Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks

VIVEK SESHADRI, SAMIHAN YEDKAR, HONGYI XIN, and ONUR MUTLU,
Carnegie Mellon University
PHILLIP B. GIBBONS and MICHAEL A. KOZUCH, Intel Pittsburgh
TODD C. MOWRY, Carnegie Mellon University

Many modern high-performance processors prefetch blocks into the on-chip cache. Prefetched blocks can potentially pollute the cache by evicting more useful blocks. In this work, we observe that both accurate and inaccurate prefetches lead to cache pollution, and propose a comprehensive mechanism to mitigate prefetcher-caused cache pollution.

First, we observe that over 95% of useful prefetches in a wide variety of applications are not reused after the first demand hit (in secondary caches). Based on this observation, our first mechanism simply demotes a prefetched block to the lowest priority on a demand hit. Second, to address pollution caused by inaccurate prefetches, we propose a self-tuning prefetch accuracy predictor to predict if a prefetch is accurate or inaccurate. Only predicted-accurate prefetches are inserted into the cache with a high priority.

Evaluations show that our final mechanism, which combines these two ideas, significantly improves performance compared to both the baseline LRU policy and two state-of-the-art approaches to mitigating prefetcher-caused cache pollution (up to 49%, and 6% on average for 157 two-core multiprogrammed workloads). The performance improvement is consistent across a wide variety of system configurations.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles

General Terms: Design, Performance, Memory

Additional Key Words and Phrases: Prefetching, caches, cache pollution, cache insertion/promotion policy

ACM Reference Format:
Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Mitigating prefetcher–caused pollution using informed caching policies for prefetched blocks. ACM Trans. Architec. Code Optim. 11, 4, Article 51 (January 2015), 22 pages.
DOI: http://dx.doi.org/10.1145/2677956

1. INTRODUCTION

Hardware caching and prefetching are two techniques typically employed by modern high-performance processors to mitigate the impact of long memory access latency. In many such processors [Intel 2006; Oracle 2011; Kalla et al. 2010], the hardware prefetcher prefetches blocks into the on-chip cache. Doing so has three benefits compared to prefetching data into a separate prefetch buffer: (1) it avoids statically partitioning the data storage between the cache and the prefetch buffer—prior work [Srinath

Authors' addresses: V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, and T. C. Mowry, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213; emails: vseshadr@cs.cmu.edu, samihanyedkar@gmail.com, hxin@cs.cmu.edu, onur@cmu.edu, tcm@cs.cmu.edu; P. B. Gibbons and M. A. Kozuch, Intel Pittsburgh, Collaborative Innovation Center, 4720 Forbes Avenue, Pittsburgh PA 15213; emails: {phillip.b.gibbons, michael.a.kozuch}@intel.com.

DOI: http://dx.doi.org/10.1145/2677956

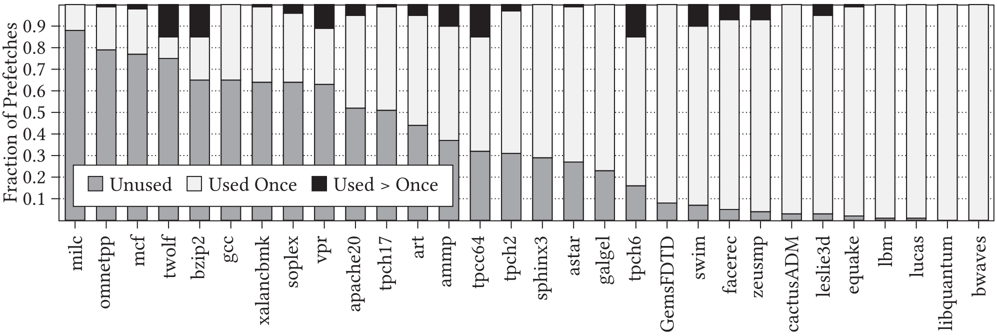


Fig. 1. Usage distribution of prefetched blocks.

et al. 2007] has shown that a large prefetch buffer (≈64KB) is required to achieve reasonably good performance with an aggressive prefetcher; (2) it simplifies on-chip cache coherence since coherence requests have to snoop only the cache; and (3) it avoids the need to design a separate prefetch buffer as most processors have a cache anyway.

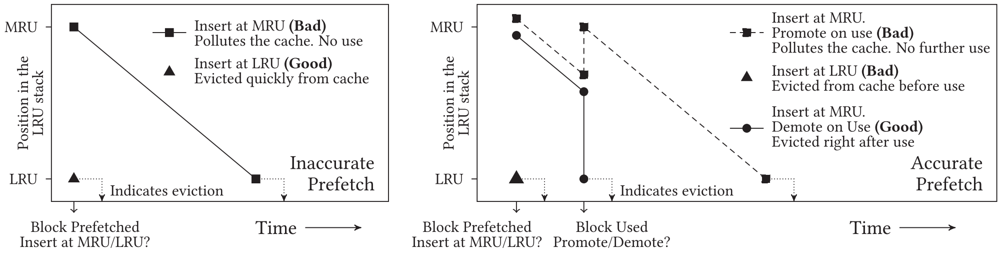
One major limitation of prefetching data into the on-chip cache is that prefetched blocks can potentially evict more useful blocks from the cache, increasing the number of cache misses. This problem is referred to as *prefetcher-caused cache pollution*. Prefetcher-caused cache pollution can degrade performance significantly, especially in a multicore system, in which prefetched blocks of one application evict useful blocks of another application. Several prior works (e.g., Srinath et al. [2007], Wu et al. [2011], Zhuang and Lee [2003], and Lin et al. [2001b]) aim to address this problem. In this work, we propose a new and simple solution to address prefetcher-caused cache pollution.

Prefetched blocks fall into one of two categories: *inaccurate prefetches* – those that are not used later by the application, and *accurate prefetches* – those that are used by the application. We first observe that a significant majority of accurate prefetches are *used only once* by the application. This observation is valid only for secondary caches (L2, L3, etc.) and not for the primary L1 caches, in which multiple demand requests may access different words within a prefetched block. The main intuition behind why the majority of the accurate prefetched blocks are used only once is that prefetching typically works well for large data structures that do not fit into the cache. Blocks of such data structures have a large reuse distance, and thus are unlikely to get used more than once while in the cache.

To confirm this observation, we conducted a study in which we ran each application on a system with a 1MB L3 cache. A multi-entry stream prefetcher [Le et al. 2007] analyzes the misses from the L2 cache and prefetches data into the LLC (Section 5 provides more details of our methodology). Figure 1 plots the fraction of prefetched blocks that are (1) unused, (2) used once, or (3) used more than once by the application. As Figure 1 indicates, a majority of the prefetched blocks are either unused or used only once. In fact, over 95% of the accurate prefetches are used only once.[1]

The traditional LRU policy is suboptimal for both unused and used-once prefetched blocks. Figure 2 demonstrates the shortcomings of the LRU policy and indicates the good policies that reduce pollution caused by inaccurate and accurate prefetches. The x-axis indicates time and the y-axis indicates the position of a prefetched block

[1] Figure 1 does not include blocks that are not prefetched. The aim of this study is *not* to show that prefetched blocks are less likely to be reused than demand-fetched blocks. Rather, it is to show that prefetched blocks are less likely to be used more than once. While there may be other demand-fetched blocks that do not exhibit any reuse, our mechanism does not aim to mitigate pollution caused by such nonprefetched blocks. Our results in Section 7 show that just mitigating pollution caused by prefetched blocks can significantly improve performance.



(a) Inaccurate Prefetch: No demand use

(b) Accurate Prefetch: Likely only one use

Fig. 2. Effect of different policies on the cache occupancy of (a) an inaccurate and (b) an accurate prefetch. Each line corresponds to a specific insertion/promotion policy followed for prefetched blocks.

in the LRU stack. Each line in Figures 2(a) and 2(b) correspond to a specific policy followed for a prefetched block. There are three points to note regarding the figure.

First, inserting an inaccurate prefetch at the MRU position is bad, as the block unnecessarily pollutes the cache without any use (—■— Figure 2(a)). Second, the LRU policy inserts an accurate prefetch at the MRU position and promotes the block on its first use (- ■ - Figure 2(b)). This is bad, as our observation indicates that such accurate prefetches will likely not be used more than once. Finally, prior work [Srinath et al. 2007] has proposed to insert all prefetches at the LRU position when prefetches are determined to cause pollution. However, inserting an accurate prefetch at the LRU position is bad since it can cause the block to be evicted even before it is used by the application, thereby rendering the prefetch useless (▲ Figure 2(b)).

In this work, we propose a mechanism to comprehensively address pollution due to both accurate prefetches and inaccurate prefetches without losing the benefits of prefetching. Our proposed mechanism, *Informed Caching policies for Prefetched blocks (ICP)* has two component policies.

First, based on our observation that over 95% of the useful prefetches are not reused after the first demand hit, the first component simply demotes a prefetched block to the lowest priority when it is used by a demand request. This ensures that the prefetched block does not pollute the cache after its single use (—●— Figure 2(b)). We refer to this mechanism as ICP-Demotion (ICP-D). This mechanism only requires a single bit per tag entry to track whether the corresponding block is prefetched, and it is very effective in mitigating pollution caused by accurate prefetches.

Second, to address the pollution caused by inaccurate prefetches, our mechanism predicts the accuracy of each prefetched block and inserts predicted-inaccurate prefetches with a low priority into the cache.[2] As such, we take an approach similar to prior work [Lin et al. 2001b; Zhuang and Lee 2003; Srinath et al. 2007]. However, we observe that the metric used by prior works to estimate the accuracy of a prefetcher, *fraction of useful prefetches generated by the prefetcher*, can potentially lead to a positive feedback loop. Inserting all predicted-inaccurate prefetches with a low priority into the cache will make it less likely for the prefetcher to generate useful prefetches. This can cause the prefetcher to get stuck in a state in which all its prefetches are classified as inaccurate, even though they may actually be useful. To address this problem, we propose a Self-Tuning Accuracy Predictor, the second component of our mechanism. We refer to this mechanism as ICP-Accuracy Prediction (ICP-AP). To detect cases when an accurate prefetch is falsely classified as inaccurate, ICP-AP tracks the addresses of a small set of recently evicted prefetched blocks in a structure called *Evicted-Prefetch*

[2] Note that while inaccurate prefetches can be dropped altogether to reduce memory bandwidth consumption [Lee et al. 2008], we do not drop prefetches, as the focus of this work is on mitigating cache pollution caused by prefetches.

Filter (EPF). On a demand miss, if the missed block address is present in the EPF, it indicates a case of a misclassified prefetch, and ICP-AP increases its estimate of the accuracy of the prefetches generated by the prefetcher.

Note that prior works [Wu et al. 2011; Srinath et al. 2007; Lin et al. 2001b; Zhuang and Lee 2003] have proposed mechanisms to address prefetcher-caused cache pollution. We perform a detailed qualitative (Section 4) and quantitative (Sections 6 and 7) comparison of ICP to these prior works.

Our analysis in Section 6 shows that both ICP-D and ICP-AP are effective in reducing the pollution caused by prefetched blocks (i.e., reducing the time for which prefetches stay in the cache) without losing the benefit of prefetching (i.e., without increasing the cache miss rate). Across 157 two-core multiprogrammed workloads, ICP improves system performance by as much as 49% (6% on average) compared to the baseline and by as much as 43% (5% on average) compared to two state-of-the-art approaches to mitigate prefetcher-caused pollution (FDP [Srinath et al. 2007] and PACMan [Wu et al. 2011]).

We make the following contributions.

—We show that promoting prefetched blocks when they receive a demand hit leads to cache pollution in a majority of cases. To address this problem, we propose a simple mechanism that demotes a prefetched block on a demand hit.

—We observe that prior mechanisms to predict prefetch accuracy can get stuck in a positive feedback loop—a prefetcher classified as inaccurate continues to get classified as inaccurate. We propose a self-tuning accuracy predictor to address this problem.

—We compare our proposed mechanism with the baseline LRU policy and two state-of-the-art approaches (FDP [Srinath et al. 2007] and PACMan [Wu et al. 2011]) on a wide variety of workloads and system configurations (e.g., varying cache sizes, memory latency, core counts, and cache replacement policies). Our evaluations show that ICP consistently outperforms both the baseline and prior approaches.

2. ICP: MECHANISM

Our proposed mechanism to mitigate prefetcher-caused cache pollution consists of two components. The first component addresses the problem of unnecessary promotion of prefetched blocks, thereby mitigating pollution caused by accurate prefetches. The second component addresses the pollution caused by inaccurate prefetches. In this section, we describe each component in detail.

2.1. ICP-Demotion (ICP-D)

The first component of our mechanism aims to address pollution caused by accurate prefetches. Our analysis of the prefetch-usage distribution (Figure 1) showed that in a system in which the prefetcher prefetches blocks into the last-level cache (LLC), over 95% of the accurate prefetches are used only once in the LLC. Based on this observation, our mechanism, ICP-Demotion (ICP-D), tracks blocks that were prefetched by the hardware prefetcher and demotes a prefetched block to the lowest priority when it receives a demand hit. Thereby, ICP-D prevents prefetched blocks from polluting the cache unnecessarily after they are used by the application.

ICP-D makes an implicit prediction that a prefetched block will not be reused after the first demand hit. Our evaluations show that this prediction is accurate for the majority (95%) of the cases. However, depending on the available cache space and working set of the application, some prefetchable data structure that is reused by the application may fit into the cache. In such cases, it may be useful to retain the prefetched blocks of such data structures so that subsequent accesses to the data structure do not have to be (pre)fetched from memory. To this end, we explored the possibility of using a reuse predictor to explicitly predict the reuse behavior of a prefetched block when it receives a demand hit. On a demand hit to a prefetched block, the cache consults the reuse predictor [Seshadri et al. 2012]. If the block is predicted to have high reuse, it is promoted to a high priority. Otherwise, it is demoted to the lowest priority. However, our evaluations showed that the reuse predictor did not provide significant benefit on top of ICP-D's implicit prediction.

2.2. ICP-Accuracy Prediction (ICP-AP)

The second component of our proposed mechanism aims to mitigate pollution caused by inaccurate prefetches. Our approach, similar to another prior work [Zhuang and Lee 2003], is to predict the accuracy of each individual prefetched request and use the prediction to determine the policy for the prefetch request. Specifically, our mechanism inserts only predicted-accurate prefetched blocks with a high priority and inserts predicted-inaccurate prefetched blocks with a low priority.

To measure the accuracy of a prefetcher, many prior works (e.g., Srinath et al. [2007], Zhuang and Lee [2003], Ebrahimi et al. [2009b, 2011], Dahlgren et al. [1995], Nesbit et al. [2004], and Lee et al. [2008]) use the fraction of useful prefetches generated by the prefetcher. At a high level, the mechanism uses two counters: one to track the total number of prefetches (`total`) and another to track the number of useful prefetches (`used`). When a block is prefetched into the cache, the `total` counter is incremented. When a prefetched block receives a demand hit, the `used` counter is incremented. If the ratio of `used` over `total` is above a threshold, the prefetcher is classified as accurate. Otherwise, it is classified as inaccurate.

Using this metric to determine the accuracy of prefetched blocks has a shortcoming. When a prefetcher is classified as inaccurate, any block prefetched is inserted with a low priority. As a result, if the prefetcher was misclassified as inaccurate, the prefetched block may get evicted from the cache even before the demand request arrives. This reduces the likelihood of an increase in the prefetcher's `used` counter, which in turn will cause the prefetcher to be classified as inaccurate.

To address this problem, we propose a self-tuning accuracy predictor, ICP-Accuracy Prediction (ICP-AP). The key idea behind ICP-AP is to detect cases when an accurate prefetch is misclassified as inaccurate, and increase the estimate of the accuracy of the prefetcher in such cases. To detect such cases of misclassification, ICP-AP tracks the addresses of a small set of recently evicted prefetched blocks that were predicted to be inaccurate in a structure called *Evicted-Prefetch Filter* (EPF).[3] On a demand miss, if the missed block address is present in the EPF, it indicates that an accurate prefetch was misclassified as inaccurate.

ICP-AP works as follows. It augments the prefetcher with a saturating counter that measures the accuracy of the prefetcher. If the counter value is above a threshold, the prefetcher is classified as accurate. Otherwise, it is classified as inaccurate. ICP-AP augments the cache with an EPF that tracks recently evicted predicted-inaccurate prefetched blocks. The accuracy counter is incremented in two cases: (1) when a demand request hits on a prefetched block, and (2) when the address of a demand miss is present in the EPF. Both cases indicate that the prefetcher generated an accurate prefetch. Similarly, the accuracy counter is decremented in two cases: (1) when a prefetched block address is evicted from the EPF, and (2) when a prefetched block predicted to be accurate is evicted from the cache. Both cases indicate that the prefetcher generated a likely inaccurate prefetch.

When a prefetch is generated, ICP-AP uses the accuracy counter to determine the accuracy of the prefetch. If the prefetch is predicted to be accurate, the prefetch is

[3] The Evicted-Prefetch Filter is similar to the Evicted-Address Filter [Seshadri et al. 2012], which keeps track of addresses of recently evicted demand-fetched blocks to predict the reuse behavior of future accesses.

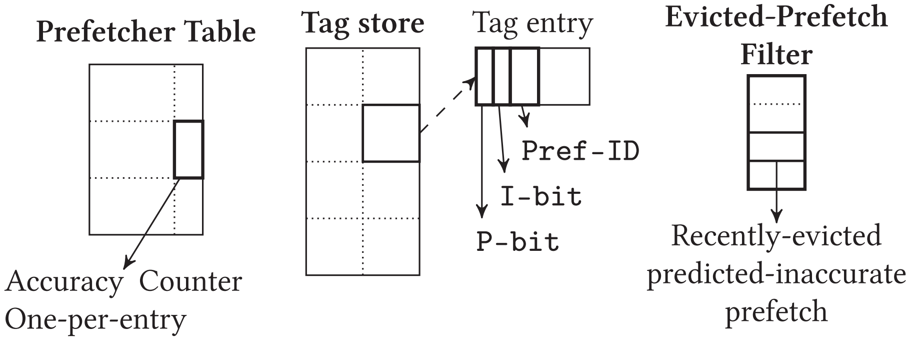


Fig. 3. Implementation of ICP. Thick lines indicate changes introduced by ICP. Each MSHR entry also requires a P-bit. We do not show this in the figure, for clarity.

inserted with a high priority (in case of a cache miss) or promoted to a high priority (in case of a cache hit). If the prefetch is predicted to be inaccurate, it is inserted with the lowest priority (cache miss) or its priority is unchanged (cache hit).

As our study in Section 1 indicates, for many applications, the prefetcher generates a significant fraction of both accurate and inaccurate prefetches. To improve the prefetch accuracy for such applications, we use a separate accuracy counter for each stream tracked by the prefetcher.[4] Our evaluations show that, on average, our proposed accuracy predictor reduces the number of misclassified prefetches from 25% to 14% compared to the accuracy predictor used by FDP [Srinath et al. 2007], which is also used in many other prior works [Zhuang and Lee 2003; Ebrahimi et al. 2009b, 2011; Dahlgren et al. 1995; Nesbit et al. 2004; Lee et al. 2008].

2.3. ICP: Summary of Changes to the Caching Policies

Our final mechanism, ICP, combines the policies employed by both ICP-D and ICP-AP. In summary, ICP introduces five changes to the baseline caching mechanism, which follows the same policy for demand-fetched and prefetched blocks.

(1) When a demand request hits in the cache, if the corresponding block was prefetched, then ICP demotes the block to the lowest priority (see Section 2.1).
(2) When a demand request misses in the cache, if there is an outstanding prefetch request for the block (as indicated by the MSHRs), the block is inserted with the lowest priority (as this scenario is equivalent to the block being prefetched and the demand request hitting in the cache).
(3) When a prefetch request hits in the cache, ICP promotes the block to a high priority only if the prefetched block is predicted to be accurate. Otherwise, the block's replacement state is left unchanged (see Section 2.2).
(4) When a prefetch request misses in the cache, ICP inserts the prefetched block with a high priority only if it is predicted to be accurate. Otherwise, the block is inserted with the lowest priority (see Section 2.2).
(5) When a predicted-inaccurate prefetched block gets evicted from the cache, its address is inserted into the EPF.

3. IMPLEMENTATION AND OVERHEAD ANALYSIS

Figure 3 illustrates the hardware changes required to implement ICP. We now describe these changes in detail.

[4] Our evaluations use per-stream accuracy counters for prior works as well.

3.1. Implementing ICP-D

Implementing ICP-D requires only a single bit in each tag entry and each MSHR entry to indicate if the entry corresponds to a prefetched block. We refer to this bit as the P-bit,. For an outstanding prefetch request, the P-bit of the corresponding MSHR entry is set. Similarly, when a block is prefetched into the cache, the P-bit of the corresponding tag entry is set. When a demand request hits in the cache (or is present in the MSHR), the P-bit is cleared and the block is demoted to the lowest priority (or inserted with the lowest priority). For a 1MB cache with 64B block size, the overhead of the P-bit is 2KB (<0.2% of the cache size). With this modest 0.2% overhead, ICP-D significantly improves system performance (as shown in our evaluations in Section 7).

3.2. Implementing ICP-AP

Implementing our accuracy predictor (ICP-AP) requires four changes on top of the P-bit, required by ICP-D: (1) one accuracy counter per entry in the prefetcher; (2) the EPF, which tracks the addresses of recently evicted prefetched blocks predicted to be inaccurate; (3) a bit per tag indicating if the prefetched block was predicted to be inaccurate (I-bit); and (4) one ID per tag (Pref-ID), indicating which entry in the prefetcher prefetched the block.

The size of the EPF determines how long after the eviction of a mispredicted-inaccurate prefetched block the demand request can arrive for the EPF to detect the misprediction. We find that for each entry keeping track of as many addresses in the EPF as the maximum distance of the prefetcher (D) is sufficient to provide high performance. To further reduce the storage overhead of EPF, we use partial address tags (T bits for each tag). The I-bit and the Pref-ID are required to train the accuracy predictor for each entry. This is in contrast to the P-bit, which is required to determine the promotion policy for each prefetched block. The final storage overhead of ICP-AP is given by $nA + nDT + B(1 + \log n)$ bits, where n is the number of entries in the prefetcher and B is the number of blocks in the cache. For the configuration used in our evaluation ($n = 16$, $A = 4$, $D = 24$, $T = 8$, $B = 16384$), the storage overhead of ICP-AP amounts to 10.38KB ($\approx$1.01% of the 1MB cache size).

4. PRIOR WORK ON MITIGATING PREFETCHER-CAUSED CACHE POLLUTION

In this section, we describe two closely related, state-of-the-art mechanisms to address prefetcher-caused pollution (Section 8 discusses other related work): (1) Prefetch-Aware Cache Management [Wu et al. 2011], and (2) Feedback-Directed Prefetching (FDP) [Srinath et al. 2007], qualitatively comparing them to ICP. In our evaluations (Sections 6 and 7), we quantitatively compare ICP with these two mechanisms. The fundamental novelty of ICP is that it avoids the unnecessary promotion of prefetched blocks on a demand hit based on the observation that an overwhelming majority of useful prefetched blocks are used only once in the cache, thereby significantly mitigating pollution caused by accurate prefetches at the level in which they are inserted.

4.1. Prefetch-Aware Cache Management (PACMan)

PACMan aims to mitigate LLC pollution in a system in which prefetched blocks are inserted into both L3 and L2. PACMan [Wu et al. 2011] proposes two policies. First, it observes that all the demand requests for the accurately prefetched blocks are served by the L2 cache itself, thus they are not used at the L3 cache and can be inserted at the LRU position in the L3 cache. Based on this observation, the first policy, PACMan-M, inserts all prefetched blocks (accurate and inaccurate) at the LRU position. Second, since blocks that receive a prefetch request are potentially “prefetchable,” the second policy, PACMan-H, does not modify the replacement state of a block that receives a hit due to a prefetch request. PACMan-HM employs both PACMan-H and PACMan-M. The final mechanism, PACMan-Dyn, uses set dueling [Qureshi et al. 2007] to choose the best of the different policies.

The main shortcoming of PACMan is that it does not mitigate pollution *at the level in which prefetched blocks are inserted*. This is because, to mitigate pollution at a particular cache level, PACMan inserts prefetched blocks in the previous level, which filters all the demand requests to the prefetched blocks. In contrast, ICP exploits our observation that a majority of useful cache prefetches are used only once to mitigate pollution *at the level in which prefetched blocks are inserted*.

Specifically, in the system evaluated by PACMan, prefetched blocks are inserted both into the L3 cache and the L2 cache. The PACMan approach mitigates pollution *only* in the L3 cache and *does not* mitigate pollution in the L2 cache. However, aggressively prefetching data into the L2 cache will pollute the L2 cache and can potentially degrade both individual application and overall system performance, especially in systems employing multithreading [Intel 2006; Kalla et al. 2010], in which the L2 cache is also shared by applications concurrently running on the same core. In contrast, ICP can mitigate pollution in the L2 cache as well as the L3 cache. In fact, we find that our observation is true for a system with only two levels of cache as well.

In our evaluations (Section 7.4), we find that the baseline that prefetches only into the L3 cache performs better than the baseline that prefetches data into both the L2 and L3 caches (as explained in Section 7.4). While PACMan improves performance in systems that insert prefetched blocks into both L2 and L3, ICP provides the best performance across all prior approaches.

4.2. Feedback-Directed Prefetching

FDP proposes mechanisms to (1) control the aggressiveness of the prefetcher and (2) mitigate prefetcher-caused cache pollution by inserting prefetched blocks at various positions in the LRU stack. To mitigate prefetcher-caused pollution, FDP first estimates the degree of prefetcher-caused pollution by counting the number of demand misses that are caused due to prefetched blocks. FDP augments the cache with a pollution filter that tracks addresses of demand blocks evicted by insertion of prefetched blocks. If the degree of prefetcher-caused pollution is high, then FDP inserts *all* prefetches at the LRU position. The main shortcoming of FDP is that it inserts all prefetches at the LRU position when prefetcher-caused pollution is high. This can lead to some accurately prefetched blocks getting inserted at the LRU position. Inserting an accurate prefetch at the LRU position may lead to the block getting evicted from the cache before a demand request that needs it arrives, resulting in one additional miss for that block. As a result, FDP can potentially degrade performance for applications with a significant fraction of both accurate and inaccurate prefetches.

5. METHODOLOGY

We use an in-house, event-driven x86 multicore simulator [Seshadri 2014] (released publicly) that models both in-order and out-of-order cores. All simulated systems use a three-level cache hierarchy similar to many modern processors (e.g., Intel [2006] and Kalla et al. [2010]). The L1 and L2 caches are private to each core, whereas the L3 cache is shared across all cores. All caches uniformly use a 64B block size. We do not enforce inclusion in any level of the cache hierarchy, similar to many recent works [Duong et al. 2012; Albericio et al. 2013; Seshadri et al. 2012; Pekhimenko et al. 2012; Khan et al. 2014; Sim et al. 2012; Wu et al. 2011], including the closely related prior work, PACMan. In Section 7.5, we describe how our mechanism can be extended to inclusive caches. All caches use the traditional LRU replacement policy. Writebacks do

Table I. Main Simulation Parameters

Core	4Ghz, in-order/out-of-order x86
L1-D Cache	Private, 32KB, 2-way associative, 1 cycle
L2 Cache	Private, 256KB, 8-way associative, 8 cycles
L3 Cache	Shared, 1MB, 16-way associative
Prefetcher	16 streams/core, Degree = 4, Distance = 24
Memory	8 banks, bank conflicts and bus conflicts modeled, 200-cycle bank access latency

not update the replacement policy state. All the proposed mechanisms aim to mitigate prefetcher-caused pollution in the shared LLC.

For our evaluations, we use a multistream prefetcher similar to the one employed in IBM Power6 [Le et al. 2007] used by many prior works [Srinath et al. 2007; Lee et al. 2009; Ebrahimi et al. 2009a, 2009b, 2011; Lee et al. 2008]. The prefetcher monitors the L2 cache misses and prefetches blocks into the LLC. At a high level, each entry in the prefetcher tracks a potential stream. It learns the direction of the stream based on the first few accesses (2 in our evaluations) and then starts prefetching data. The prefetcher always stays a fixed *distance* (number of cache blocks between current demand access and the most recently prefetched block) ahead of the current demand access. The number of simultaneous prefetches sent by the prefetcher is limited by the *degree* of the prefetcher. Table I describes the details of the major system parameters.

As mentioned in Section 4.1, PACMan [Wu et al. 2011] uses a different prefetcher configuration wherein prefetched blocks are inserted both into the L2 and L3 caches. Our evaluations (in Section 7.4) show that our baseline configuration outperforms this other possible prefetcher configuration. However, for completeness, in Section 7.4, we describe three possible prefetcher configurations and compare prior approaches and ICP on top of them. Our evaluations show that ICP on top of our baseline, which inserts prefetched blocks only into the L3 cache, outperforms all other combinations.

For our evaluations, we use benchmarks from SPEC CPU2000/CPU2006 suites, three TPC-H queries, one TPC-C server and an Apache web server. We exclude benchmarks with L2 cache misses per 1000 instructions less than 5 from our evaluations as these benchmarks do not exert significant pressure on the L3 cache. All results are collected by running a representative portion of the benchmarks for 1 billion instructions determined using Simpoint [Sherwood et al. 2002]. The first 500 million instructions are used to warm up the system and the statistics are collected for the next 500 million instructions. For multiprogrammed workloads, all benchmarks keep running till the slowest running benchmark completes 1 billion instructions. Section 7 describes our methodology for generating multiprogrammed workloads in more detail.

For our single-core systems, we use instructions-per-cycle (IPC) to evaluate performance. For multiprogrammed systems, we use four metrics to evaluate performance and fairness: (1) weighted speedup [Snavely and Tullsen 2000; Eyerman and Eeckhout 2008], (2) instruction throughput (IPC), (3) harmonic speedup [Luo et al. 2001; Eyerman and Eeckhout 2008], and (4) maximum slowdown [Kim et al. 2010a, 2010b; Vandierendonck and Seznec 2011; Das et al. 2009].

We compare ICP to three different mechanisms from prior work: (1) Feedback-Directed Prefetching[5] (FDP) [Srinath et al. 2007], (2) Prefetch-Aware Cache Management (PACMan) [Wu et al. 2011], and (3) Feedback-Directed Prefetching-Accuracy Prediction (FDP-AP), which uses the accuracy prediction mechanism proposed by

[5] For FDP, we only implement the portion that modifies the cache insertion policy based on degree of cache pollution. The feedback control mechanism proposed by Srinath et al. [2007] can be combined with our mechanism, as it is solving a complementary problem.

Srinath et al. [2007] to determine the insertion priority for prefetched blocks. Since FDP and PACMan employ a similar approach to address prefetcher-caused pollution, we present results only for PACMan, as PACMan performs better than FDP.

6. SINGLE-CORE ANALYSIS

In this section, we analyze the effect of prefetcher-caused pollution on individual applications. Not all applications are sensitive to cache space. Therefore, even if the prefetches of an application cause high pollution, it may not affect the application's performance if the application is not sensitive to cache space. However, in a multicore system, pollution caused by prefetches of one application can affect the performance of another application. Therefore, there is still benefit in mitigating pollution caused by an application even though that application by itself may not benefit from it.

Our mechanisms primarily aim to mitigate cache pollution caused by prefetches. Since the amount of cache pollution caused by an application does not necessarily correlate with its own performance, we need a metric to directly measure the amount of pollution caused by prefetches. For this purpose, we introduce a metric called *prefetch lifetime*. The goal of our mechanisms is to reduce the prefetch lifetime (i.e., mitigate cache pollution), without losing the benefit of prefetching (i.e., without degrading performance). Note that even though our mechanism may greatly reduce the prefetch lifetime of an application, it may not lead to performance improvement, especially if the application is running alone and is not sensitive to cache space. However, we provide an analysis of the prefetch lifetime metric, as it provides more insight into how our mechanisms can potentially improve overall performance in multicore systems.

We define the *prefetch lifetime* metric in Section 6.1. In Section 6.2, we analyze the effectiveness of different policies on the prefetch lifetime and performance of different applications.

6.1. Quantifying Prefetcher-Caused Cache Pollution

The prefetch lifetime metric is a measure of the time a prefetched block stays in the cache unnecessarily. There are two phases when a prefetched block stays in the cache unnecessarily: (1) before its first use, and (2) after its last use. The first phase indicates how early the block was prefetched before it was actually required by the application. The second phase indicates how long the block stays in the cache without any further use. We define the *prefetch lifetime* of a prefetched block as the sum of the time intervals of the above two phases. We measure prefetch lifetime of a block in terms of the number of misses to the corresponding cache set. For an inaccurate prefetch, the prefetch lifetime consists only of phase 1, as the block never gets used by the application. For a used-once prefetch, the prefetch lifetime is the same as the total amount of time the block stays in the cache, as the first use of the block is also its last use. Finally, for a prefetched block that is used more than once, we just count phase 1, as after the first use, the block becomes "demand-fetched" and any change to the promotion policy of prefetched blocks will not affect the phase 2 of such a block.

For an application that has only inaccurate or used-once prefetched blocks, the ideal average prefetch lifetime is 1 – any inaccurate prefetch is evicted from the cache on the next miss, and any accurately prefetched block is used immediately after it is brought in and evicted on the next miss.[6] If all accurate prefetches of an application are used immediately after they are brought into the cache, the average prefetch lifetime for the application is the same as the associativity of the cache (16 in our experiments).

[6] If inaccurate prefetches can be dropped altogether (as in Lee et al. [2008]), the ideal prefetch lifetime would be less than 1. However, since the focus of this work is on mitigating cache pollution caused by prefetches, we do not evaluate the benefits of dropping inaccurate prefetches (e.g., memory bandwidth savings).

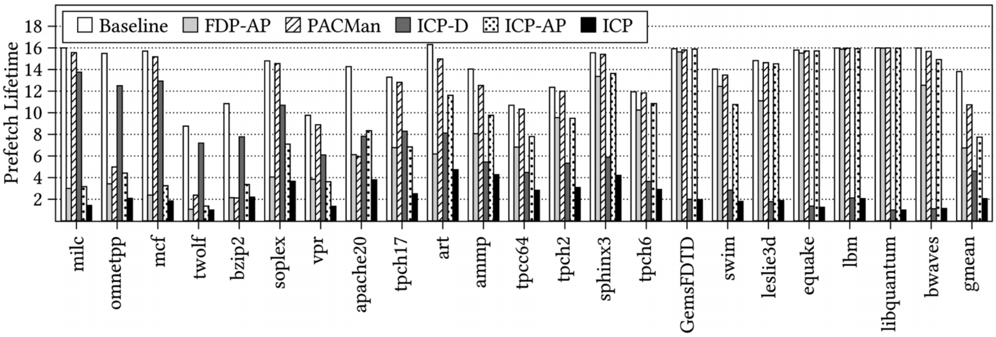


Fig. 4. Reduction in prefetch lifetime due to different mechanisms.

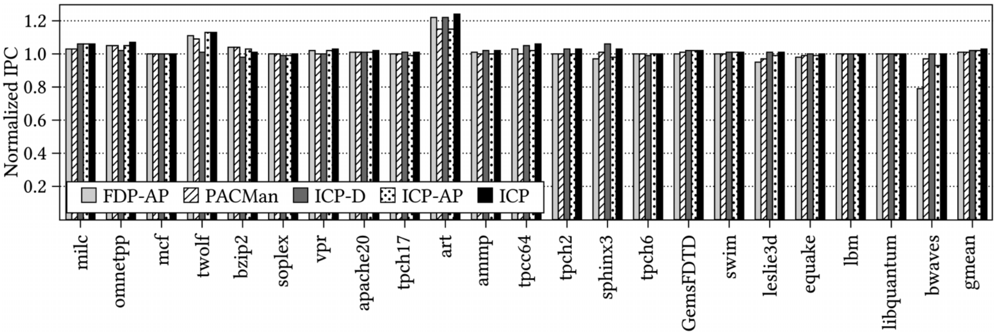


Fig. 5. Effect of different mechanisms on IPC.

As mentioned in the beginning of this section, not all applications are sensitive to cache space. As a result, reduction in prefetch lifetime may not necessarily result in improved performance. For example, a mechanism that inserts all prefetched blocks at the LRU position will likely have a prefetch lifetime of 1 (as most prefetched blocks will be evicted on a subsequent miss to the corresponding cache set). However, this mechanism can significantly degrade performance, as many useful prefetches may get evicted before they are actually used. Therefore, the goal of our mechanisms is to reduce the average prefetch lifetime for an application while improving, or at least not degrading, the performance of the application.

6.2. Comparison Between Different Mechanisms

Figure 4 plots the average prefetch lifetime of different applications[7] using six different policies: the baseline system, the two prior approaches (FDP-AP and PACMan), the two components of our proposed mechanism (ICP-D and ICP-AP), and our final mechanism (ICP). Figure 5 plots the effect of these mechanisms on the performance (IPC) of different applications. The benchmarks are sorted based on the increasing order of fraction of inaccurate prefetches generated by the prefetcher. We draw three conclusions from Figures 4 and 5.

First, both our individual mechanisms ICP-D and ICP-AP significantly reduce the prefetch lifetime of different applications. On the one hand, for applications with a significant fraction of unused prefetches (left end of Figure 4, e.g., *milc*, *omnetpp*),

[7] For clarity, we exclude applications from Figures 4 and 5 that are not sensitive to cache space and those that do not benefit from prefetching.

ICP-AP reduces prefetch lifetime more compared to ICP-D. This is because, unlike ICP-D, ICP-AP inserts most of the inaccurate prefetches of such applications at the LRU position. On the other hand, for applications with a high fraction of prefetches that are used exactly once (right end of Figure 4, e.g., *bwaves*, *libquantum*), ICP-D reduces prefetch lifetime more compared to ICP-AP. This is because, ICP-D demotes the used-once prefetched blocks to the LRU position after their first use, thereby preventing them from unnecessarily polluting the cache. Combining the two policies, ICP significantly reduces average prefetch lifetime across all applications from 13.81 to 2.16 (ideal = 1). Therefore, our results indicate that ICP eliminates most of the pollution caused by prefetches for most applications.

Second, for certain cache-sensitive applications (e.g., *twolf*, *art*, *tpcc64*), the reduction in prefetch lifetime results in significant improvement in performance (up to 24% for *art*). However, as mentioned in the beginning of this section, not all applications are cache sensitive. Therefore, the reduction in prefetch lifetime for such cache-insensitive applications (e.g., *lbm*, *libquantum*) does not result in any performance improvement. However, as we will show in our multicore evaluations (Section 7), ICP significantly improves performance of workloads for which pollution caused by prefetches of one application (e.g., *libquantum*) degrades the performance of other co-running cache-sensitive applications (e.g., *twolf*).

Third, both FDP-AP and PACMan reduce the prefetch lifetime of certain applications that have a high fraction of unused prefetches (e.g., *omnetpp*, *twolf*, *bzip2*). For these applications, both mechanisms also improve performance compared to the baseline. FDP-AP is more effective in reducing prefetch lifetime than PACMan because, while FDP-AP always inserts predicted-inaccurate prefetched blocks at the LRU position, PACMan inserts prefetched blocks at the LRU position only when inserting prefetched blocks at the MRU position degrades performance. However, except for *bzip2*, which has a significant number of prefetches used more than once (Figure 1), ICP outperforms or performs comparably to FDP-AP and PACMan on both prefetch lifetime and performance for all applications. There are two reasons for this. First, unlike FDP-AP or PACMan, ICP mitigates the pollution caused by used-once prefetched blocks. Second, the accuracy predictor used by ICP is better than the accuracy predictor used by FDP-AP. This allows ICP to mitigate the pollution caused by inaccurate prefetches while retaining the benefit of useful prefetches.

In summary, ICP significantly mitigates prefetcher-caused pollution and, as a result, improves performance for cache-sensitive applications (as much as 24% for *art* and 3%, on average, compared to the baseline), outperforming two state-of-the-art approaches to address prefetcher-caused pollution.

7. MULTICORE ANALYSIS

For evaluating multicore systems, we first classify the set of applications in our suite into four categories based on whether their performance is sensitive to cache size and whether they benefit from prefetching. For this classification, we ran each application with four different system configurations: 256KB LLC and 1MB LLC, with and without prefetching. An application is said to benefit from caching if increasing the cache size from 256KB to 1MB improves its performance by at least 5% (both with and without prefetching). Similarly, an application is said to benefit from prefetching if turning on prefetching improves its performance by at least 5% (for both 256KB and 1MB LLC). Table II lists the benchmarks in each category.

Applications in the NC-NP category are neither cache sensitive nor do they benefit from prefetching. Applications in the NC-P category are not cache sensitive, but they benefit significantly from prefetching. These applications have large working sets

Table II. Benchmark Classification

Category	Label	Benchmarks
Not cache sensitive, No prefetch benefit	NC-NP	*astar*, *galgel*, *milc*, *omnetpp*, *xalancbmk*, *zeusmp*
Not cache sensitive, Prefetch benefit	NC-P	*bwaves*, *sphinx3*, *equake*, *GemsFDTD*, *libquantum*
Cache sensitive, No prefetch benefit	C-NP	*ammp*, *apache20*, *bzip2*, *mcf*, *soplex*, *twolf*, *vpr*
Cache sensitive, Prefetch benefit	C-P	*art*, *leslie3d*, *swim*, *tpcc64*, *tpch2*, *tpch6*, *tpch17*

Table III. Two-Core Workload Types

Type	App-1	App-2	Primary Source of Pollution	# Workloads
Type-1	NC*	NC*	Neither	30
Type-2	NC-P	C-NP	NC-P (Acc)	35
Type-3	NC-P	C-P	NC-P (Acc)	35
Type-4	C-P	C-P	Both (Acc + Inacc)	28
Type-5	C-NP	C-NP	Both (Inacc)	28

*NC includes NC-NP and NC-P. Acc: Pollution due to accurate prefetches. Inacc: Pollution due to inaccurate prefetches.

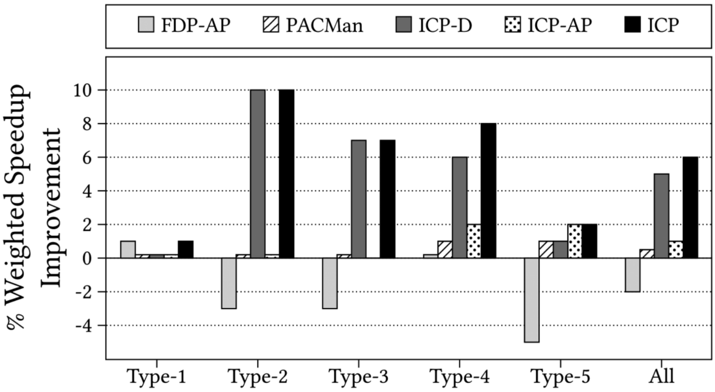


Fig. 6. Two-core – System performance.

and almost all of their useful prefetches are used only once (as shown in Figure 1). Hence, although they do not benefit from caching themselves, when following the traditional policies, the prefetched blocks of these applications cause high pollution (e.g., *libquantum*, *bwaves*). Applications from the C-NP category are sensitive to cache space. However, they do not benefit from prefetching. Therefore, the performance of these applications can get affected by the pollution caused by prefetches of other co-running applications. Finally, applications in the C-P category benefit both from additional cache space and from prefetching. They can potentially suffer performance degradation due to cache pollution and can also cause high pollution.

7.1. Two-Core: System Performance

Based on this classification, we generate five different groups of 2-core workloads to evaluate our 2-core system. Table III lists the benchmark categories for each workload type and the number of evaluated workloads in each type. In this section, we present the results comparing the efficacy of different mechanisms to mitigate prefetcher-caused pollution for such workloads.

Figure 6 shows the improvement in weighted speedup due to different mechanisms compared to the baseline system that employs prefetching. For each workload type, Figure 6 shows the average weighted speedup improvement across all workloads that belong to that type. We draw several conclusions from the figure.

First, Type-1 workloads have no cache-sensitive applications. As such, there is little opportunity to improve performance by mitigating prefetcher-caused cache pollution. This is reflected in our results, which show that ICP improves performance by only 1% compared to the baseline.

Second, Type-2 and Type-3 workloads have one application from the NC-P category. While applications in the NC-P category benefit significantly from prefetching, existing policies for managing their prefetched blocks cause a lot of pollution by unnecessarily promoting prefetched blocks that are used only once. On the other hand, ICP-D mitigates this problem by demoting the prefetched blocks after their first use, thereby significantly improving performance (10%, on average, for Type-2 and 7%, on average, for Type-3).

Third, Type-4 workloads have both applications from the C-P category. Many applications in this category (e.g., *art*, *tpcc64*) have a significant fraction of both inaccurate and used-once prefetches. As a result, both ICP-D and ICP-AP improve weighted speedup by 6% and 2%, respectively, compared to the baseline. ICP combines the benefits of both policies and improves weighted speedup by 8% compared to the baseline.

Fourth, Type-5 workloads have both applications from the C-NP category. The primary source of prefetcher-caused cache pollution for these workloads is inaccurate prefetches that evict more useful blocks from the cache. ICP-AP, which reduces prefetch lifetime of such applications by inserting inaccurate prefetches at the LRU position, improves weighted speedup by 2% across all 28 Type-5 workloads.

Fifth, FDP-AP, in general, degrades performance for multicore workloads, in contrast to single-core systems. This is because the accuracy predictor used by FDP-AP suffers from the positive feedback problem wherein a prefetcher entry classified as inaccurate continues to get classified as inaccurate. As mentioned earlier, this problem becomes worse in multicore systems, in which a misclassified accurate-prefetch of one application can get evicted by a block of another application. Our proposed accuracy predictor, ICP-AP, avoids this problem by keeping track of a small set of prefetched blocks predicted to be inaccurate.

Finally, we find that PACMan has little impact on performance for most workloads. As we described in Section 4.1, the policies of PACMan were specifically designed to address prefetcher pollution in the L3 cache when blocks were prefetched into the L2 cache. Our evaluations in Section 7.4 show that the PACMan policies improve performance on such systems. However, ICP on top of our baseline prefetcher configuration performs best across all possible prior approaches. In addition, in configurations that prefetch blocks into both L2 and L3 caches, unlike prior approaches, our approach can be used to mitigate pollution in both cache levels.

Summary of Multicore Results. Figure 7 shows the system performance improvement of ICP across all 2-core workloads. The workloads are sorted based on the performance improvement due to ICP. On average, across all the 157 evaluated 2-core workloads, ICP improves performance by 6% compared to both the baseline (with prefetching) and the best previous mechanism. ICP performs better than or comparably to the baseline for 151 of the 157 workloads. For the remaining 6 workloads, ICP degrades performance by 1% at most. We conclude that ICP is effective at improving system performance by reducing prefetcher-caused cache pollution.

In general, the performance improvements due to ICP-AP are not significant in spite of its ability to significantly reduce the prefetch lifetime of the inaccurately prefetched blocks. ICP-D, on the other hand, significantly improves performance by mitigating pollution caused by accurately prefetched blocks.

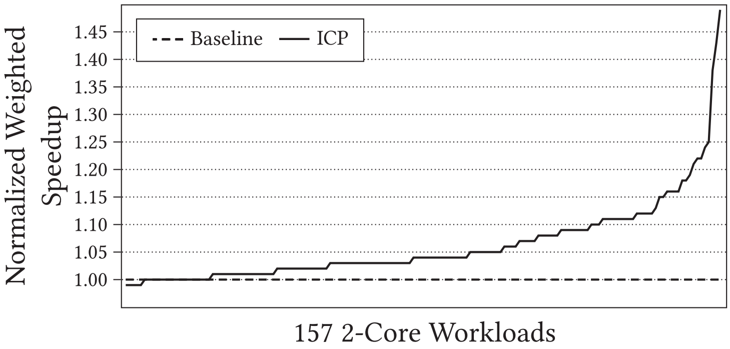


Fig. 7. Two-core: System performance for all workloads.

Table IV. Two-core: Other Metrics

Type	Instruction Throughput Improvement	Harmonic Speedup Improvement	Maximum Slowdown Reduction
Type-1	3%	3%	3%
Type-2	10%	10%	6%
Type-3	7%	7%	4%
Type-4	9%	8%	8%
Type-5	2%	2%	1%
All	6%	6%	4%

7.2. Two-Core: Other Metrics

Table IV plots the improvement of ICP over baseline for 2-core workloads on three other metrics: (1) instruction throughput, (2) harmonic speedup [Luo et al. 2001], and (3) maximum slowdown [Kim et al. 2010b, 2010a; Vandierendonck and Seznec 2011; Das et al. 2009]. As our results indicate, ICP improves both performance and fairness, compared to the baseline, across all evaluated metrics.

7.3. Sensitivity to Different Parameters

In this section, we analyze the sensitivity of our proposed mechanisms to different system parameters. We focus our attention on the 98 Type-2, Type-3, and Type-4 workloads that suffer significantly from prefetcher-caused pollution.

7.3.1. Effect of Varying Cache Size. Figure 8 shows the effect of varying the cache size on the performance improvement due to our proposed mechanisms compared to the baseline system. As the results indicate, in general, system performance improvement decreases with increasing cache size. This is expected, as prefetcher-caused pollution becomes less of a problem as cache size increases. However, even for 4MB and 8MB cache sizes, ICP improves performance by as much as 14% and 27% compared to the baseline, respectively.

7.3.2. Effect of Varying Memory Latency. Figure 9 shows the effect of increasing memory latency on performance improvement due to our proposed mechanisms compared to the baseline system. With increasing memory latency, the performance improvements due to ICP also increase. This is because, as the memory latency increases, effective cache utilization becomes more important for system performance. By mitigating prefetcher-caused pollution, ICP creates more cache space for the cache-sensitive applications in the workloads, thereby improving overall cache utilization. As such, we conclude that ICP is an attractive mechanism for future systems, which are expected to have

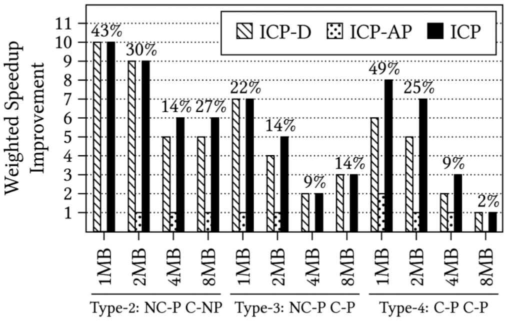


Fig. 8. Effect of varying cache size (percentage on top indicates the peak improvement of ICP for the corresponding category).

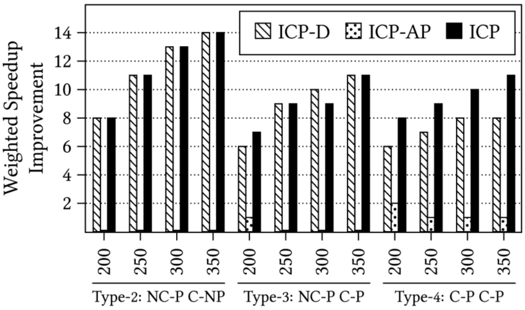


Fig. 9. Effect of varying memory latency (the x-axis indicates the memory latency in terms of number of cycles).

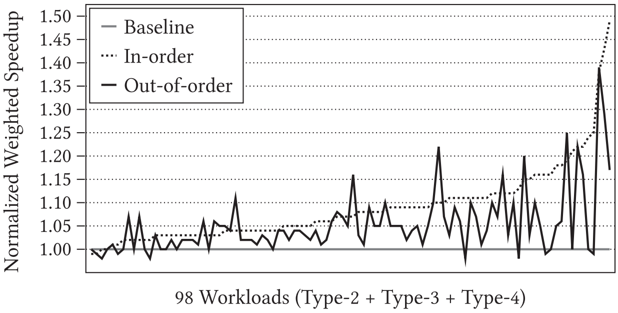


Fig. 10. In-order vs. out-of-order cores.

high memory access latency due to contention between multiple concurrently running applications.

7.3.3. In-Order vs. Out-of-Order. So far, we have presented results with systems that use in-order cores. Figure 10 compares the performance improvement due to ICP over a baseline system using in-order cores and out-of-order cores. Our simulator models a single-issue, out-of-order core with a 64-entry instruction window. For each workload on the x-axis, the "in-order" curve plots the performance improvement of ICP in an in-order system compared to the in-order baseline. The corresponding "out-of-order" curve plots the performance improvement of ICP in an out-of-order system compared to the out-of-order baseline. In general, the performance improvement of ICP in an out-of-order system is less than the corresponding improvement in an in-order system. This is expected, as the out-of-order core is more tolerant of memory latency. This is because it generates multiple concurrent cache misses that are served in parallel by the memory system, overlapping the latency of such misses. However, even in the out-of-order system, ICP improves performance by as much as 39% compared to the baseline (5% on average). Only 7 out of the 98 workloads experience small (at most 2%) performance degradation. These results indicate that ICP is effective even with out-of-order cores.

7.3.4. Benefits with Stride Prefetching. The results presented so far use a stream prefetcher [Le et al. 2007]. However, our proposed mechanisms can be naturally implemented with any prefetching scheme. To evaluate this benefit, we ran experiments with a system that uses a stride prefetcher [Baer and Chen 1995]. Across the 2-core

Table V. Effect of the Cache Replacement Policy on ICP

Mechanism	Weighted Speedup Increase over Baseline
ICP + LRU	6%
DRRIP [Jaleel et al. 2010b]	1%
ICP + DRRIP	8%

Table VI. Effect of Varying Number of Cores

Weighted speedup improvement over baseline	2-core	4-core	8-core
	6%	12%	26%

workloads, ICP improves system performance by as much as 28% (2% on average). The average improvements are low compared to a system using stream prefetching because, unlike the stream prefetcher used in our main evaluations, the stride prefetcher is conservative and issues prefetches only when it detects a regular stride. The pollution caused by prefetches is thus low to begin with. However, our baseline stream prefetcher provides better performance compared to the stride prefetcher (10% across the single-core applications).

7.3.5. Effect of the Cache Replacement Policy. So far, in our evaluations, the LLC uses the LRU replacement policy. However, ICP can be applied on top of any cache replacement policy (e.g., Jaleel et al. [2010b], Hallnor and Reinhardt [2000], Qureshi et al. [2007, 2006], Rajan and Ramaswamy [2007], Chaudhuri [2009], Keramidas et al. [2007], Basu et al. [2007], and Seznec [1993]). We evaluate this by implementing ICP on top of the recently proposed Dynamic Re-reference Interval Prediction (DRRIP) policy [Jaleel et al. 2010b]. We insert predicted-accurate prefetches with the *long* re-reference prediction value (RRPV), demote prefetched blocks to the *distant* RRPV (lowest priority). Predicted-inaccurate prefetches are inserted with the *distant* RRPV. The insertion and promotion policies for demand-fetched blocks are not changed. We refer the reader to Jaleel et al. [2010b] for more details on the DRRIP policy. Table V shows the weighted speedup of ICP using different replacement policies normalized to the baseline.

We make two observations. First, in the presence of prefetching, the DRRIP policy only slightly improves performance compared to the baseline LRU policy for SPEC CPU2006 applications. This is in line with the results and observations made by prior work, PACMan [Wu et al. 2011] (refer to Figures 1 and 12 of the PACMan paper).

Second, ICP on DRRIP improves performance by 7% compared to the baseline DRRIP policy. Based on this, we conclude that ICP can significantly improve performance on top of the state-of-the-art, prefetch-unaware replacement policies.

7.3.6. Sensitivity to Number of Cores. We study the effect of increasing the number of cores on the performance improvement due to ICP. For these experiments, we fix the size of the LLC at 4MB. We generate 20 four-core and 20 eight-core workloads (similar to Type-2 two-core workloads). Table VI summarizes the results of these experiments.

As the number of cores increases, the performance improvement due to ICP also increases. This is because, for a fixed cache size, the cache pressure increases with increasing number of cores. Consequently, the negative impact of prefetcher-caused pollution also increases. We conclude that ICP is an attractive mechanism for future many-core systems with a shared LLC.

7.4. Other Prefetcher Configurations

So far, we have discussed results with a prefetcher that trains on L2 misses and inserts prefetched blocks only into the L3. However, prior approaches (e.g., Wu et al. [2011]) have used other potential prefetcher configurations. Table VII lists three

Table VII. Baseline Prefetcher Configurations

Label	Prefetcher trains on	Prefetches inserted into
C1	L2 misses	L3 only
C2	L2 misses	L2 and L3
C3	L1 misses	L2 and L3

Table VIII. Two-Core Performance with Different Prefetcher Configurations

Mechanism	Normalized Weighted Speedup
Baseline C1	1.00
Baseline C2	0.88
Baseline C3	0.97
PACMan [Wu et al. 2011] + C1	1.00
PACMan + C2	0.96
PACMan + C3	1.01
ICP + C1	1.06

possible prefetcher configurations. The baseline used so far in our evaluations corresponds to C1. The configuration C2 is where the prefetcher is trained on L2 misses and the prefetches are inserted both into L2 and L3. Finally, configuration C3 is one in which the prefetcher is trained on L1 misses and prefetches are inserted both into L2 and L3.

Table VIII presents average 2-core performance across all 157 two-core workloads for various combinations of prefetcher configurations and pollution mitigating approaches. The results are normalized to the baseline C1 configuration.

Several conclusions are in order. First, our baseline C1 configuration outperforms the other two configurations. This is because both C2 and C3 lead to L2 cache pollution. In addition, C2 filters the accesses to prefetched blocks from the prefetcher, preventing the prefetcher from staying ahead of the demand access stream. Second, the insights proposed by PACMan [Wu et al. 2011] work well when the prefetched blocks are inserted both into the L2 and L3, as indicated by the performance improvement of PACMan over the C2 (9%) and C3 (4%) configurations. Note that although PACMan was proposed as a modification to DRRIP, the insights of PACMan are very much applicable on top of an LRU baseline, as evidenced by the 4% improvement in performance for the C3 baseline across 157 workloads. However, our mechanism ICP with the C1 configuration performs the best across all systems.

The goal of this experiment was to establish the performance of different possible baselines and show that ICP improves performance significantly on top of the best baseline. Having said this, we believe that our observations can be exploited to improve performance for any prefetcher configuration.

7.5. Extending ICP to Inclusive Caches

Although we have presented and evaluated our mechanism in a noninclusive LLC, we believe our observation can be used to improve performance for any cache/prefetcher configuration. For example, in a system with an inclusive cache hierarchy, evicting a block from the LLC requires the block to be evicted from all previous levels of the cache. Our mechanism demotes a prefetched block to the lowest priority on a demand hit. This may lead to premature eviction of the block from all the caches and may result in L1 cache miss for accesses that may have otherwise hit in the L1 cache. This is a problem with any cache management policy that inserts/demotes blocks at/to the lowest priority (e.g., Qureshi et al. [2007] and Jaleel et al. [2010b]). A recent prior work [Jaleel et al. 2010a] has proposed a number of temporal-locality aware policies to address this problem. The idea behind their mechanism is to not evict a block from the LLC if the block is still present in any previous level of cache. While we are unable to evaluate the effect of our mechanism on inclusive caches due to limitations in our infrastructure, it is clear that our observations can be effectively combined with this mechanism to further improve performance for inclusive caches. While ICP can demote a prefetched block as soon as it is consumed by a demand access, the policies proposed by Jaleel et al. [2010a] can be used to delay the eviction until the block is fully utilized by the higher-level caches (e.g., L1). As such, the insights in this article can be adapted and applied to different ways of handling inclusion/exclusion in caches.[8]

8. RELATED WORK

The primary contribution of this work is a comprehensive mechanism to address cache pollution caused by *both inaccurate and accurate prefetches*. To our knowledge, this is the first work to identify the shortcoming of the promotion policy employed by existing cache management policies when a demand request hits on a prefetched block. We have already provided qualitative comparisons to the most closely related prior work: Feedback-Directed Prefetching (FDP) [Srinath et al. 2007], and Prefetch-Aware Cache Management (PACMan) [Wu et al. 2011]. In this section, we discuss other related work.

Our ICP-D mechanism essentially uses a demand hit on a prefetched block as an indicator that the block is dead. Prior works [Lai et al. 2001; Khan et al. 2010; Hu et al. 2002] have proposed many mechanisms to predict dead blocks. Lai et al. [2001] and Hu et al. [2002] propose dead block prediction mechanisms for L1 caches. On the one hand, these mechanisms are complex to implement in large L2 and L3 caches. On the other hand, our approach does not work effectively for primary L1 caches, where different words within a prefetched block may be accessed by different demand requests. Therefore, we believe these mechanisms can be combined favorably with our proposed mechanisms. Khan et al. [2010] propose a sampling dead block predictor for LLCs. This mechanism identifies program counters that generate the last access to different cache blocks and uses this information to predict dead blocks. In contrast to this mechanism, our mechanism is simpler—it requires only one bit per tag entry and does not require the program counter information from the processor core to be propagated to the LLC.

The Prefetch Pollution Filter [Zhuang and Lee 2003] aims to filter away inaccurate prefetches from polluting the L1 cache by using a table of counters (indexed based on the address of blocks evicted from the cache) to track the accuracy of prefetches generated. Extending this mechanism to the LLC will require a large table to track blocks evicted from the LLC. This mechanism also suffers from the positive feedback problem described in Section 2.2. Lin et al. [2001b] propose a density vector, a mechanism to drop superfluous prefetches generated by a Scheduled Region Prefetcher [Lin et al. 2001a]. The proposed mechanism is very specific to the Scheduled Region Prefetcher.

Prior works [Srinath et al. 2007; Ebrahimi et al. 2009a, 2009b, 2011; Dahlgren et al. 1995; Nesbit et al. 2004] proposed techniques to control the aggressiveness of the prefetcher depending on the various metrics (e.g., accuracy, coverage, degree of pollution, interference caused to other cores). While varying the aggressiveness of a prefetcher may help mitigate pollution caused by inaccurate prefetches, it does not address the pollution caused by promoting used-once prefetches. We believe ICP is complementary to these throttling techniques because (1) ICP-D can mitigate the pollution caused by accurate prefetches, and (2) ICP-AP can be used to make better throttling decisions.

[8] Note that many modern processors do not employ a strictly inclusive cache hierarchy (e.g., AMD [2012] and VIA [2005]). Our mechanism can be easily integrated into such processors to significantly improve performance.

Jain et al. [2001] proposed a mechanism to throttle prefetching to mitigate cache pollution. Their mechanism used software hints to mark cache blocks as dead and prefetches are inserted into the cache only if there is a dead block in the corresponding set. This mechanism can potentially throttle useful prefetches, degrading the performance of applications that benefit from aggressive prefetching.

Alameldeen and Wood [2007] proposed a mechanism to identify prefetcher-caused pollution by using the extra hardware tags provisioned for cache compression. Similar to the techniques proposed by FDP [Srinath et al. 2007], this approach can only mitigate pollution caused by inaccurate prefetches.

Prior works [Cao et al. 1995; Patterson et al. 1995; Albers and Büttner 2003] have studied the interaction between prefetching and caching in the context of file systems. However, these works assume that the future reference stream to the data blocks in disk is known *a priori* (e.g., through software hints). Based on that information, they make informed decisions about whether and when to prefetch a disk block. However, such a priori information is difficult to obtain for main memory access streams.

9. CONCLUSION

Caching and prefetching are techniques employed by modern high-performance processors to mitigate the impact of long memory latency. Prefetching data into the on-chip caches, as done by many such processors, can lead to prefetcher-caused cache pollution, that is, prefetched blocks evict more useful blocks from the cache.

In this work, we identified two root causes for prefetcher-caused pollution: (1) unnecessarily promoting an accurately prefetched block on a demand hit, and (2) inserting inaccurate prefetches with a high priority into the cache. We presented Informed Caching policies for Prefetched blocks (ICP), a comprehensive mechanism to mitigate prefetcher-caused cache pollution. ICP mitigates pollution by demoting a prefetched block to the lowest priority on a demand hit (as such blocks are rarely reused) and predicting the accuracy of each prefetched block and inserting only likely-accurate prefetches with a high priority into the cache. ICP incurs only a modest storage overhead (1.25% of the LLC size).

Our evaluations show that ICP significantly mitigates prefetcher-caused pollution and, as a result, improves performance for workloads that suffer from prefetcher-caused pollution (up to 47%) on a wider variety of workloads and system configurations. We conclude that ICP is an effective and low-cost mechanism to mitigate the performance degradation caused by prefetcher-caused pollution.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We acknowledge members of the SAFARI and LBA groups for their feedback and for the stimulating research environment they provide. We specifically thank Lavanya Subramanian and Nandita Vijaykumar for their feedback and comments on early versions of this paper. We acknowledge the generous support of Intel, Qualcomm, and Samsung. This work is supported in part by NSF grants 0953246, 1212962, 1320531, the Intel Science and Technology Center for Cloud Computing, and the Semiconductor Research Corporation.

REFERENCES

Alaa R. Alameldeen and David A. Wood. 2007. Interactions between compression and prefetching in chip multiprocessors. In *HPCA*.

Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. 2013. The reuse cache: downsizing the shared last-level cache. In *MICRO*.

Susanne Albers and Markus Büttner. 2003. Integrated prefetching and caching in single and parallel disk systems. In *SPAA*.

AMD. 2012. AMD Phenom II processor model. Retrieved November 11, 2014 from http://www.amd.com/en-us/products/processors/desktop/phenom-ii. (2012).

Jean-Loup Baer and Tien-Fu Chen. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE TC* (1995).

Arkaprava Basu, Nevin Kirman, Meyrem Kirman, Mainak Chaudhuri, and Jose F. Martinez. 2007. Scavenger: a new last level cache architecture with global block priority. In *MICRO*.

Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. 1995. A study ofintegrated prefetching and caching strategies. In *SIGMETRICS*.

Mainak Chaudhuri. 2009. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. In *MICRO*.

Fredrik Dahlgren, Michel Dubois, and Per Stenström. 1995. Sequential hardware prefetching inshared-memory multiprocessors. *IEEE TPDS*.

Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *MICRO*.

Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving cache managementpolicies using dynamic reuse distances. In *MICRO*.

Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. In *ISCA*.

Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009b. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO*.

Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009a. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching dystems. In *HPCA*.

Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro*.

Erik G. Hallnor and Steven K. Reinhardt. 2000. A fully associative software-managed cache design. In *ISCA*.

Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. 2002. Timekeeping in the memory system: predicting and optimizing memory behavior. In *ISCA*.

Intel. 2006. Inside Intel Core microarchitecture and smart memory access. Intel White Paper.

Prabhat Jain, Srini Devadas, and Larry Rudolph. 2001. *Controlling Cache Pollution in Prefetching with Software-assisted Cache Replacement*. Technical Report CSG-462. Massachusetts Institute of Technology, Cambridge, MA.

Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. 2010a. Achieving non-inclusive cache performance with inclusive caches: temporal locality aware (TLA) cache management policies. In *MICRO*.

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010b. High performance cache replacement using re-reference intervalprediction (RRIP). In *ISCA*.

Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. 2010. Power7: IBM's next-generation server processor. *IEEE Micro*.

Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *ICCD*.

Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jimenez. 2014. Improving cache performance using read-write partitioning. In *HPCA*.

Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. 2010. Sampling dead block prediction for last-level caches. In *MICRO*.

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010a. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*.

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010b. Thread cluster memory scheduling: exploiting differences in memory access behavior. In *MICRO*.

An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction and dead-block correlating prefetchers. In *ISCA*.

H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. 2007. IBM Power6 microarchitecture. *IBM JRD*.

Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. 2008. Prefetch-aware DRAM controllers. In *MICRO*.

Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. 2009. Improving memory bank-level parallelism in the presence of prefetching. In *MICRO*.

Wei-Fen Lin, Steven K. Reinhardt, and Doug Burger. 2001a. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA*.

Wei-Fen Lin, Steven K. Reinhardt, Doug Burger, and Thomas R. Puzak. 2001b. Filtering superfluous prefetches using density vectors. In *ICCD*.

Kun Luo, Jayanth Gummaraju, and Manoj Franklin. 2001. Balancing throughput and fairness in SMT processors. In *ISPASS*.

Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. 2004. AC/DC: An adaptive data cache prefetcher. In *PACT*.

Oracle. 2011. Oracle's Sparc T4 server architecture. Oracle White Paper.

R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed prefetching and caching. In *SOSP*.

Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In *PACT*.

Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *ISCA*.

Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A case for MLP-aware cache replacement. In *ISCA*.

Kaushik Rajan and Govindarajan Ramaswamy. 2007. Emulating optimal replacement with a shepherd cache. In *MICRO*.

Vivek Seshadri. 2014. Source code for Mem-Sim. Retrieved November 11, 2014 from www.ece.cmu.edu/∼safari/tools.html. (2014).

Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. 2012. The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. In *PACT*.

André Seznec. 1993. A case for two-way skewed-associative caches. In *ISCA*.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ASPLOS*.

Jaewoong Sim, Jaekyu Lee, Moinuddin K. Qureshi, and Hyesoon Kim. 2012. FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion. In *ISCA*.

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *ASPLOS*.

Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA*.

Hans Vandierendonck and André Seznec. 2011. Fairness metrics for multithreaded processors. *IEEE Computer Architecture Letters* (Jan. 2011).

VIA. 2005. VIA C7 Processor. Retrieved November 11, 2014 from http://www.via.com.tw/en/products/processors/c7/. (2005).

Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. 2011. PACMan: prefetch-aware cache management for high performance caching. In *MICRO*.

Xiaotong Zhuang and Hsien-Hsin S. Lee. 2003. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *ICPP*.

Received February 2014; revised October 2014; accepted October 2014