counters, one for each request type. If there is a sampler miss, KPC-R searches for a victim block using the LRU replacement policy within the sampler. If the victim was never used in the sampler, KPC-R increments the global hysteresis based on its allocation type. Note that this global hysteresis is a per-core counter and not shared by different cores on the same chip.

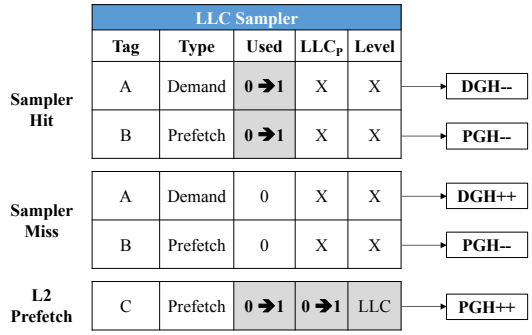


Figure 8: KPC-R global hysteresis update mechanism.

3.2.2 KPC-R Training

Figure 8 shows the update algorithm of KPC-R. In the figure there are three different training scenarios: Sampler Hit, Sampler Miss and L2 prefetch. The first two cases are straightforward. If there is a hit in the sampler, KPC-R marks the used bit for that cache block and decrements the Demand Global Hysteresis (DGH) or Prefetch Global Hysteresis (PGH) based on its allocation type, indicating greater likelihood that references during this phase will be reused. Once the used bit is set, additional hits to the same cache block do not decrease the DGH.

If there is a miss in the sampler, KPC-R replaces the LRU victim in the sampler. As shown in the figure, if the used bit is set, the victim does not update either the DGH or PGH. Alternately, if the victim is not accessed by either demand or prefetch request, KPC-R increases the global hysteresis value, indicating greater likelihood of references being dead during this program phase.

When there is an L2 prefetch hit in the sampler, KPC-R checks whether this block was allocated by an LLC prefetch through the 1-bit "Level" status in the sampler. If the hit block was filled with an LLC prefetch, KPC-R marks the used bit and LLC prefetch bit in the sampler. Remembering an LLC prefetched block in the sampler allows KPC-R to dynamically update the fill level threshold T_F. Figure 9 shows how KPC-R updates T_F based on timely prefetch

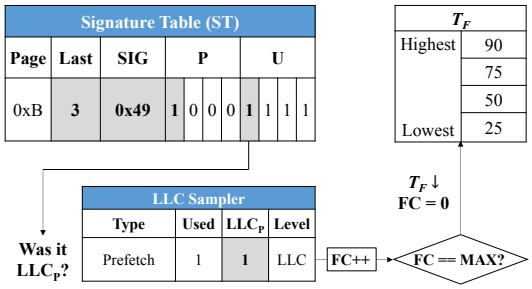


Figure 9: KPC-R updates fill level threshold for KPC-P

	Demand	Prefetch
Promotion	Always Promote	if ($C_n > T_F$) =¿ Promote
Insertion	if (DGH == MAX) =¿ Dead	if (PGH == MAX) =¿ Dead
Fill Level	Always L2 and LLC	if ($C_n < T_F$) =¿ LLC

Table 1: KPC-R prediction table.

feedback from KPC-P. Because KPC-P has prefetch and used bits in the ST, it can detect timely prefetched cache blocks. When a timely prefetch is detected by the ST, KPC-P probes the LLC sampler to see if this block was initially prefetched into the LLC and brought up to the L2 later by an additional prefetch request. If so, the additional L2 prefetch request would have been unnecessary if the prefetch fill level was initially set to the L2. Whenever this event is detected by sampler, KPC-R increments the fill level counter (FC) by one. If the FC becomes saturated, then the fill level threshold T_F decreases to allow more prefetches to be filled into the L2 cache.

To avoid T_F becoming too low and KPC-P polluting the L2 cache, we track the global prefetching accuracy α to increase the fill level threshold. A low α value implies that KPC-P is likely to pollute the L2 cache with aggressive prefetching. Therefore, KPC-P increases T_F by one level when α becomes lower than T_F. Thus, KPC-R helps the data prefetcher by setting a dynamic prefetching level and provides another tool for holistic cache design. Table 1 summarizes the prediction mechanism of KPC-R.

3.2.3 KPC-R Placement/Replacement

Based on the insertion and promotion policy described in Table 1, KPC-R predicts an incoming demand (or prefetch) block to be dead when the DGH (or PGH) is saturated. Dead blocks are inserted to the "LRU" position (RRPV = 3) and do not change other cache blocks' RRPV status. If the global hysteresis is not saturated, the incoming block is inserted to the "near LRU" position (RRPV = 2). Demand requests are always inserted into both L2 and LLC, while a prefetch's fill level is determined by its confidence value. If there is a cache hit, the promotion scheme is based on its access type and confidence value. For demand requests, KPC-R always promotes reused block to the "MRU" position (RRPV = 0). For prefetch requests, KPC-R promotes a reused block only when its prediction confidence is higher than the current fill level threshold ($C_n > T_F$), as described in Section 3.1.3.

Core Parameters	1-4 Cores, 3.2 GHz 256 entry ROB, 4-wide 64 entry scheduler 64 entry load buffer
Branch Predictor	16K entry gshare 20 cycle mispredict penalty
Private L1 Dcache	32KB, 8-way, 4 cycles 8 MSHRs, LRU
Private L2 Cache	256KB, 8-way, 11 cycles 16 MSHRs, LRU, Non-inclusive
Shared L3 (LLC)	2MB/core, 16-way, 34 cycles, LRU, Non-inclusive
Main Memory	1-2 64-bit channels 2 ranks/channel, 8 banks/rank 1600 MT/s

Table 2: Simulator parameters.

4. Evaluation

In this section, we evaluate the KPC system. First, we present the evaluation methodology and compare the performance of KPC with recently proposed replacement algorithms and data prefetchers. We also show in-depth analysis on prefetching coverage with a sensitivity study.

4.1 Methodology

We compare KPC with various prefetching and replacement algorithms using the ChampSim simulator, an extended version of the simulation infrastructure used in the 2nd Data Prefetching Championship (DPC-2) [25]. The detailed simulation parameters can be found in Table 2. We collect SimPoint [24] traces from 16 memory intensive SPEC CPU2006 [1] applications, 3 server workloads from CloudSuite [7], and one machine learning workload trace from mlpack [3] that does collaborative filtering on real world data sets [8]. Since our SimPoint methodology does not work with the server workloads (CloudSuite and mlpack), we instead collect the server workload traces after fast-forwarding at least 30B instructions to get past the benchmark's initialization phase. For all experiments, each trace is warmed up with 200M instructions and simulation results are collected over the next 1B instructions. The baseline replacement policy is LRU replacement for all caches unless otherwise stated. We compare against two prefetch-aware replacement policies, UMO [11] and PACMan [37]. We also compare against two prefetch non-aware replacement policies, EAF [28] and SHiP [36]. Since the original SHiP algorithm does not work well with a data prefetcher as it requires a PC for every insertion, we implement a modified version of SHiP (SHiP+) that uses a single PC value to represent all prefetch requests.

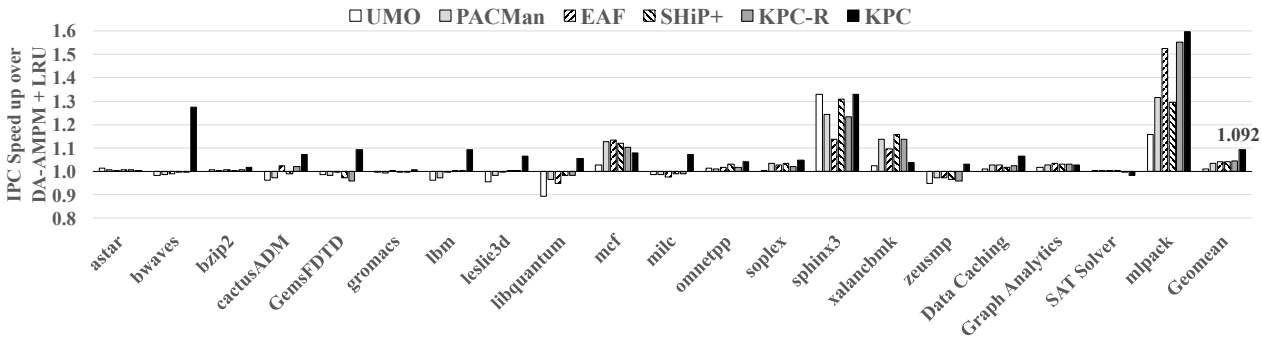


Figure 10: Single core performance compared to DA-AMPM + LRU

4.2 Performance

Single-core Performance: Figure 10 shows the single-core IPC speedup. All results are normalized to the baseline configuration, where the DA-AMPM prefetcher [11] is used with LRU replacement in the caches. In general, KPC has similar or better performance versus the other cache management schemes. The geometric mean improvement of KPC is 9.2%, 5% higher than the best prior work scheme, the optimized PC-based replacement policy (SHiP+) [36], 5.8% higher than the prior work on prefetch aware mechanism (PACMan) [37], and 8.1% higher than previously proposed unified memory architecture (UMO) [11]. We also plot the performance of our replacement scheme KPC-R combined with the prior work DA-AMPM. Since KPC-R is not co-designed for operation with DA-AMPM, we see that the performance of this combination is only marginally better than SHiP+.

Non-PC-based algorithms such as EAF and KPC-R show less performance improvement for mcf, sphinx3, and xalancbmk. These benchmarks exhibit a large number of LLC misses; the reuse behavior of missing cache blocks in these apps is correlated to a large set of various PCs. In this case, each PC value serves as a unique identifier for reuse prediction and shows higher performance gain than non-PC-based schemes. Still, EAF and KPC-R show reasonable performance on these benchmarks. When both prefetching and replacement policy are integrated into one holistic system (KPC), mcf and xalancbmk experience more performance degradation. In this case, both benchmarks benefit from DA-AMPM's aggressive prefetching, which brings more cache lines, achieving more coverage with lower utilization. When confidence is low, KPC dynamically throttles down its lookahead prefetching and has fewer timely prefetches leading to lower performance gains. Aggressive prefetching does not hurt overall performance in a single core environment since only one core is exclusively using the LLC. However, when there are multiple applications competing for the shared resource [4], prior cache management schemes fail to achieve good performance. The multi-core performance of KPC in the next section clearly shows the benefit of dynamic throttling.

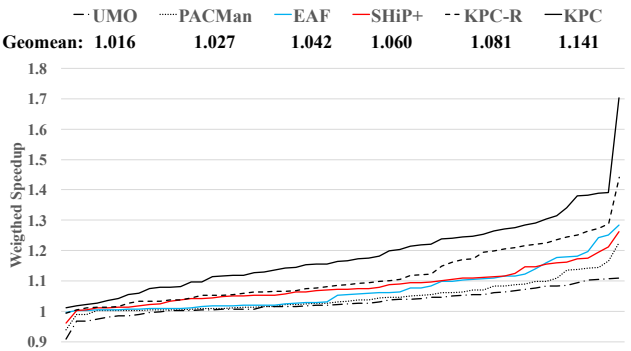


Figure 11: 4-core multiprogrammed workloads performance.

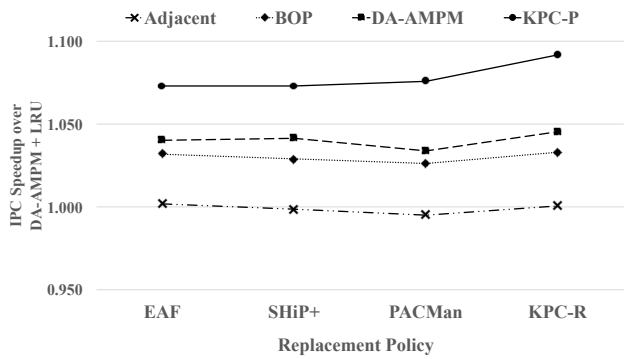


Figure 12: Various combinations of prefetching and replacement algorithms.

Multi-core Performance: Since other schemes have no mechanism to regulate the aggressiveness of the prefetcher based on use and replacement, KPC shows superior performance improvement in a multi-core environment. For multi-core experiments, we randomly generate 54 mixes of 4-core workloads and assign each workload to a different core. Figure 11 shows the normalized weighted speedup of the six different techniques we tested in this work. The

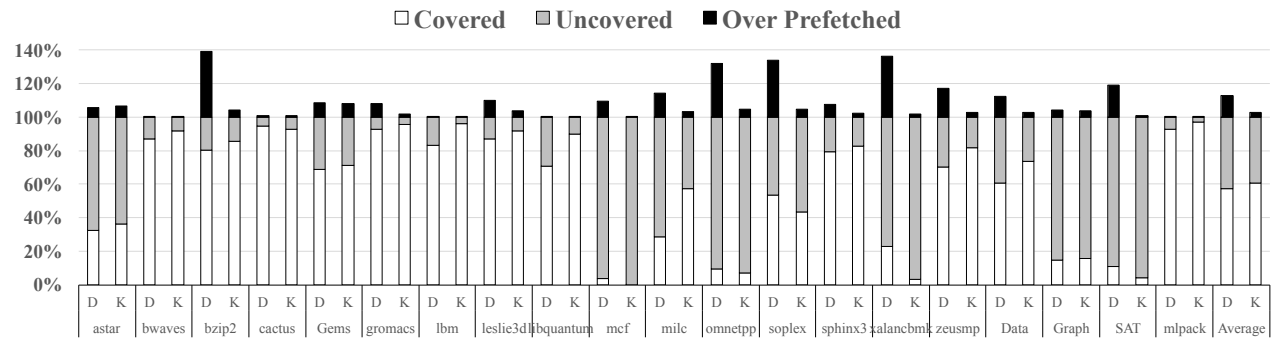


Figure 13: Prefetching coverage: DA-AMPM vs. KPC-P.

graph is sorted by weighted speedup order. On average, KPC achieves a 14.1% speedup and outperforms SHiP+ by 8.1%. Out of the 54 mixes, KPC shows best performance on 52 mixes. For the remaining mixes, SHiP+ or EAF beats KPC by less than 2%. The substantial performance gap between KPC and other techniques is mainly due to the cooperation between KPC-P and KPC-R described in Sections 3.1 and 3.2.

KPC-R with other data prefetchers: To further validate the synergy between KPC-R and KPC-P, we also evaluate different types of data prefetchers with various replacement policies. Figure 12 plots the performance of various combinations between data prefetchers and replacement policies. Here, along with KPC-P and DA-AMPM, we also evaluate an adjacent cache line prefetcher (Adjacent) [9] and Best-Offset Prefetcher (BOP) [23] (BOP was the winner of the 2nd Data Prefetching Competition (DPC-2) [25]). Both prefetchers are offset-based spatial prefetchers which exhibit different characteristic compared to streaming based prefetchers (*e.g.,* DA-AMPM and KPC-P). The performance is normalized to DA-AMPM with LRU. Since UMO shows the least improvement, we have removed it from this figure. Figure 12 shows that KPC-R still achieves great performance versus other techniques regardless of prefetching algorithms. In particular, KPC-R achieves the best performance when it is combined with KPC-P. KPC-R adds 3.8% performance on the top of KPC-P while PACMan shows the second best performance of 2.3% above KPC-P with LRU. To summarize, the global hysteresis replacement scheme works well for both spatial and streaming based prefetchers, however, its gain is maximized when KPC-R is used with KPC-P due to the synergistic effect of holistic cache management.

4.3 Analysis

Prefetching Coverage: Figure 13 shows prefetching coverage of DA-AMPM and KPC-P. Each prefetcher is labeled by its first letter. On average, KPC-P covers 61% of on-chip cache misses with 3% over prefetched blocks while DA-AMPM covers 59% misses with 13% over prefetches. Over prefetches represent the sum of prefetches never used prior to eviction, caused by aggressive prefetching. Compared to KPC-P, DA-AMPM produces 10% more over prefetched blocks. In particular, we can see that DA-AMPM generates a higher fraction of over prefetching for mcf, xalancbmk, and SAT solver, where DA-AMPM achieves greater performance. Note that for these benchmarks, the actual number of prefetches is also higher for DA-AMPM. While this does provide some gain for these benchmarks, the over prefetching becomes a serious issue when multiple applications contend for shared resources such as LLC capacity and memory bandwidth, as shown in Figure 11. Since KPC is designed to adapt prefetch placement and aggressiveness by closely integrating the prefetcher and replacement policy, KPC exhibits less performance gain for single-core but achieves much higher benefit in a multi-core environment.

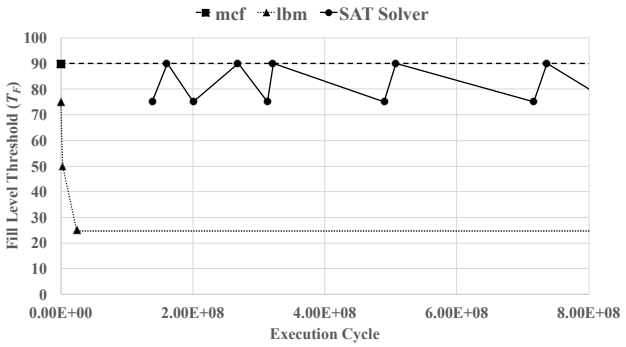


Figure 14: Dynamic adaptation of fill level thrshold T_F.

Dynamic Fill Level Thrshold: Figure 14 shows the dynamic adaptation of the fill level threshold T_F over program execution. As we see, each benchmark prefers a different T_F value. For example, KPC lowers T_F for lbm since most of prefetches are useful in the L2 cache. The timely prefetch feedback from KPC-P to KPC-R quickly adjusts the T_F value. On the other hand, KPC does not change T_F for mcf since its reference pattern is unpredictable. For SAT Solver, T_F frequently changes due to program phase, based on its global prefetching accuracy. Thus, KPC dynamically adapts its fill level threshold to each individual application.

Structure	Entry	Component	Storage
Signature Table	256	Valid (1 bit) Tag (16 bit) Last offset (6 bit) Signature (12 bit) Prefetch (64 bit) Used (64 bit) LRU (8 bit)	43776 bits
Pattern Table	512	C_{sig} (4 bit) C_{delta} (4*4 bit) Delta (4*7 bit)	24576 bits
LLC Sampler	64	Valid (1 bit) Tag (16 bit) Type (1 bit) Used (1 bit) LLC_P (1 bit) Level (1 bit) LRU (4 bit)	1600 bits
Hysteresis	1 1	DGH (3 bit) PGH (3 bit)	3 bits 3 bits
Misc.	-	Registers	284 bits
43776 + 24576 + 1600 + 6 + 284 = 70242 bits ≈ 8.57 KB			

Table 3: KPC storage overhead.

Prefetcher	Replacement	Total Storage
DA-AMPM: 4.8 KB	UMO: 20 bits	≈ 4.81 KB
	PACMan: 30 bits	≈ 4.81 KB
	EAF: 32 KB	≈ 36.8 KB
	SHiP: 7.18 KB	≈ 11.98 KB
	KPC-R: 0.23 KB	≈ 5.03 KB
KPC-P: 8.34 KB	KPC-R: 0.23 KB	≈ 8.57 KB

Table 4: Storage overhead comparison.

Global Hysteresis Sensitivity: We also investigated the performance sensitivity of the global hysteresis to its counter bit-width (figure removed for brevity). We changed the width of global hysteresis from 1-bit to 10-bits and measured the geomean performance. Surprisingly, we find that, for single-core, the performance of KPC is very insensitive to the width of the global hysteresis counter. Even for a 1-bit counter, performance drops by only 1.2%. For multi-core, however, we found that using a counter width below 2-bits suffers from substantial performance degradation. Though the global hysteresis is a per-core counter, multiple applications put additional memory pressure on the LLC and frequently toggle the prediction of a 1-bit global hysteresis, causing performance degradation. The final design of KPC uses 3-bit global hysteresis because there is no visible difference in performance above 3 bits.

Storage Overhead: Table 3 shows the storage overhead of KPC. All together, KPC requires a modest 8.57KB of state per core, with only 2% of the overhead coming from KPC-R. Most of the storage overhead lies with KPC-P, since KPC-R only uses a small sampler and two narrow hysteresis counters per core. Table 4 presents a storage comparison between KPC and previous cache management schemes. Note that the storage overhead of KPC is comparable to existing state-of-the-art replacement algorithms while providing higher performance.

Moreover, unlike PC-based schemes, KPC does not require extra logic/wires to propagate PC values through from the instruction fetch unit, to the load store queue, and then the entire cache hierarchy. Both UMO and PACMan rely on set dueling [27] that only requires two or three global counters. In doing so, their storage overheads are very small but come with the cost of low performance improvement. In particular, in multi-core environments, set dueling based policies show less performance gain than others because the learning overhead from set dueling monitors grows as the number of cores increases [37].

5. Related Work

This section reviews some of the most relevant techniques proposed in cache replacement policy and data prefetching.

5.1 Replacement Policy

PC-based replacement: Lai *et al.* [20] first introduced a concept of dead block prediction to prefetch data into dead blocks in the L1 cache. The prediction is made by learning a trace of PCs that leads to the last access for a cache block. If a specific trace is known to generate a dead block, the trace based predictor speculates that other blocks accessed by the same trace is likely to be dead in near future. Khan *et al.* proposed a much simplified but more powerful sampling dead block predictor (SDBP) [17] using a single instance of PC rather than a trace of PCs. Rather than learning every access to the cache, SDBP samples a small number of sets using partial tags. If the last PC that accessed a cache block is known to be dead in the sampled sets, SDBP updates prediction counter. Once the prediction counter goes above certain threshold, then the corresponding block is predicted dead. Wu *et al.* proposed a signature-based hit predictor (SHiP) [36] to predict how soon a block will be re-referenced. SHiP's structure consists of a table of 3-bit saturating counters that are indexed by the PC of the memory instruction that caused the block's fill. Counters are incremented on every hit, and decremented on an unused eviction. The predictions are used to determine the placement position of incoming blocks following the RRIP policy [14]. If the block is predicted to have a far reuse, its placement position will be "distant re-reference interval", otherwise the block will be predicted to have an "intermediate re-reference prediction" and be place accordingly. Two recent replacement policies have been proposed which attempt to approximate Belady's optimal replacement policy, either through sampling-based learning [12] or a perceptron learning approach [34].

Non-PC-based replacement: Qureshi *et al.* [27] proposed a simple bimodal insertion policy (BIP) that places most incoming cache blocks in the LRU position. In this way, BIP protects the cache from thrashing caused by streaming data and results in higher hit rate. In addition, to choose be-

tween bimodal policy and traditional LRU policy, Qureshi *et al.* [27] propose a set dueling mechanism that dedicates few sets of the cache to each of the two competing policies. Later, Jaleel *et al.* [13, 14] further improves the bimodal insertion and set dueling mechanism by predicting the re-reference interval using the RRIP chain, as discussed in Section 3.2.1. Seshadri *et al.* proposed a mechanism that tracks the addresses of recently evicted blocks in a structure called Evicted-Address Filter(EAF) [28]. If an incoming block's address is present in the EAF it is predicted to have high reuse, all other blocks are predicted to have low reuse. These predictions are used to decide the placement positions of blocks. Blocks predicted to have high reuse will be inserted with a high priority, *i.e.,* at the MRU position, blocks predicted to have low reuse will be inserted in a low priority position such as the LRU position, and will be quickly evicted from the cache.

5.2 Data Prefetcher

Srinath *et al.* proposed a feedback directed prefetcher which estimates accuracy and decides where in the LRU stack to insert the prefetched blocks in the cache [32]. However, the feedback directed prefetcher requires additional prefetch bit in the every cache block and its replacement policy does not cover dead blocks brought by demand requests. The main prefetching algorithm of KPC-P is inspired by prior confidence based data prefetchers [16, 19]. B-Fetch [16] introduced a lookahead prefetching mechanism with the support from branch predictor. SPP [19] eliminates the complex hardware support from branch predictor and register file while covering both sequential and complex memory access patterns. Though the confidence mechanism is adopted from prior works [16, 19], the KPC-P uniquely leverages prefetching confidence to control both prefetching and cache placement/replacement.

5.3 Prefetch-aware Replacement

As described in Section 2.1, a handful of techniques exist which attempt to integrate prefetching and replacement into a combined scheme. PACMan [37] is designed to mitigate the degree of prefetch-induced interference. The main idea is to have multiple cache insertion/promotion policies and select the best policy using set dueling [27]. However, as noted in Section 2.1, the learning overhead of set dueling with multiple policies increases as the number of core grows in a processor. Seshadri *et al.* propose ICP [29] for LLC prefetching. ICP always demotes prefetched blocks after their first hit. This is mainly because most prefetches are dead after the first access. However, prefetching every block into the LLC does not maximize the performance of data prefetchers. In addition, to demote a prefetched block after its first hit, ICP requires additional prefetch bit for every cache line in the LLC. Unified Memory Architecture [11] is the most recent work which examines holistic cache design. UMO uses the DA-AMPM prefetcher to increase the DRAM row buffer locality and change the replacement algorithm that exploits the prefetcher's map structure. UMO requires additional access to the L2 data prefetcher on every LLC access since the replacement policy needs to lookup the map table of DA-AMPM at the L2 cache.

To the best of our knowledge, KPC is the first technique to leverage prefetch confidence in its replacement policy and cache level placement. This integrated approach provides significant performance improvement versus all prior techniques.

6. Conclusion

Kill-the-PC reconstructs program behavior in the cache hierarchy by taking a holistic approach to cache management. Tightly integrating L2 prefetching and LLC management into one unified solution, KPC is effective at making sure data is as close to the processor core as possible when it is needed by a demand access. In addition to being an effective prefetcher for complex delta patterns, the prefetching component, KPC-P, feeds confidence information about each individual prefetch to the LLC replacement component, KPC-R. A low confidence prefetch is less likely to interfere with the contents of the LLC, and as confidence in that prefetch increases, its position within the LLC replacement stack is solidified, and it eventually is brought into the L2 cache, close to where it will be used in the processor core. In addition to effectively predicting dead LLC blocks by observing program phase behaviors, KPC-R also gives feedback to KPC-P to help decide on the optimal fill level for prefetches.

While KPC-P and KPC-R can each stand on its own as an effective solution within its own domain, the combination of both is more effective than replacing either component with a state-of-the-art solution. KPC provides a 9.2% performance benefit versus a baseline system with a top-of-class prefetcher, besting the nearest competitor by 5%. Further, across our suite of multicore mixes, KPC improves performance by 14.1% versus a prefetching+LRU baseline and extends its lead over the best of class competitor to 8.1%.

Acknowledgments

We thank the National Science Foundation, which partially supported this work through grants CCF-1320074 and I/UCRC-1439722, and Intel Corp. for their generous support.

References

[1] Standard Performance Evaluation Corporation CPU2006 Benchmark Suite. `http://www.spec.org/cpu2006/`.

[2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 176–186. IEEE, 1991.

[3] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[4] N. D. Enright Jerger, E. L. Hill, and M. H. Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 177–188, 2006.

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer*

Architecture (ISCA), 2011 38th Annual International Symposium on, pages 365–376. IEEE, 2011.

[6] V. V. Fedorov, S. Qiu, A. L. Reddy, and P. V. Gratz. Ari: Adaptive llc-memory traffic management. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):46, 2013.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, volume 47, pages 37–48. ACM, 2012.

[8] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 5(4):19, 2016.

[9] R. Hegde. Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers. *Intel Software Network*, 2008.

[10] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, 13:1–24, 2011.

[11] Y. Ishii, M. Inaba, and K. Hiraki. Unified memory optimizing architecture: memory subsystem control with a unified predictor. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 267–278. ACM, 2012.

[12] A. Jain and C. Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 78–89. IEEE, 2016.

[13] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219. ACM, 2008.

[14] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[15] D. A. Jiménez. Insertion and promotion for tree-based pseudolru last-level caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 284–296. ACM, 2013.

[16] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jiménez. B-fetch: Branch prediction directed prefetching for chip-multiprocessors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 623–634. IEEE Computer Society, 2014.

[17] S. Khan, Y. Tian, and D. A. Jiménez. Sampling dead block prediction for last-level caches. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 175–186. IEEE, 2010.

[18] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jiménez. Improving cache performance by exploiting read-write disparity. In *Proceedings of the 20th Internatial Symposiym on High Performance Computer Architecture (HPCA)*, pages 452–463. IEEE, 2014.

[19] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *Microarchitecture (MICRO), 2016 49rd Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[20] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pages 144–154. IEEE, 2001.

[21] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 53. ACM, 2015.

[22] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Los Alamitos, CA, USA, 2008. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/MICRO.2008.4771793.

[23] P. Michaud. A best-offset prefetcher. In *High Performance Computer Architecture (HPCA), 2016 IEEE 20th International Symposium on*. IEEE, 2016.

[24] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 318–319. ACM, 2003.

[25] S. H. Pugsley, A. R. Alameldeen, C. Wilkerson, and H. Kim. The 2nd Data Prefetching Championship (DPC-2). `http://comparch-conf.gatech.edu/dpc2/`.

[26] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 626–637. IEEE, 2014.

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[28] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 355–366. ACM, 2012.

[29] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):51, 2015.

[30] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[31] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 252–263. IEEE Computer Society, 2006.

[32] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 63–74. IEEE, 2007.

[33] E. Teran, Y. Tian, Z. Wang, and D. A. Jiménez. Minimal disturbance placement and promotion. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 201–211. IEEE, 2016.

[34] E. Teran, Z. Wang, and D. A. Jiménez. Perceptron learning for reuse prediction. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[35] J.-Y. Won, P. Gratz, S. Shakkottai, and J. Hu. Having your cake and eating it too: Energy savings without performance loss through resource sharing driven power management. In *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pages 255–260. IEEE, 2015.

[36] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441. ACM, 2011.

[37] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer. Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453. ACM, 2011.

[38] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comp. Arch. News*, 23:20–24, March 1995. ISSN 0163-5964.