The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing

Vivek Seshadri[†] vseshadr@cs.cmu.edu
Onur Mutlu[†] onur@cmu.edu
Michael A Kozuch[⋆] michael.a.kozuch@intel.com
Todd C Mowry[†] tcm@cs.cmu.edu

[†]Carnegie Mellon University [⋆]Intel Labs Pittsburgh

SAFARI Technical Report No. 2012-003

Abstract

Off-chip main memory has long been a bottleneck for system performance. With increasing memory pressure due to multiple on-chip cores, effective cache utilization is important. In a system with limited cache space, we would ideally like to prevent 1) cache pollution, i.e., blocks with low reuse evicting blocks with high reuse from the cache, and 2) cache thrashing, i.e., blocks with high reuse evicting each other from the cache.

In this paper, we propose a new, simple mechanism to predict the reuse behavior of missed cache blocks in a manner that mitigates both pollution and thrashing. Our mechanism tracks the addresses of recently evicted blocks in a structure called the *Evicted-Address Filter* (EAF). Missed blocks whose addresses are present in the EAF are predicted to have high reuse and all other blocks are predicted to have low reuse. The key observation behind this prediction scheme is that if a block with high reuse is prematurely evicted from the cache, it will be accessed soon after eviction. We show that an EAF-implementation using a Bloom filter, which is cleared periodically, naturally mitigates the thrashing problem by ensuring that only a portion of a thrashing working set is retained in the cache, while incurring low storage cost and implementation complexity.

We compare our EAF-based mechanism to five state-of-the-art mechanisms that address cache pollution or thrashing, and show that it provides significant performance improvements for a wide variety of workloads and system configurations.

1 Introduction & Motivation

Off-chip main memory has long been a bottleneck for system performance. Many modern applications demand increasingly more bandwidth and reduced latency from main memory. In addition, with the evolution of chip-multiprocessors, multiple applications run simultaneously on the same chip, increasing the pressure on the memory subsystem. In most modern processor designs, such concurrently running applications share the on-chip last-level cache [1, 2, 3, 22]. As a result, effective use of the available cache space is critical for high system performance.

Ideally, to ensure high performance, the cache should be filled only with blocks that have high temporal reuse – blocks that are likely to be accessed multiple times within a short time interval. However, as

identified by prior work (e.g., [17, 19, 34, 35]), two problems degrade cache performance significantly. First, cache blocks with little or no reuse can evict blocks with high reuse from the cache. This problem is referred to as *cache pollution*. Second, when there are a large number of blocks with high reuse, they start evicting each other from the cache due to lack of space. This problem is referred to as *cache thrashing*. Both pollution and thrashing increase the miss rate of the cache and consequently reduce system performance. Prior work proposed to modify the cache insertion policy to mitigate the negative effects of pollution and thrashing [16, 17, 19, 34, 35, 52, 54].

To prevent cache pollution, prior approaches [19, 34, 52, 54] predict the reuse behavior of missed cache blocks and insert blocks predicted to have low reuse with a low priority – e.g., at the least-recently-used (LRU) position for the LRU replacement policy. This ensures that such low-reuse blocks get evicted from the cache quickly, thereby preventing them from polluting the cache. To predict the reuse behavior of missed cache blocks, these mechanisms *group* blocks based on the program counter that accessed them [34, 54] or the memory region to which the blocks belong [19, 54]. The mechanisms subsequently learn the reuse behavior of each group and use that to predict the reuse behavior of an individual cache block. As such, they do not distinguish between the reuse behavior of blocks within a group.

To prevent thrashing, recent prior work [35] proposed the use of a thrash-resistant bimodal insertion policy (BIP). BIP inserts a majority of missed blocks with low priority (at the LRU position) and a small fraction of blocks with high priority (at the most-recently-used (MRU) position). By doing so, a fraction of the working set can be retained in the cache, increasing the hit rate when the working set is larger than the cache size. When multiple threads share the cache, prior approaches [16, 17] learn the thrashing behavior of individual threads using a technique called set-dueling [35, 36], and use BIP for those threads that are determined to suffer from thrashing. As such, these approaches do not distinguish between the reuse behavior of cache blocks within a thread.

The Problem: In this work, we observe that previous proposals are not effective in preventing both pollution and thrashing at the same time. On one hand, proposals that address cache pollution do not distinguish between the reuse behavior of blocks that belong to the same group. As a result, these mechanisms can lead to many mispredictions when the reuse behavior of an individual block does not correlate with the reuse behavior of the group to which it belongs. Moreover, these approaches do not have any inherent mechanism to detect and prevent thrashing. If a large number of blocks are predicted to have high reuse, they will evict each other from the cache. On the other hand, since proposals that address thrashing use thread-level behavior to detect thrashing, they cannot distinguish between high-reuse blocks and low-reuse blocks within a thread. As a result, they either insert low-reuse blocks of a non-thrashing thread with high priority and hence, pollute the cache, or they repeatedly insert high-reuse blocks of a thrashing thread with the bimodal insertion policy, potentially leading to cache under-utilization. Since approaches to prevent pollution and approaches to prevent thrashing both modify the cache insertion policy using different mechanisms, it is difficult to combine them to address both problems concurrently. Our goal in this work is to design a mechanism that seamlessly reduces both pollution and thrashing to improve system performance.

Our Approach: To prevent cache pollution, one would like to predict the reuse behavior of a missed cache block and prevent low-reuse blocks from polluting the cache by choosing an appropriate cache insertion policy. As such, we take an approach similar to prior work, but unlike prior work which predicts the reuse behavior of a *group* of blocks, we predict the reuse behavior of *each* missed block based on *its own* past behavior. Unfortunately, keeping track of the behavior of all blocks in the system would incur large storage overhead and lookup latency. We eliminate this high cost by taking advantage of the following observation: *If a block with high reuse is prematurely evicted from the cache, it will likely be accessed soon after eviction. On the other hand, a block with low reuse will not be accessed for a long time after eviction.*

This observation indicates that it is sufficient to keep track of a small set of recently evicted blocks to predict the reuse behavior of missed blocks – blocks evicted a long time ago are unlikely to have high reuse. To implement this prediction scheme, our mechanism keeps track of *addresses* of recently evicted blocks in a hardware structure called the *Evicted-Address Filter* (EAF). If a missed block's address is present in the EAF, the block is predicted to have high reuse. Otherwise, the block is predicted to have low reuse.

Cache thrashing happens when the working set is larger than the cache. In the context of EAF, there are two cases of thrashing. First, the working set can be larger than the aggregate size of the blocks tracked by the cache and the EAF together. We show that this case can be handled by using the thrash-resistant bimodal insertion policy [35] for low-reuse blocks, which ensures that a fraction of the working set is retained in the cache. Second, the working set can be larger than the cache but smaller than the aggregate size of the blocks tracked by the cache and the EAF together. In this case, thrashing can be mitigated by restricting the number of blocks predicted to have high reuse to a value smaller than the number of blocks in the cache. We find that implementing EAF using a Bloom filter enables this effect by forcing the EAF to be periodically cleared when it becomes full. Doing so results in the EAF predicting only a portion of the working set to have high reuse, thereby mitigating thrashing. We describe the required changes that enable the EAF to mitigate thrashing in detail in Section 2.3, and evaluate them quantitatively in Section 7.4.

Thus, our mechanism can reduce the negative impact of *both* cache pollution and thrashing using a single structure, the Evicted-Address Filter.

Summary of Operation: Our mechanism augments a conventional cache with an Evicted-Address Filter (EAF) that keeps track of addresses of recently evicted blocks. When a block is evicted from the cache, the block's address is inserted into the EAF. On a cache miss, the cache tests whether the missed block address is present in the EAF. If yes, the block is predicted to have high reuse and inserted with a high priority into the cache. Otherwise, the block is predicted to have low reuse and inserted with the bimodal insertion policy. When the EAF becomes full, it is completely cleared. We show that EAF naturally lends itself to a low-cost and low-complexity implementation using a Bloom filter (Section 3.2).

Using EAF to predict the reuse behavior of missed cache blocks has three benefits. First, the hardware implementation of EAF using a Bloom filter has low overhead and complexity. Second, the EAF is completely outside the cache. As a result, our mechanism does not require any modifications to the existing cache structure, and consequently, integrating EAF in existing processors incurs low design and verification cost. Third, EAF operates only on a cache miss and does not modify the cache hit operation. Therefore, it can be favorably combined with other techniques that improve performance by monitoring blocks while they are in the cache (e.g., better cache replacement policies).

We compare our EAF-augmented cache (or just EAF-cache) with five state-of-the-art cache management approaches that aim to prevent pollution *or* thrashing: 1) Thread-aware dynamic insertion policy [16] (TA-DIP), 2) Thread-aware dynamic re-reference interval prediction policy [17] (TA-DRRIP), 3) Signature-based Hit Prediction using Instruction pointers (SHIP) [54], 4) Run-time cache bypassing [19] (RTB), and 5) Miss classification table [9] (MCT). Our evaluations show that EAF-cache significantly outperforms all prior approaches for a wide variety of workloads and on a number of system configurations.

We make the following contributions:

- We show that keeping track of a small set of recently evicted blocks can enable a new and low-cost per-block reuse prediction mechanism.
- We provide a low-overhead, practical implementation of a new cache insertion policy based on Evicted-Address Filters (EAF), which mitigates the negative impact of both cache pollution and thrashing.

- We compare the EAF-cache with five state-of-the-art cache management mechanisms using a wide variety of workloads and show that EAF-cache provides better overall system performance than all of them (21% compared to the baseline LRU replacement policy and 8% compared to the best previous mechanism, SHIP [54], for a 4-core system). We also show that EAF-cache is orthogonal to improvements in cache replacement policy, by evaluating it in conjunction with two different replacement policies.

2 The Evicted-Address Filter

As mentioned before, our **goal** in this work is to devise a mechanism to prevent both cache pollution and thrashing. Unlike prior approaches, instead of predicting the reuse behavior of a missed cache block indirectly using program counter, memory region or application behavior, our approach predicts the reuse behavior of a missed block based on *its own past behavior*. This can be achieved by remembering the history of reuse behavior of every block based on its past accesses. However, keeping track of the reuse behavior of *every* cache block in the system is impractical due to the associated high storage overhead.

In this work, we make an observation that leads to a simple per-block reuse prediction mechanism to address cache pollution. We first describe this observation and our basic mechanism. We show that a naïve implementation of our basic mechanism 1) has high storage and power overhead, and 2) does not address thrashing. We then describe an implementation of our mechanism using a Bloom filter [6], which addresses both of the above issues.

2.1 Addressing Cache Pollution

> **Observation**: *If a cache block with high reuse is prematurely evicted from the cache, then it will likely be accessed soon after eviction. On the other hand, a cache block with little or no reuse will likely not be accessed for a long time after eviction.*

The above observation indicates that to distinguish blocks with high reuse from those with low reuse, it is sufficient to keep track of a set of recently evicted blocks. Based on this, our mechanism augments a cache with a structure that tracks the *addresses* of recently evicted blocks in a FIFO manner. We call this structure the **Evicted-Address Filter** (EAF). On a cache miss, if the missed block address is present in the EAF, it is likely that the block was prematurely evicted from the cache. Therefore, the cache predicts the block to have high reuse. On the other hand, if the missed block address is not present in the EAF, then either the block is accessed for the first time or it was evicted from the cache a long time ago. In both cases, the cache predicts the block to have low reuse.

Depending on the replacement policy used by the cache, when a missed block's address is present in the EAF, the block (predicted to have high reuse) is inserted with a high priority, which keeps it in the cache for a long period – e.g., at the most-recently-used (MRU) position for the conventional LRU replacement policy. In this case, the corresponding address is also removed from the EAF, as it is no longer a recently evicted block address. This also allows the EAF to track more blocks, and thus make potentially better reuse predictions. When a missed block address is not present in the EAF, the block (predicted to have low reuse) is inserted with a low priority such that it is less likely to disturb other blocks in the cache – e.g., at the LRU position.

The size of the EAF – i.e., the number of recently evicted block addresses it can track – determines the boundary between blocks that are predicted to have high reuse and those that are predicted to have

low reuse. Intuitively, blocks that will be reused in the cache should be predicted to have high reuse and all other blocks should be predicted to have low reuse. For this purpose, we set the size of the EAF to be the same as the number of blocks in the cache. Doing so ensures that if any block that can be reused gets prematurely evicted, it will be present in the EAF. This also ensures any block with a large reuse distance will likely not be present in the EAF. Section 7.4 analyzes the effect of varying the size of the EAF.

In summary, the EAF keeps track of as many recently evicted block addresses as the number of blocks in the cache. When a block gets evicted from the cache, the cache *inserts* its address into the EAF. On a cache miss, the cache *tests* if the missed block address is present in the EAF. If yes, it *removes* the address from the EAF and inserts the block into the cache set with a high priority. Otherwise, it inserts the block with a low priority. When the EAF becomes full, the cache *removes* the least-recently-evicted block address from the EAF (in a FIFO manner).

Although the EAF-based mechanism described above, which we refer to as the *plain-EAF*, can distinguish high-reuse blocks from low-reuse blocks, it suffers from two problems: 1) a naïve implementation of the plain-EAF (a set-associative tag store similar to the main tag store) has high storage and power overhead, and 2) the plain-EAF does not address thrashing.

> **Observation**: *Implementing EAF using a Bloom filter addresses the first problem, as a Bloom filter has low storage and power overhead. In addition, we observe that a Bloom-filter-based EAF implementation also mitigates thrashing.*

We defer a detailed description of the Bloom filter and how it enables a low-overhead implementation of EAF to Section 3. We first provide a deeper understanding of why the plain-EAF suffers from the thrashing problem (Section 2.2) and how a Bloom-filter-based EAF implementation mitigates thrashing (Section 2.3).

2.2 The Problem with Large Working Sets

Consider an application that accesses a working set larger than the cache size in a cyclic manner. In the context of EAF, there are two possible cases.

Case 1: The working set is larger than the aggregate size of blocks tracked by the cache and EAF together. In this case, *no* missed block's address will be present in the EAF. Therefore, all missed blocks will be predicted to have low reuse and inserted into the cache with a low priority. This leads to cache under-utilization as no block is inserted with a high priority, even though there is reuse in the working set.[1] In this case, we would like to always retain a fraction of the working set in the cache as doing so will lead to cache hits at least for that fraction.

Case 2: The working set is smaller than the aggregate number of blocks tracked by both the cache and EAF together. In this case, every missed block's address will be present in the EAF. As a result, every missed block will be predicted to have high reuse and inserted into the cache with a high priority. This will lead to blocks of the application evicting each other from the cache. However, similar to the previous case, we would like to always retain a fraction of the working set in the cache to prevent this problem.

The above discussion indicates that simply using the plain-EAF for predicting the reuse behavior of missed blocks would degrade cache performance when the working set is larger than the cache because using the plain-EAF causes cache under-utilization (Case 1) and does not address thrashing (Case 2).

To address the cache under-utilization problem (Case 1), we insert blocks predicted to have low reuse with the bimodal insertion policy (BIP) [35], as opposed to always inserting them with a low priority. BIP

[1] We assume that the cache is already filled with some other data.

inserts a small fraction of blocks with a high priority and the remaining blocks with a low priority, retaining a fraction of the large working set in the cache. As a result, using BIP for low-reuse blocks allows EAF to adapt to working sets larger than the cache size and the EAF size combined. In the following section, we describe our solution to address cache thrashing (Case 2).

2.3 Addressing Cache Thrashing

The reason why the plain-EAF suffers from thrashing is that it does not have any mechanism to prevent a large number of high-reuse blocks from getting into the cache with a high priority. To mitigate thrashing, the plain-EAF should be augmented with a mechanism that can restrict the number of blocks predicted to have high reuse. Such a mechanism would ensure that not all blocks of a large working set are inserted into the cache with a high priority, and thereby prevent them from evicting each other from the cache. We show that implementing EAF using a Bloom filter achieves this goal.

EAF using a Bloom Filter: Note that the EAF can be viewed as a *set* of recently evicted block addresses. Therefore, to reduce its storage cost, we implement it using a Bloom filter, which is a *compact* representation of a set, in this case, a set of addresses. However, a Bloom filter does not perfectly match the requirements of the plain-EAF. As summarized in Section 2.1, the plain-EAF requires three operations: *insertion*, *testing* and *removal* of an address. Although the Bloom filter supports the insertion and testing operations, it does not support removal of an individual address (Section 3.1). Therefore, to make the plain-EAF implementable using a Bloom filter, we slightly modify the plain-EAF design to eliminate the need to remove an individual address from the EAF. As described in Section 2.1, there are two cases when an address is removed from the EAF. First, when a missed block address is present in the EAF, the cache removes it from the EAF. In this case, we propose to simply *not remove* the address from the EAF. Second, when the EAF becomes full, the least-recently-evicted address is removed from the EAF, in a FIFO manner. In this case, we propose to *clear* the EAF completely, i.e. remove all addresses in the EAF, as the Bloom filter supports such a *clear* operation (Section 3.1). With these two changes, the EAF design becomes amenable for low-cost implementation using a Bloom filter.

Serendipitously, we also found that the changes we make to the plain-EAF design to enable its implementation with a Bloom filter has the benefit of mitigating thrashing. To see why this happens, let us examine the effect of the two changes we make to the plain-EAF. First, *not removing* a cache block address from the EAF ensures that the EAF becomes full even when a thrashing working set fits into the cache and the EAF together. Second, once the EAF becomes full, its gets *cleared*. These two changes together enable the periodic clearing of the EAF. Such periodic clearing results in only a fraction of the blocks of a thrashing working set to get predicted as high-reuse blocks, retaining only that fraction in the cache. This improves the cache hit-rate for such a working set, and thereby mitigates thrashing.

In summary, we propose three changes to the plain-EAF to handle working sets larger than the cache size: 1) insert low-reuse blocks with the *bimodal insertion policy*, 2) *not remove* a missed block address from the EAF, when it is present in the EAF, and 3) *clear* the EAF when it becomes full. We evaluate and analyze the effect of each of these three changes in Section 7.4. Hereafter, we use "EAF" to refer to an EAF with all the above proposed changes. In Section 7.7, we evaluate other possible changes to the plain-EAF design that also mitigate thrashing. We find that those designs perform similarly to the EAF design we proposed in this section.

2.4 Putting it All Together

Figure 1 summarizes our EAF-based cache management mechanism. There are three operations involving the EAF. First, when a block is evicted from the cache, the cache *inserts* the block's address into the EAF ❶. Second, on a cache miss, the cache *tests* whether the missed block address is present in the EAF ❷. If so, the cache inserts the missed block with a high priority, otherwise it inserts the block with the bimodal policy. Third, when the EAF becomes full – i.e., when the number of addresses in the EAF is the same as the number of blocks in the cache – the cache clears the EAF ❸.

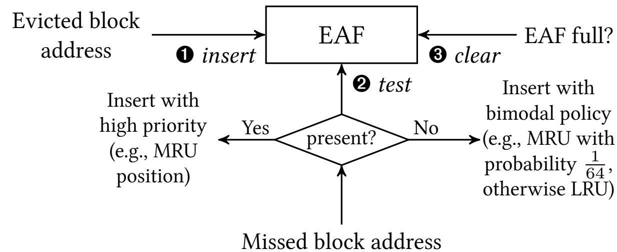


Figure 1: EAF: Final mechanism.

2.5 Handling LRU-friendly Applications

When a block with high reuse is accessed for the first time, it will not be present in the EAF. In this case, the EAF will falsely predict that the block has low reuse. Hence, the block will likely get inserted with a low priority (due to BIP) and get evicted from the cache quickly. As a result, for an LRU-friendly application – i.e., one whose most recently accessed blocks have high reuse – EAF-cache incurs one additional miss for a majority of blocks by not inserting them with high priority on their first access. In most multi-core workloads with LRU-friendly applications, we find that this does not impair performance as the cache is already filled with useful blocks. However, we observe that, for some workloads with *all* LRU-friendly applications, EAF-cache performs considerably worse than prior approaches because of the additional mispredictions.

To address this problem, we use set-dueling [35, 36] to determine if the system as a whole benefits from an always-high-priority insertion policy. The cache chooses two groups of 32 random sets. A missed block that belongs to a set of the first group is always inserted with the priority chosen based on the EAF prediction. A missed block that belongs to a set of the second group is always inserted with high priority. A saturating counter is used to keep track of which group incurs fewer misses. Blocks that do not belong to either of the groups are inserted with the policy that leads to fewer misses. We call this enhancement *Dueling-EAF* (*D-EAF*) and evaluate its performance in Section 7. Our results indicate that D-EAF mitigates the performance loss incurred by EAF-cache for workloads with all LRU-friendly applications and does not significantly affect performance of other workloads.

3 Practical Implementation

As described in Section 2.3, the EAF is designed to be implementable using a Bloom filter [6]. In this section, we describe the problems with a naïve implementation of the EAF that forced us to consider alternate implementations. We follow this with a detailed description of the hardware implementation of a Bloom

filter, which has much lower hardware overhead and complexity compared to the naïve implementation. Finally, we describe the implementation of EAF using a Bloom filter.

A naïve implementation of the EAF is to organize it as a set-associative tag store similar to the main tag store. Each set of the EAF can keep track of addresses of the most recently evicted blocks of the corresponding cache set. However, such an implementation is not desirable for two reasons. First, it incurs a large storage overhead as it requires a separate tag store of the same size as the main tag store. Second, testing whether an address is present in the EAF requires an associative look-up. Although the test operation is not on the critical path, associative look-ups consume additional power.

3.1 Bloom Filter

A Bloom filter is a probabilistic data structure used as a compact representation of a set. It allows three simple operations: 1) *insert* an element into the set, 2) *test* if an element is present in the set, and 3) *remove all* the elements from the set. The *test* operation can have false positives – i.e., the Bloom filter can falsely declare that an element is present in the set even though that element might never have been inserted. However, the false positive rate can be controlled by appropriately choosing the Bloom filter parameters [6].

The hardware implementation of a Bloom filter consists of a bit-array with m bits and a set of k hash functions. Each hash function maps elements (e.g., addresses) to an integer between 0 and $m-1$. Figure 2 shows one such implementation with $m = 16$ and $k = 2$. To insert an element into the set, the Bloom filter computes the values of all the k hash functions on the element, and sets the corresponding bits in the bit-array – e.g., in the figure, inserting the element x sets the bits at locations 1 and 8. To test if an element is present in the set, the Bloom filter computes the values of all the k hash functions on the element, and checks if *all* the corresponding bits in the bit-array are set. If so, it declares that the element is present in the set – e.g., $test(x)$ in the figure. If any of the corresponding bits is not set, the filter declares that the element is not present in the set – e.g., $test(w)$ in the figure. Since the hash functions can map different elements to the same bit(s), the bits corresponding to one element could have been set as a result of inserting other elements into the set – e.g., in the figure, the bits corresponding to the element z are set as a result of inserting elements x and y. This is what leads to a *false positive* for the test operation. Finally, to remove all the elements from the set, the Bloom filter simply resets the entire bit-array. Since multiple elements can map to the same bit, resetting a bit in the Bloom filter can potentially remove multiple elements from it. This is the reason why an individual element cannot be removed from a Bloom filter.

Since a Bloom filter does not directly store elements, it generally requires much smaller storage than a naïve implementation of a set. In addition, all operations on a Bloom filter require only indexed lookups into the bit-array, which are more energy efficient than associative lookups.

3.2 EAF Using a Bloom Filter and a Counter

As described in Section 2.3, the EAF is designed so that the three operations associated with it, namely, *insert*, *test*, and *clear*, match the operations supported by the Bloom filter. In addition to the Bloom filter required to implement the EAF, our mechanism also requires a counter to keep track of the number of addresses currently inserted into the Bloom filter. This is required to determine when the EAF becomes full, so that the cache can clear it.

Figure 3 shows the operation of an EAF implemented using a Bloom filter and a counter. Initially, both the Bloom filter and the counter are reset. When a cache block gets evicted, the cache inserts the block's address into the Bloom filter and increments the counter. On a cache miss, the cache tests if the missed

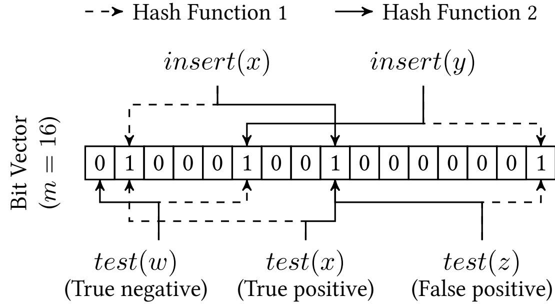


Figure 2: Hardware implementation of a Bloom filter using a bit vector of size 16 and two hash functions. The insert operations are performed before the test operations.

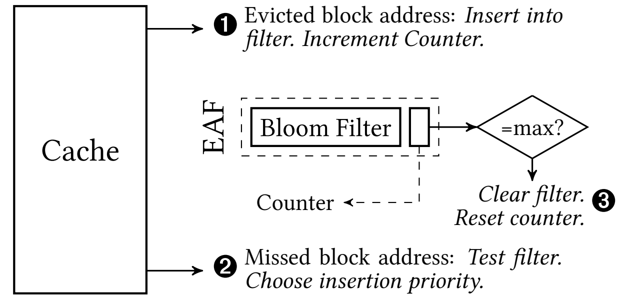


Figure 3: EAF implementation using a Bloom filter and a counter. The figure shows the three events that trigger operations on the EAF.

block address is present in the Bloom filter and chooses the insertion policy based on the result. When the counter reaches the number of blocks in the cache, the cache resets both the counter and the Bloom filter.

For our Bloom filter implementation, we use the H_3 class of hash functions [41]. These functions require only XOR operations on the input address and, hence, are simple to implement in hardware.

3.3 Hardware Implementation Cost

Implementing EAF on top of a conventional cache requires three additional hardware structures: 1) a *bit-array* for the Bloom filter, 2) a *counter* that can count up to the number of blocks in the cache, and 3) peripheral logic structure to determine the insertion priority of each missed cache block. The primary source of storage overhead is the bit-array required to implement the Bloom filter.

The number of bits required for the bit-array scales linearly with the maximum number of addresses expected to be stored in the Bloom filter, which in our case, is same as the number of blocks in the cache (C) (as described in Section 2.1). We set the number of bits in the bit-array to be αC, where α is the average number of bits required per address stored in the Bloom filter. We will discuss the trade-offs of this parameter shortly. For a system where cache block size is B and the size of each cache tag entry is T, the storage overhead of EAF compared to the cache size is given by:

$$\frac{\text{Bloom filter size}}{\text{Cache size}} = \frac{\alpha C}{(T+B)C} = \frac{\alpha}{T+B} \tag{1}$$

The false positive probability (p) of the Bloom filter is given by (as shown in [12]):

$$p = 2^{-\alpha \ln 2} \tag{2}$$

Since the false positive rate of the Bloom filter directly affects the accuracy of EAF predictions, the parameter α presents a crucial trade-off between the storage overhead of EAF and its performance improvement. From Equations (1) and (2), it can be seen that as α increases, the storage overhead also increases, but the false positive rate decreases. We find that, for most systems we evaluate, setting α to 8 provides the performance benefits of an EAF with no false positives (Section 7.4). For a 2-MB last-level cache with 64-byte block size, this amounts to a modest 1.47% storage overhead. Our evaluations show that, for a 4-core system, EAF-cache provides 21% performance improvement compared to the baseline.

4 Advantages & Disadvantages

As we will show in Section 7, EAF-cache, which addresses both pollution and thrashing, provides the best performance compared to five prior approaches. In addition, EAF-cache has three advantages.

Ease of hardware implementation: As Figure 3 shows, EAF-cache can be implemented using a Bloom filter and a counter. Bloom filters have low hardware overhead and complexity. Several previous works [8, 29, 33, 40] have proposed the use of Bloom filters in hardware for various applications. Therefore, our EAF implementation using a Bloom filter incurs low design and implementation cost.

No modifications to cache design: Since both the Bloom filter and the counter are outside the cache, our mechanism does not require any modifications to the cache. This leads to low design and verification effort for integrating EAF into existing processor designs.

No modifications to cache hit operation: Finally, EAF operates only on a cache miss. The cache hit operation remains unchanged. Therefore, EAF can be combined with mechanisms that improve cache performance by monitoring only cache hits – e.g., the cache replacement policy. As an example, we show that EAF-cache provides better performance with an improved cache replacement policy (Section 7.3).

One shortcoming of EAF-cache is its storage overhead compared to certain other prior approaches [9, 16, 17] to address cache thrashing or pollution individually (see Table 1). However, as we show in Section 7, EAF-cache significantly outperforms these techniques, thereby justifying its additional storage overhead.

5 Qualitative Comparison to Prior Work

We qualitatively compare our proposed mechanism, the EAF-cache, with five state-of-the-art high-performance mechanisms to address cache pollution or thrashing. The fundamental difference between the EAF-cache and prior approaches is that *unlike the EAF-cache, previous mechanisms do not concurrently address the negative impact of both pollution and thrashing.* As a result, they do not obtain the best performance for all workloads. Table 1 lists the previous mechanisms that we compare with the EAF-cache. The table indicates whether each mechanism addresses pollution and thrashing, along with its implementation complexity. We now describe each mechanism individually.

Thread-Aware Dynamic Insertion Policy (TA-DIP) [16] addresses the thrashing problem by determining thrashing at a thread granularity. It does so by using set-dueling [35, 36] to determine if a thread incurs fewer misses with the conventional LRU policy or the bimodal insertion policy (BIP). *All* blocks of a thread are inserted with the policy that has fewer misses. As a result, TA-DIP cannot distinguish between high-reuse blocks and low-reuse blocks within a thread. When following the LRU policy, this can lead to low-reuse blocks polluting the cache. When following BIP, this can lead to high-reuse blocks getting repeatedly inserted with the bimodal insertion policy, causing a low cache hit rate. As a result, TA-DIP does not provide the best performance when blocks within a thread have different reuse behavior.

Thread-Aware Dynamic Re-Reference Interval Prediction (TA-DRRIP) [17] improves upon TA-DIP by using a better replacement policy than LRU, RRIP. Unlike the LRU policy, which inserts all incoming blocks with the highest priority (MRU position), RRIP inserts all incoming blocks with a lower priority. This allows RRIP to reduce the performance degradation caused by low-reuse blocks, when compared with LRU. However, as identified by later work [54], TA-DRRIP does not completely address the pollution problem as it monitors the reuse behavior of a block *after* inserting the block into the cache. In addition, like TA-DIP,

Mechanism	Addresses Pollution?	Addresses Thrashing?	Storage Overhead	Changes to Cache?	Modifies Hit Behavior?
TA-DIP [16]	No	Yes	2 bytes per application	No changes to cache	No changes to cache hits
TA-DRRIP [17]	Partly	Yes	2 bytes per application	No changes to cache	No changes to cache hits
SHIP [54]	Yes	No	4-16KB prediction table*, 1.875KB instruction tags	Requires storing program counter in the tag store	Updates to the prediction table
RTB [19]	Yes	No	3KB for a 1024 entry MAT*	No changes to cache	Updates to the memory access table
MCT [9]	Yes	No	1KB miss classification table	No changes to cache	No changes to cache hits
EAF-Cache	Yes	Yes	16KB for Bloom filter	No changes to cache	No changes to cache hits

Table 1: Storage overhead and implementation complexity of different mechanisms (for a 1MB 16-way set-associative cache with 64B block size). *In our evaluations, we use an infinite size table for both SHIP and RTB to eliminate interference caused by aliasing.

since TA-DRRIP operates at a thread (rather than a block) granularity, it suffers from the shortcomings of TA-DIP described before.

Signature-based HIt Prediction (SHIP) [54] addresses the pollution problem by using program counter information to distinguish blocks with low reuse from blocks with high reuse. The key idea is to group blocks based on the program counter that loaded them into the cache and learn the reuse behavior of each group using a table of counters. On a cache miss, SHIP indexes the table with the program counter that generated the miss and uses the counter value to predict the reuse behavior of the missed cache block. SHIP suffers from two shortcomings: 1) blocks loaded by the same program counter often may not have the same reuse behavior. In such cases, SHIP leads to many mispredictions, 2) when the number of blocks predicted to have high reuse exceeds the size of the cache, SHIP cannot address the resulting cache thrashing problem.[2]

Run-time Cache Bypassing (RTB) [19] addresses cache pollution by predicting reuse behavior of a missed block based on the memory region to which it belongs. RTB learns the reuse behavior of 1KB memory regions using a table of reuse counters. On a cache miss, RTB compares the reuse behavior of the missed block with the reuse behavior of the to-be-evicted block. If the missed block has higher reuse, it is inserted into the cache. Otherwise, it bypasses the cache. As RTB's operation is similar to that of SHIP, it suffers from similar shortcomings as SHIP.[3]

Miss Classification Table (MCT) [9] also addresses the cache pollution problem. MCT keeps track of one most recently evicted block address for each set in the cache. If a subsequent miss address matches this evicted block address, the miss is classified as a conflict miss. Otherwise, the miss is classified as a capacity miss. MCT inserts only conflict-missed blocks into the cache, anticipating that they will be reused. All other blocks are inserted into a separate buffer. MCT has two shortcomings. First, for a highly associative last-level cache, keeping track of only one most recently evicted block per set leads to many conflict misses getting falsely predicted as capacity misses, especially in multi-core systems with shared caches. Second, naïvely extending MCT to keep track of more evicted blocks can lead to the number of predicted-conflict-

[2]We also implemented single-usage block prediction [34] (SU), which also uses the program counter to identify reuse behavior of missed blocks. Both SU and SHIP provide similar performance.

[3]The SHIP paper [54] also evaluates mechanisms that use memory regions and other signatures to group blocks. The paper identifies that program counter-based grouping provides the best results. We reach a similar conclusion based on our experimental results.

Core		4 Ghz processor, in-order x86
L1-D Cache		32KB, 2-way associative, LRU replacement policy, single cycle latency, 64B block size
Private L2 Cache		256KB, 8-way associative, LRU replacement policy, latency = 8 cycles, 64B block size
L3 Cache	1-core	1 MB, 16-way associative, latency = 21 cycles, 64B block size
	2-core	Shared, 1MB, 16-way associative, latency = 21 cycles, 64B block size
	4-core	Shared, 2MB, 16-way associative, latency = 28 cycles, 64B block size
Main memory		DDR2 parameters, Row hits = 168 cycles, Row conflicts = 408 cycles, 4 Banks, 8 KB row buffers

Table 2: Main configuration parameters used for our simulations

miss blocks to exceed the number of blocks in the cache. In such a scenario, MCT cannot address the resulting thrashing problem.

In contrast to previous approaches, our proposed mechanism, the **EAF-Cache**, is designed to address *both* cache pollution and thrashing with a single structure, the Evicted-Address Filter. Instead of indirectly predicting the reuse behavior of a cache block using the behavior of the application, program counter or memory region as a proxy, the EAF-cache predicts the reuse behavior of each missed cache block based on *the block's own past behavior*. As described in Section 2.3, the EAF-cache also mitigates thrashing by retaining only a fraction of blocks of a thrashing working set in the cache. As a result, EAF-cache adapts to workloads with varying working set characteristics, and as we will show in our quantitative results, it significantly outperforms prior approaches.

6 Methodology

We use an in-house event-driven x86 simulator for our evaluations. Instruction traces were collected by running the benchmarks on top of the Wind River Simics full system simulator [4], which are then fed to our simulator's core model. Our framework faithfully models all memory-related processor-stalls. All systems use a 3-level cache hierarchy similar to some modern architectures [2, 22]. The L1 and L2 caches are private to individual cores and the L3 cache is shared across all the cores. We do not enforce inclusion in any level of the hierarchy. All caches use a 64B block size. Writebacks do not update the replacement policy state.

Other major simulation parameters are provided in Table 2. The cache sizes we use for our primary evaluations (1MB for 2-core and 2MB for 4-core) were chosen to account for the small working set sizes of most of our benchmarks. Section 7.5 presents results with larger cache sizes (up to 16MB). For all mechanisms, except baseline LRU and DIP, the last-level cache (LLC) uses the re-reference interval prediction replacement policy [17]. For our EAF proposals, we assume that the operations on the EAF can be overlapped with the long latency of memory access (note that all EAF operations happen only on a LLC miss).

For evaluations, we use benchmarks from the SPEC CPU2000 and CPU2006 suites, three TPC-H queries, one TPC-C server and an Apache web server. All results are collected by running a representative portion of each benchmark for 500 million instructions [48]. We classify benchmarks into nine categories based on their cache sensitivity (low, medium, or high) and memory intensity (low, medium, or high). For measuring cache sensitivity, we run the benchmarks with a 1MB last-level cache and a 256KB last-level cache and use

Name	L2-MPKI		Sens.		Name	L2-MPKI		Sens.		Name	L2-MPKI		Sens.		Name	L2-MPKI		Sens.	
ammp	5.76	M	36%	H	fma3d	1.14	L	5%	M	lucas	3.11	L	0%	L	vpr	6.13	M	46%	H
applu	1.92	L	2%	L	galgel	7.94	M	17%	M	mcf	49.58	H	26%	H	wupwise	1.33	L	1%	L
art	40.56	H	52%	H	gcc	4.08	L	3%	M	mgrid	3.14	L	5%	M	xalancbmk	10.89	H	16%	M
astar	25.49	H	6%	M	GemsFDTD	16.57	H	1%	L	milc	12.33	H	0%	L	zeusmp	5.77	L	1%	L
bwaves	15.03	H	0%	L	gobmk	1.92	L	2%	L	omnetpp	12.73	H	10%	M	**Server Benchmarks**				
bzip2	7.01	M	32%	H	h264ref	1.52	L	5%	M	parser	2.0	L	18%	H	apache20	5.8	L	9%	M
cactusADM	4.4	L	8%	M	hmmer	2.63	L	2%	L	soplex	25.31	H	18%	H	tpcc64	11.48	H	31%	H
dealII	1.51	L	9%	M	lbm	24.64	H	1%	L	sphinx3	14.86	H	9%	M	tpch2	17.02	H	31%	H
equake	9.22	M	6%	M	leslie3d	14.02	H	7%	M	swim	17.7	H	46%	H	tpch6	3.93	L	23%	H
facerec	4.61	L	18%	H	libquantum	14.31	H	1%	L	twolf	10.21	M	56%	H	tpch17	13.97	H	26%	H

Table 3: Benchmark classification based on intensity (L2-MPKI) and sensitivity (Sens.). (L - low, M - Medium, H - High). Intensity: L2-MPKI < 5 (Low); > 10 (High); Rest (Medium). Sensitivity: Perf. Degradation < 5% (Low); > 18% (High); Rest (Medium)

the performance degradation as a metric. We define a benchmark's memory intensity as its misses per 1000 instructions (MPKI) on a 256KB L2 cache. We do not consider benchmarks with L2-MPKI less than one for our studies as they do not exert significant pressure on the last-level cache. Table 3 shows the memory intensity and cache sensitivity of different benchmarks used in our evaluations along with the criteria used for classifying them.

We evaluate multi-programmed workloads running on 2-core and 4-core CMPs. We generate nine categories of multi-programmed workloads with different levels of aggregate intensity (low, medium, or high) and aggregate sensitivity (low, medium, or high).[4] We evaluate the server benchmarks separately with ten 2-core and five 4-core workload combinations. In all, we present results for 208 2-core and 135 4-core workloads.[5]

We evaluate system performance using the weighted speedup metric [49], which is shown to correlate with the system throughput [11]. We also evaluate three other metrics, namely, instruction throughput, harmonic speedup [28], and maximum slowdown [25, 26], in Section 7.6 for completeness.

$$\text{Weighted Speedup} \quad = \quad \sum_i \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{alone}}}$$

7 Results

7.1 Single-core Results

Figure 4 compares the performance, instructions per cycle (IPC), for different benchmarks with different mechanisms.[6] The percentages on top of the bars show the reduction in the last-level cache misses per kilo instruction of D-EAF compared to LRU. We draw two conclusions from the figure.

First, D-EAF, with its ability to address both pollution and thrashing, provides performance comparable to the better of the previous best mechanisms to address pollution (SHIP) or thrashing (DRRIP) for most

[4]We compute aggregate intensity/sensitivity of a workload as the sum of individual benchmark intensities/sensitivities (low = 0, medium = 1, high = 2).

[5]Appendices A and B provide details of individual workloads.

[6]For clarity, results for DIP and RTB are excluded from Figures 4 and 6 (left) as their performance improvements are lower compared to the other mechanisms. We also don't present single-core results for benchmarks where all mechanisms perform within 1% of the baseline LRU.

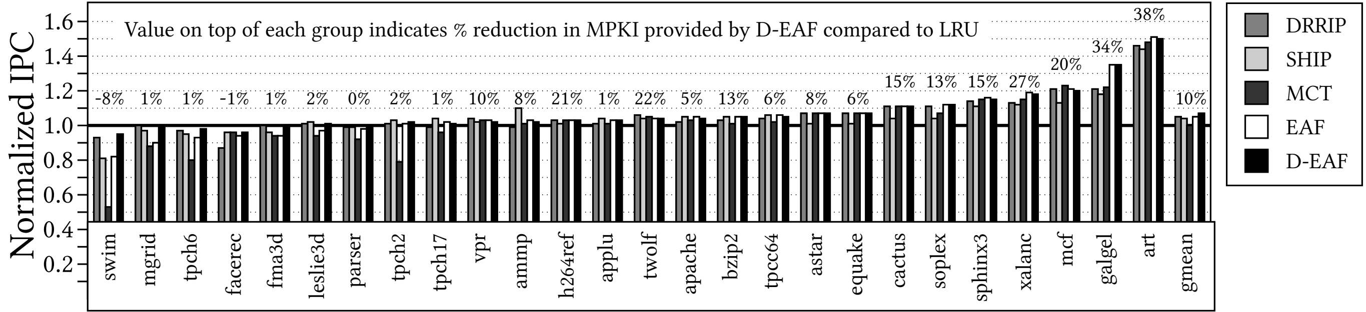


Figure 4: System performance: Single-core system

benchmarks. On average, D-EAF provides the best performance across all benchmarks (7% IPC improvement over LRU). In fact, except for benchmarks *swim*, *facerec* and *parser*, D-EAF always reduces the miss rate compared to the LRU policy. As the figure shows, MCT considerably degrades performance for several benchmarks. This is because MCT keeps track of only one most recently evicted block address per set to identify conflict misses. This leads to many conflict-miss blocks to get falsely predicted as capacity misses (as described in Section 5). As a result, such blocks are inserted with low priority, leading to poor performance. This effect becomes worse in a multi-core system, where there is interference between concurrently running applications.

Second, as expected, for LRU-friendly benchmarks like *swim* and *mgrid*, EAF degrades performance considerably. This is because, when a block is accessed for the first time, EAF predicts it to have low reuse. However, the most recently accessed blocks in these benchmarks have high reuse. Therefore, EAF incurs one additional miss for most blocks. D-EAF identifies this problem and inserts all blocks of such applications with high priority, thereby mitigating the performance degradation of EAF. For example, in case of *swim*, D-EAF reduces the MPKI by 18% compared to EAF. We conclude that D-EAF, with the ability to dynamically adapt to different benchmark characteristics, provides the best average performance.

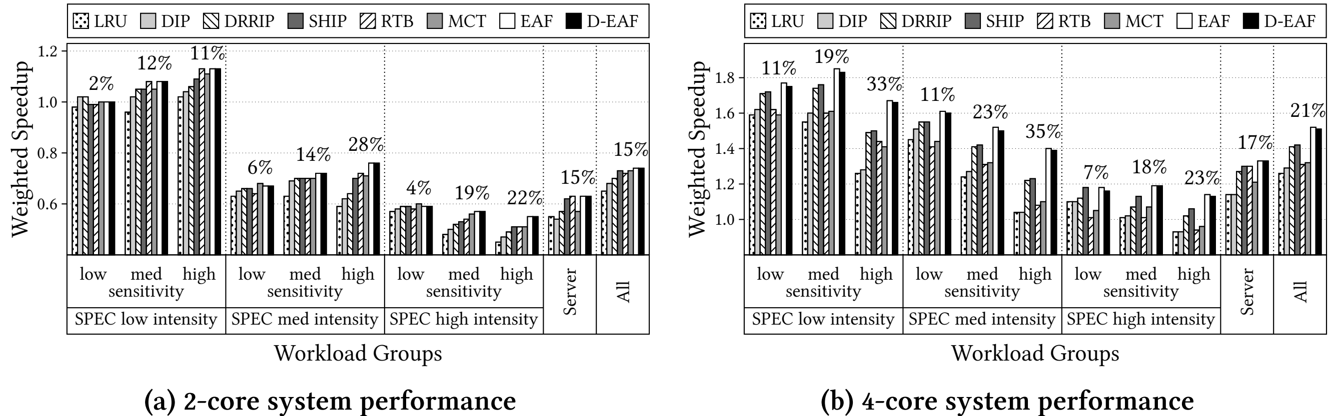
7.2 Multi-core Results

In multi-core systems, the shared last-level cache is subject to varying access patterns from the concurrently-running applications. In such a scenario, although pollution or thrashing caused by an application may not affect its own performance, they can adversely affect the performance of the concurrently-running applications. Therefore, the cache management mechanism should efficiently handle all the access patterns to provide high system performance. The EAF-cache with its ability to concurrently handle both cache pollution and thrashing provides better cache utilization than prior approaches that were designed to handle only one of the two problems. As a result, we find that the EAF-cache significantly outperforms prior approaches for multi-programmed workloads (on both 2-core and 4-core systems).

Performance by workload categories: Figure 5 shows the absolute weighted speedup for our 2-core and 4-core systems grouped based on different workload categories (as described in Section 6). The percentage on top of the bars show the improvement in weighted speedup of D-EAF compared to the LRU policy. For SPEC workloads, as expected, overall system performance decreases as the intensity of the workloads increases from low to high.[7] Regardless, both EAF and D-EAF outperform other prior approaches for all workload categories (21% over LRU and 8% over the best previous mechanism, SHIP). The figure also shows

[7] Our server benchmarks have little scope for categorization. Therefore, we present average results for all server workloads.

that within each intensity category, the performance improvement of our mechanisms increases with increasing cache sensitivity of the workloads. This is because the negative impact of cache under-utilization increases as workloads become more sensitive to cache space. Although not explicitly shown in the figure, the performance improvement of D-EAF over the best previous mechanism (SHIP) also increases as the cache sensitivity of the workloads increases.



(a) 2-core system performance

(b) 4-core system performance

Figure 5: System performance for different workload categories. The value on top of each group indicates % improvement in weighted speedup provided by D-EAF compared to LRU.

For server workloads, D-EAF improves weighted speedup by 17% compared to the LRU policy. These workloads are known to have *scans* [17] (accesses to a large number of blocks with no reuse) which pollute the cache, making pollution the major problem. This is the reason why SHIP and RTB, which were designed to address only cache pollution, are able to perform comparably to D-EAF for these workloads. In contrast, TA-DIP, which is designed to address only cache thrashing, offers no improvement over LRU.

We conclude that our EAF-based approach is better than other prior approaches for both SPEC and server workloads and its benefits improve as workloads become more sensitive to cache space.

Overall performance analysis: Figures 6 and 7 plot the performance improvements of the different approaches compared to the baseline LRU policy for all 2-core and 4-core workloads, respectively. The workloads are sorted based on the performance improvement of EAF. The goal of these figures is *not* to indicate how different mechanisms perform for all workloads. Rather, the figures show that both EAF and D-EAF (thick lines) significantly outperform all prior approaches for most workloads.

Two other observations can be made from these figures. First, there is no consistent trend in the performance improvement of the prior approaches. Different approaches improve performance for different workloads. This indicates that addressing only cache pollution or only cache thrashing is not sufficient to provide high performance for all workloads. Unlike these prior approaches, EAF and D-EAF, with their ability to address both problems concurrently, provide the best performance for almost all workloads. Second, there is no visible performance gap between EAF and D-EAF for the majority of the workloads except for a small set of LRU-friendly 2-core workloads (indicated in the far left of Figure 6), where D-EAF performs better than EAF. This is because, for most workloads (even with LRU-friendly applications), the cache (with only EAF) is already filled with high-reuse blocks. As a result, the additional misses caused by EAF for LRU-friendly applications do not significantly affect performance. But, as evidenced by our results for single-core and the small fraction of LRU-friendly 2-core workloads, D-EAF is more robust in terms of performance than EAF.

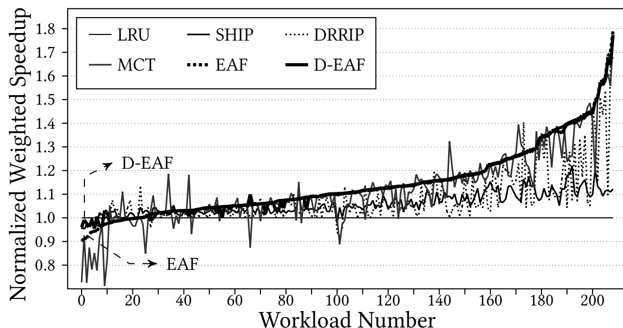


Figure 6: System performance: 2-core

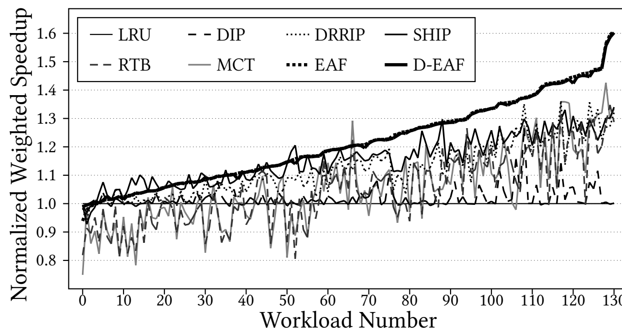
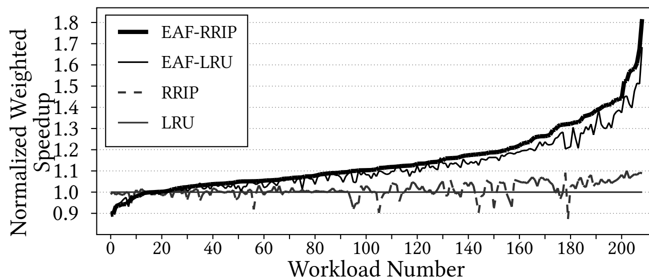


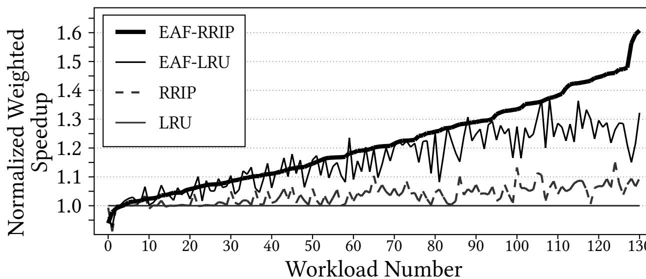
Figure 7: System performance: 4-core

7.3 Interaction with the Replacement Policy

As EAF-cache modifies only the cache insertion policy, it can be used with any cache replacement policy. Figure 8 shows the benefits of augmenting our EAF mechanism to 1) a cache following the LRU replacement policy, and 2) a cache following the RRIP [17] policy for all 4-core workloads. As the figure shows, EAF-cache consistently improves performance in both cases for almost all workloads (11% on average for LRU and 12% for RRIP). We conclude that the benefits of EAF-cache are orthogonal to the benefits of using an improved cache replacement policy.



(a) 2-core Workloads



(b) 4-core Workloads

Figure 8: EAF-cache with different replacement policies

7.4 Effect of EAF Design Choices

EAF size – Number of evicted addresses: The size of the EAF – i.e., the number of evicted addresses it keeps track of – determines the boundary between blocks that are considered to be recently evicted and those that are not. On one hand, having a small EAF increases the likelihood of a high-reuse block getting incorrectly predicted to have low reuse. On the other hand, a large EAF increases the likelihood of a low-reuse block getting incorrectly predicted to have high reuse. As such, we expect performance to first increase as the size of the EAF increases and then decrease beyond some point. Figure 10 presents the results of our experiments studying this trade-off. As indicated in the figure, the performance improvement of EAF peaks when the size of the EAF is approximately the same as the number of blocks in the cache. This concurs with our intuition (Section 2.1) – sizing the EAF to be around the same size as the number of blocks in the cache ensures that a block that can potentially be reused in the cache will likely be predicted to have high reuse while a block with a larger reuse distance will likely be predicted to have low reuse.

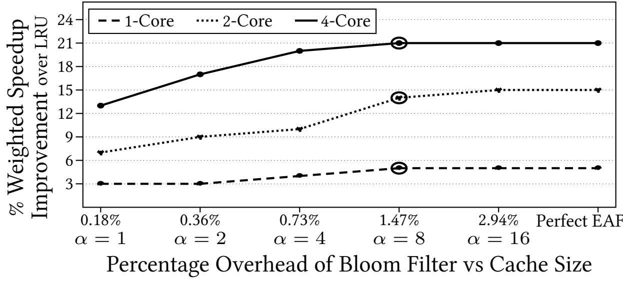


Figure 9: Sensitivity to Bloom filter overhead

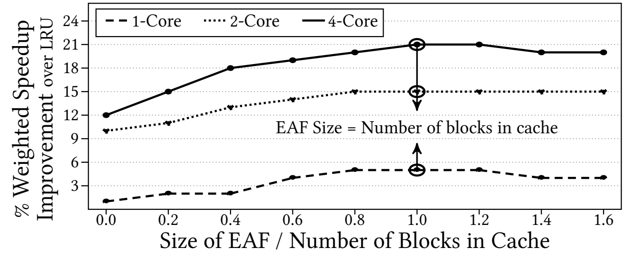


Figure 10: Sensitivity to EAF size

Storage overhead of the Bloom filter: As described in Section 3.3, for a fixed EAF size, the parameter α associated with the Bloom filter presents a trade-off between the false positive rate and the storage overhead of the Bloom filter. A large value of α leads to a low false positive rate but incurs high storage overhead. On the other hand, a small value of α incurs small storage overhead but leads to a high false positive rate. As a false positive can result in an actually low-reuse block to be predicted to have high reuse, the choice of α is critical for performance. Figure 9 shows the effect of varying α on performance, when the EAF size is the same as the number of blocks in the cache. The figure also shows the corresponding storage overhead of the Bloom filter with respect to the cache size. As α increases, performance improves because false positive rate decreases. However, almost all of the potential performance improvement of having a perfect EAF (EAF with no false positives) is achieved with $\alpha = 8$ (1.47% storage overhead).

Effect of EAF policies for cache thrashing: In Section 2.3, we proposed three changes to the plain-EAF, to address cache thrashing: 1) using the *bimodal insertion policy* (BIP) for low-reuse blocks (B), 2) *not removing* a block address when it is present in the EAF (N), and 3) *clearing* the EAF when it becomes full (C). Figure 11 shows the performance improvement of adding all combinations of these policies to the *plain-EAF* (described in Section 2.1). Two observations are in order.

First, the *not remove* (+N) and the *clear* (+C) policies individually improve performance, especially significant in the 4-core system. However, applying both policies together (+NC) provides significant performance improvement for *all* systems. This is because, when the working set is larger than the cache size but smaller than the aggregate size of the blocks tracked by the cache and EAF together, the *not remove* (+N) policy is required to ensure that the EAF gets full, and the *clear* (+C) policy is required to clear the EAF when it becomes full. Second, the best performance improvement is achieved for all systems when *all* the three policies are applied to the EAF (+NCB), as doing so improves the hit-rate for both categories of large working sets (as discussed in Section 2.2).

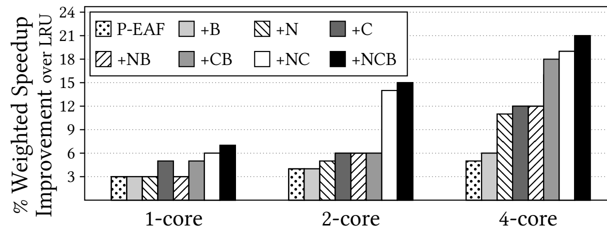
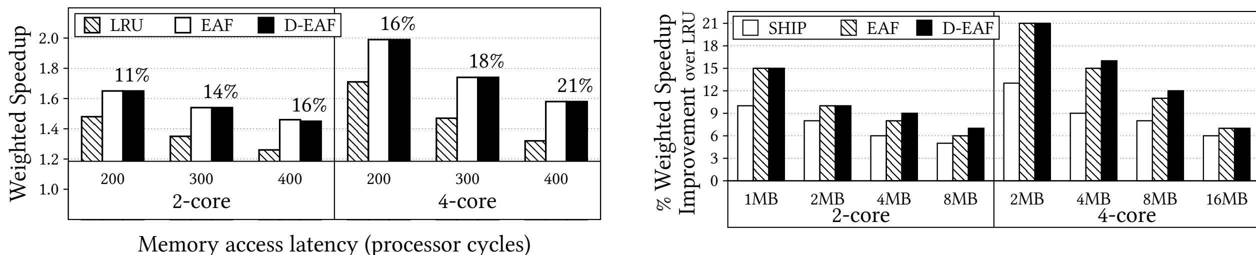


Figure 11: Effect of changes to plain-EAF (P-EAF)

7.5 Sensitivity to System Parameters

Varying Cache Size: Figure 12b plots the performance improvements of SHIP (best previous mechanism), EAF and D-EAF compared to LRU for different cache sizes for multi-core systems. The performance improvement of different mechanisms decreases with increasing cache size. This is expected because with increasing cache size, cache under-utilization becomes less of a problem. However, the EAF-cache provides significant performance improvements even for 8MB/16MB caches, and better performance than SHIP for all cache sizes.

Varying Memory Latency: Figure 12a shows the effect of varying the memory latency. For ease of analysis, we use a fixed latency for all memory requests in these experiments. As expected, system throughput decreases as the memory latency increases. However, the performance improvement of EAF-cache increases with increasing memory latency, making it a more attractive mechanism for future systems, which are likely to have higher memory access latencies due to contention.



(a) EAF-cache with different memory latencies.

(b) EAF-cache with different cache sizes

Figure 12: Sensitivity to system parameters. Left: Memory latency. The value on top of each group indicates % weighted speedup improvement of D-EAF compared to LRU. Right: Cache size.

7.6 Multi-core Systems: Other Metrics

Table 4 shows the percentage improvement of the EAF-cache over the baseline LRU policy and the best previous mechanism, SHIP, on four metrics: weighted speedup [11, 49], instruction throughput, harmonic speedup [28] and maximum slowdown [25, 26]. For both 2-core and 4-core systems, EAF-cache significantly improves weighted speedup, instruction throughput and harmonic speedup, and also considerably reduces the average maximum slowdown. We conclude that EAF-cache improves both system performance and fairness compared to prior approaches.

7.7 Alternate EAF Designs

In this section, we describe two different approaches to mitigate cache pollution and thrashing by making slight modifications to the proposed EAF design: 1) Segmented-EAF, and 2) Decoupled-Clear-EAF. We describe these two designs below and compare their performance to our original EAF-cache design.

Segmented-EAF: The plain-EAF (described in Section 2.1) and the EAF (with the three changes described in Section 2.3) are two ends of a spectrum. While the former removes just one block address when the EAF becomes full, the latter clears the entire EAF when it becomes full. One can think of an approach which clears a fraction of the EAF (say $\frac{1}{2}$) when it becomes full. This can be accomplished by segmenting the EAF into multiple Bloom filters (organized in a FIFO manner) and clearing only the oldest one when the

Metric	2-core		4-core	
	+ LRU	+ SHIP	+ LRU	+ SHIP
Weighted Speedup	15%	6%	21%	8%
Instruction Throughput	14%	6%	22%	7%
Harmonic Speedup	15%	6%	29%	6%
Max. Slowdown Reduction	16%	7%	31%	8%

Table 4: Improvement of EAF on different metrics

EAF becomes full. We call this variant of the EAF *Segmented-EAF*. A Segmented-EAF trades off the ability to mitigate thrashing (number of addresses removed from the EAF) with the ability to mitigate pollution (amount of information lost due to clearing).

Decoupled-Clear-EAF: In EAF-cache, the number of blocks that enter the cache with high priority is reduced by clearing the EAF when it becomes full. However, this approach requires the size of the EAF to be around the same as the number of blocks in the cache, so as to limit the number of blocks predicted to have high-reuse between two EAF-clear events. This restricts the EAF's visibility of recently evicted blocks. However, having a larger visibility can potentially allow EAF to better identify the reuse behavior of more cache blocks. The *Decoupled-Clear-EAF* addresses this visibility problem by decoupling the size of the EAF from determining when the EAF is cleared. In this approach, the EAF keeps track of more addresses than the number of blocks in the cache (say $2\times$). An additional counter keeps track of the number of blocks inserted with high priority into the cache. The EAF is cleared on two events: 1) when it becomes full, and 2) when the number of high-priority blocks reaches the number of blocks in the cache.

Table 5 compares the performance improvement (over baseline LRU) of the above designs with our proposed EAF design. Our results indicate that there is not much performance gap (less than 2%) across all three designs. We conclude that all the three heuristics perform similarly for our workloads. We leave an in-depth analysis of these design points and other potential designs as future work.

Design	1-core	2-core	4-core
EAF	7%	15%	21%
Segmented-EAF	6%	13%	21%
Decoupled-Clear-EAF	6%	14%	21%

Table 5: Comparison of EAF with other design alternatives

8 Related Work

The primary contribution of this paper is the EAF-cache, a low-complexity cache insertion policy that mitigates the negative impact of both cache pollution and thrashing, by determining the reuse behavior of a missed block based on the block's own past behavior. We have already provided qualitative and quantitative comparisons to the most closely related work on addressing pollution or thrashing [9, 16, 17,

19, 34, 54], showing that EAF-cache outperforms all these approaches. In this section, we present other related work.

A number of cache insertion policies have been proposed to prevent L1-cache pollution [43, 52]. Similar to some approaches we have discussed in our paper [19, 34, 54], these mechanisms use the instruction pointer or the memory region to predict the reuse behavior of missed blocks. As we showed, our EAF-based approach provides better performance than these prior approaches as it can address both pollution and thrashing simultaneously.

Prior work has proposed compiler-based techniques (e.g., [10, 42, 53]) to mitigate the negative impact of pollution and thrashing by either modifying the data layout or by providing hints to the hardware – e.g., non-temporal load/store in x86 [14]. However, the scope of such techniques is limited to cases where reuse and/or locality of accesses can be successfully modeled by the programmer/compiler, which often corresponds to array accesses within loop nests. Our EAF-cache proposal can potentially improve performance for a much broader set of memory accesses, and hence complements these compiler-based techniques.

Much prior research (e.g., [13, 17, 23, 27, 39, 46]) has focused on improving the cache replacement policy. Researchers have also proposed mechanisms to improve cache utilization (e.g., [38, 45, 44, 47, 56]) by addressing the set imbalance problem (i.e., certain cache sets suffer many conflict misses while others are under-utilized). The EAF-cache can be combined with any of these mechanisms to further improve performance.

A number of page replacement policies have been proposed to improve virtual memory performance (e.g., [5, 18, 20, 30, 32]). As these mechanisms were designed for software-based DRAM buffer management, they usually employ complex algorithms and assume large amounts of storage. As a result, applying them to hardware caches incurs higher storage overhead and implementation complexity than the EAF-cache.

Jouppi proposed victim caches [21] to improve the performance of direct mapped caches by reducing the latency of conflict misses. A victim cache stores recently evicted cache blocks (including data) in a fully associative buffer. As a result, it has to be a small structure. In contrast, our proposed EAF mechanism is a reuse predictor which keeps track of *only addresses* of a larger number of recently evicted blocks. As such, the two techniques can be employed together.

Cache partitioning is one technique that has been effectively used to improve shared-cache performance [37, 50, 51, 55] or provide QoS in multi-core systems with shared caches [7, 15, 24, 31]. The mechanisms to improve performance use partitioning to provide more cache space to applications that benefit from the additional space. For QoS, the proposed approaches ensure that applications are guaranteed some minimum amount of cache space. Since blocks with low reuse contribute neither to system performance nor to fairness, these mechanisms can be employed in conjunction with the EAF-cache, which can filter out low-reuse blocks to further improve performance or fairness. Evaluation of this is part of our future work.

9 Conclusion

Efficient cache utilization is critical for high system performance. Cache pollution and thrashing reduce cache utilization and consequently, degrade cache performance. We show that prior works do not concurrently address cache pollution and thrashing.

We presented the EAF-cache, a mechanism that mitigates the negative impact of *both* cache pollution and thrashing. EAF-cache handles pollution by keeping track of addresses of recently evicted blocks in a structure called the *Evicted-Address Filter* (EAF) to distinguish high-reuse blocks from low-reuse blocks.

Implementing the EAF using a Bloom filter naturally mitigates the thrashing problem, while incurring low storage and power overhead. Extensive evaluations using a wide variety of workloads and system configurations show that the EAF-cache provides the best performance compared to five different prior approaches.

We conclude that EAF-cache is an attractive mechanism that can handle both cache pollution and thrashing, to provide high performance at low complexity. The concept of EAF provides a substrate that can enable other cache optimizations, which we are currently exploring.

ACKNOWLEDGEMENTS

Many thanks to Chris Wilkerson, Phillip Gibbons, and Aamer Jaleel for their feedback during various stages of this project. We thank the anonymous reviewers for their valuable feedback and suggestions. We acknowledge members of the SAFARI and LBA groups for their feedback and for the stimulating research environment they provide. We acknowledge the generous support of AMD, Intel, Oracle, and Samsung. This research was partially supported by grants from NSF, GSRC, Intel University Research Office, and Intel Science and Technology Center on Cloud Computing. We thank Lavanya Subramanian, David Andersen, Kayvon Fatahalian and Michael Papamichael for their feedback on this paper's writing.

References

[1] AMD Phenom II key architectural features. `http://goo.gl/iQBfK`.

[2] Intel next generation microarchitecture. `http://goo.gl/3eskx`.

[3] Oracle SPARC T4. `http://goo.gl/KZSnc`.

[4] Wind River Simics. `www.windriver.com/producs/simics`.

[5] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *FAST*, 2004.

[6] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM Communications*, 13, July 1970.

[7] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, 2007.

[8] Y. Chen, A. Kumar, and J. Xu. A new design of Bloom filter for packet inspection speedup. In *GLOBECOM*, 2007.

[9] J. Collins and D. M. Tullsen. Hardware identification of cache conflict misses. In *MICRO*, 1999.

[10] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *JPDC*, 2004.

[11] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 2008.

[12] A. Goel and P. Gupta. Small subset queries and Bloom filters using ternary associative memories, with applications. In *SIGMETRICS*, 2010.

[13] E. G. Hallnor and S. K. Reinhardt. A fully associative software managed cache design. In *ISCA*, 2000.

[14] Intel. Intel 64 and IA-32 architectures software developer's manuals, 2011.

[15] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *ICS*, 2004.

[16] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT*, 2008.

[17] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction. In *ISCA*, 2010.

[18] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002.

[19] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu. Run-time cache bypassing. *IEEE TC*, 1999.

[20] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.

[21] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[22] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 2010.

[23] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, 2007.

[24] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[25] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA*, 2010.

[26] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO*, 2010.

[27] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO*, 2008.

[28] K. Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in SMT processors. In *ISPASS*, 2001.

[29] M. J. Lyons and D. Brooks. The design of a Bloom filter hardware accelerator for ultra low power systems. In *ISLPED*, 2009.

[30] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, 2003.

[31] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA*, 2007.

[32] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, 1993.

[33] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *ICS*, 2002.

[34] T. Piquet, O. Rochecouste, and A. Seznec. Exploiting single-usage for effective memory management. In *ACSAC*, 2007.

[35] M. K. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[36] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA*, 2006.

[37] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[38] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way cache: Demand based associativity via global replacement. In *ISCA*, 2005.

[39] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO*, 2007.

[40] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*, 2005.

[41] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE TC*, 1997.

[42] G. Rivera and C.-W. Tseng. Compiler optimizations for eliminating cache conflict misses. Technical Report UMIACS-TR-97-59, University of Maryland, College Park, 1997.
[43] J. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *ICPP*, 1996.
[44] D. Rolan, B. B. Fraguela, and R. Doallo. Adaptive line placement with the set balancing cache. In *MICRO*, 2009.
[45] D. Rolan, B. B. Fraguela, and R. Doallo. Reducing capacity and conflict misses using set saturation levels. In *HiPC*, 2010.
[46] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *MICRO*, 2010.
[47] A. Seznec. A case for two-way skewed-associative caches. In *ISCA*, 1993.
[48] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.
[49] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.
[50] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE TC*, Sep. 1992.
[51] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, 2002.
[52] G. S. Tyson, M. K. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO*, 1995.
[53] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *PACT*, 2002.
[54] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. Steely Jr., and J. Emer. SHIP: Signature-based hit predictor for high performance caching. In *MICRO*, 2011.
[55] Y. Xie and G. H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA*, 2009.
[56] D. Zhan, H. Jiang, and S. C. Seth. STEM: Spatiotemporal management of capacity for intra-core last level caches. In *MICRO*, 2010.

A List of 2-core Workloads

Low-int Low-sens
fma3d-cactusADM
fma3d-lucas
lucas-mgrid
cactusADM-mgrid
h264ref-cactusADM
h264ref-wupwise

Low-int Medium-sens
cactusADM-applu
ammp-wupwise
lucas-applu
mgrid-h264ref
facerec-lucas
bzip2-lucas
bzip2-cactusADM
h264ref-mgrid
cactusADM-ammp
mgrid-fma3d
h264ref-fma3d
parser-cactusADM
bzip2-wupwise
vpr-lucas
ammp-cactusADM
twolf-lucas
wupwise-vpr
wupwise-ammp
wupwise-facerec

Low-int High-sens
facerec-vpr
applu-twolf
bzip2-vpr
facerec-twolf
parser-vpr
twolf-vpr
applu-facerec
facerec-parser
ammp-twolf
ammp-bzip2
ammp-parser
applu-vpr
bzip2-facerec
ammp-facerec
ammp-vpr
applu-bzip2
bzip2-parser
applu-parser
ammp-applu
parser-twolf
bzip2-fma3d
apsi-vpr
mgrid-vpr
h264ref-parser
ammp-mgrid
h264ref-vpr
ammp-h264ref
bzip2-mgrid
fma3d-parser
apsi-facerec
apsi-bzip2
fma3d-vpr
bzip2-h264ref
facerec-h264ref
facerec-fma3d
mgrid-parser
fma3d-twolf
mgrid-twolf
apsi-parser
applu-mgrid

Medium-int Low-sens
cactusADM-galgel
lucas-galgel
wupwise-equake
cactusADM-equake
zeusmp-mgrid
equake-cactusADM
zeusmp-h264ref
galgel-wupwise
wupwise-galgel
galgel-lucas
equake-lucas

Medium-int Medium-sens
equake-mgrid
lucas-omnetpp
lucas-xalancbmk
vpr-zeusmp
omnetpp-cactusADM
galgel-fma3d
fma3d-equake
facerec-zeusmp
bzip2-zeusmp
xalancbmk-cactusADM
zeusmp-bzip2
fma3d-galgel
xalancbmk-wupwise
parser-zeusmp
ammp-zeusmp
h264ref-galgel
twolf-zeusmp
cactusADM-omnetpp
equake-h264ref
cactusADM-xalancbmk
omnetpp-lucas
galgel-vpr
h264ref-omnetpp
bzip2-equake
applu-galgel
fma3d-xalancbmk
equake-twolf
ammp-equake
fma3d-omnetpp
mgrid-xalancbmk
equake-parser

Medium-int High-sens
bzip2-xalancbmk
facerec-xalancbmk
ammp-omnetpp
twolf-xalancbmk
applu-omnetpp
omnetpp-parser
applu-xalancbmk
ammp-xalancbmk
facerec-omnetpp
omnetpp-vpr

High-int Low-sens
apsi-lbm
astar-lucas
h264ref-lbm
apsi-bwaves
sphinx3-cactusADM
lbm-mgrid
zeusmp-galgel
astar-cactusADM
fma3d-bwaves
soplex-cactusADM
wupwise-soplex
cactusADM-sphinx3
milc-h264ref
galgel-zeusmp
libquantum-h264ref
lucas-sphinx3
bwaves-h264ref
fma3d-libquantum
fma3d-milc
cactusADM-leslie3d
wupwise-astar

High-int Medium-sens
art-cactusADM
fma3d-soplex
libquantum-facerec
wupwise-mcf
wupwise-swim
soplex-mgrid
mgrid-astar
bwaves-vpr
ammp-bwaves
bzip2-lbm
soplex-fma3d
twolf-lbm
gobmk-leslie3d
leslie3d-apsi
art-lucas
lucas-swim
bzip2-bwaves
parser-milc
cactusADM-art
omnetpp-zeusmp
art-wupwise
facerec-lbm
equake-galgel
twolf-libquantum
soplex-twolf
facerec-soplex
art-h264ref
ammp-astar
applu-sphinx3
equake-xalancbmk
astar-facerec
astar-twolf
parser-soplex
astar-vpr
galgel-omnetpp
parser-sphinx3
h264ref-mcf
equake-omnetpp
sphinx3-vpr
astar-bzip2
bzip2-sphinx3
h264ref-swim

High-int High-sens
applu-swim
mcf-vpr
omnetpp-xalancbmk
art-twolf
bzip2-swim
art-parser
facerec-mcf
art-bzip2
parser-swim
art-vpr
ammp-swim
ammp-mcf
swim-twolf
mcf-parser
art-facerec
ammp-art
bzip2-mcf
swim-vpr
mcf-twolf
facerec-swim

Server Workloads
apache20-tpch2
apache20-tpch6
apache20-tpch17
apache20-tpcc64
tpch2-tpch6
tpch2-tpch17
tpch2-tpcc64
tpch6-tpch17
tpch6-tpcc64
tpch17-tpcc64

B List of 4-core Workloads

Low-int Low-sens
ammp-bwaves-cactusADM-wupwise
bwaves-bzip2-cactusADM-lucas
ammp-bwaves-cactusADM-lucas
cactusADM-fma3d-mgrid-milc
fma3d-lbm-lucas-mgrid
astar-cactusADM-h264ref-wupwise
cactusADM-fma3d-soplex-wupwise
apsi-cactusADM-fma3d-milc
bzip2-libquantum-lucas-wupwise

Low-int Medium-sens
equake-mgrid-omnetpp-parser
equake-facerec-fma3d-xalancbmk
bzip2-fma3d-leslie3d-vpr
bzip2-fma3d-h264ref-swim
art-bzip2-fma3d-h264ref
apsi-bzip2-sphinx3-twolf
facerec-fma3d-h264ref-swim
applu-equake-facerec-galgel
ammp-omnetpp-wupwise-xalancbmk
applu-apsi-art-fma3d
applu-omnetpp-parser-zeusmp
ammp-apsi-h264ref-swim
apsi-parser-sphinx3-vpr
astar-mgrid-parser-vpr
astar-bzip2-h264ref-twolf
applu-art-cactusADM-vpr
bzip2-leslie3d-mgrid-parser
bzip2-lucas-mcf-vpr
astar-h264ref-twolf-vpr
ammp-omnetpp-wupwise-xalancbmk
art-h264ref-lucas-mgrid
cactusADM-equake-galgel-parser
ammp-cactusADM-fma3d-soplex
cactusADM-facerec-omnetpp-zeusmp
bzip2-lucas-milc-vpr
art-lucas-twolf-wupwise
apsi-facerec-h264ref-libquantum
cactusADM-facerec-h264ref-soplex
cactusADM-leslie3d-mgrid-parser
fma3d-lucas-parser-soplex

Low-int High-sens
bzip2-facerec-parser-swim
ammp-art-facerec-twolf
ammp-facerec-swim-twolf
art-facerec-twolf-vpr
applu-facerec-mcf-twolf
bzip2-facerec-omnetpp-xalancbmk
art-bzip2-facerec-parser
ammp-bzip2-mcf-twolf
applu-omnetpp-twolf-xalancbmk
omnetpp-parser-twolf-xalancbmk
ammp-omnetpp-parser-xalancbmk
ammp-facerec-mcf-vpr
applu-facerec-mcf-twolf
facerec-omnetpp-twolf-xalancbmk
ammp-applu-mcf-parser

Medium-int Low-sens
ammp-facerec-parser-swim
applu-facerec-mcf-vpr
facerec-swim-twolf-vpr
applu-omnetpp-vpr-xalancbmk
ammp-applu-bzip2-mcf
bwaves-libquantum-lucas-vpr
apsi-bwaves-cactusADM-leslie3d
libquantum-lucas-omnetpp-zeusmp
bwaves-cactusADM-fma3d-soplex
apsi-bwaves-cactusADM-sphinx3
ammp-cactusADM-lbm-libquantum
facerec-libquantum-milc-wupwise
astar-lbm-mgrid-wupwise
apsi-fma3d-lbm-libquantum

Medium-int Medium-sens
mgrid-sphinx3-swim-wupwise
fma3d-lbm-soplex-twolf
art-bwaves-fma3d-mgrid
art-bwaves-fma3d-h264ref
h264ref-lucas-soplex-swim
apsi-bwaves-soplex-twolf
bwaves-bzip2-lucas-mcf
bzip2-lbm-swim-wupwise
astar-bwaves-h264ref-vpr
bwaves-equake-galgel-vpr
ammp-libquantum-mcf-parser
apsi-leslie3d-mcf-twolf
art-h264ref-soplex-twolf
astar-bzip2-facerec-soplex
apsi-galgel-omnetpp-swim
art-bzip2-mgrid-sphinx3
mgrid-sphinx3-swim-vpr
fma3d-omnetpp-soplex-xalancbmk
ammp-art-lucas-mcf
facerec-lbm-mcf-twolf

Medium-int High-sens
art-facerec-parser-swim
art-facerec-swim-twolf
ammp-art-mcf-parser
mcf-parser-swim-vpr
art-mcf-parser-twolf
ammp-art-swim-vpr
ammp-art-mcf-vpr
bzip2-omnetpp-swim-xalancbmk
art-mcf-parser-vpr
mcf-swim-twolf-vpr

High-int Low-sens
bwaves-lbm-xalancbmk-zeusmp
astar-lbm-lucas-soplex
astar-bwaves-cactusADM-sphinx3
bwaves-fma3d-lbm-sphinx3
art-bwaves-lbm-lucas
astar-bwaves-galgel-zeusmp
apsi-astar-lbm-libquantum
cactusADM-lbm-milc-swim
astar-bwaves-mgrid-milc

High-int Med-sens
art-lbm-milc-twolf
apsi-art-libquantum-soplex
astar-lbm-parser-soplex
bwaves-facerec-leslie3d-soplex
fma3d-lbm-leslie3d-mcf
art-libquantum-xalancbmk-zeusmp
bwaves-galgel-leslie3d-xalancbmk
astar-galgel-mcf-zeusmp
ammp-lbm-milc-swim
equake-milc-soplex-xalancbmk

High-int High-sens
leslie3d-mcf-soplex-twolf
applu-art-lbm-swim
art-omnetpp-swim-zeusmp
art-bzip2-leslie3d-sphinx3
leslie3d-soplex-swim-vpr
mcf-milc-swim-vpr
astar-galgel-mcf-omnetpp
bzip2-leslie3d-soplex-swim
art-galgel-leslie3d-omnetpp
facerec-soplex-sphinx3-swim

Server Workloads
apache20-tpcc64-tpch17-tpch2
apache20-tpcc64-tpch2-tpch6
apache20-tpch17-tpch2-tpch6
apache20-tpcc64-tpch17-tpch6
tpcc64-tpch17-tpch2-tpch6