Improving Cache Performance using Victim Tag Stores

SAFARI Technical Report No. 2011-009

Vivek Seshadri†, Onur Mutlu†, Todd Mowry†, Michael A Kozuch‡
{vseshadr,tcm}@cs.cmu.edu, onur@cmu.edu, michael.a.kozuch@intel.com
†Carnegie Mellon University, ‡Intel Corporation

September 26, 2011

Abstract

Main memory latency has been a major bottleneck for system performance. With increasing memory bandwidth demand due to multiple on-chip cores, effective cache utilization is important. In a system with limited cache space, we would ideally like to prevent an incoming block with low reuse from evicting another block with high reuse. This requires predicting the reuse behavior of the incoming block before inserting into the cache.

In this paper, we propose a new, simple mechanism that predicts the reuse behavior of a missed block and decides where to insert the block in the cache based on the prediction. A block predicted to have high reuse is inserted at the most recently used position while a majority of the remaining blocks are inserted at the least recently used position. The key observation behind our prediction mechanism is that if a block with high reuse gets prematurely evicted from the cache, it will be accessed soon after eviction. To implement this mechanism, we augment the cache with a Bloom filter that keep tracks of addresses of recently evicted blocks. Missed blocks whose addresses are present in the Bloom filter are predicted to have high reuse.

We compare our mechanism to five state-of-the-art cache management mechanisms that use different insertion and replacement policies and show that it provides significant performance improvements.

1 Introduction

Off-chip memory latency has always been a major bottleneck for system performance. With increasing number of on-chip cores, the problem is bound to get only worse as the demand on the memory system will increase with more concurrently running applications. As a result, on-chip last-level cache (LLC) utilization becomes increasingly important. In systems with limited cache space, preventing blocks with low reuse – i.e., blocks that do not receive further hits after insertion – from polluting the cache becomes critical to system performance.

Ideally, an incoming block with low reuse should be prevented from evicting a more useful block from the cache. This requires the cache to predict the reuse behavior of a missed cache block and based on the outcome of the prediction, decide where to insert the block in the cache. We would like to insert a block with high reuse with a policy that retains it in the cache for a long period – e.g. at the most recently used (MRU) position for the commonly used least recently used replacement (LRU) policy. On the other hand, we would like to insert a block with low reuse with a policy that prevents it from evicting other useful blocks – e.g. at the least recently used position. A major question is, how to predict the reuse behavior of a missed cache block?

Prior work [6, 13, 24, 29, 38] has proposed solutions to answer this question. The proposed mechanisms can be broadly classified into two categories: 1) *program-counter* based approaches [24, 38, 39]:

the program counter of the instruction that created the last-level cache miss is used to predict the reuse behavior of blocks, 2) *memory-region* based approaches [13, 39]: The address space is divided into multiple regions and the reuse prediction for a missed block is based on which region of memory it belongs to. Both these approaches use a table to learn the predictions for either different program counters or different memory regions.

A major shortcoming with these two approaches is that they do not adapt their prediction based on the behavior of individual cache blocks. Instead, they group a set of blocks based on the program counter that loaded those blocks or the memory region to which they belong and make a single prediction for all the blocks within a group. However, blocks loaded by the same program counter or those belonging to the same memory region can have very different reuse behavior. Some of the prior studies [6, 29] propose techniques to adapt the cache insertion policy on a per-block basis, but they target direct-mapped L1 caches. As a result, they do not address the issues related to highly-associative last-level caches present in modern processors.

Our **goal** in this work is to design a mechanism that adapts the reuse prediction on a per-block basis, with low hardware cost and implementation complexity. To this end, we propose a mechanism that predicts the reuse behavior of a block based on how recently it was evicted from the cache after its last access. The key observation behind our prediction mechanism is that if a block with high reuse gets prematurely evicted from the cache, then it will be accessed again *soon* after eviction. On the other hand, a block with low reuse will likely not be accessed for a long time after getting evicted from the cache.

For a cache following the conventional LRU policy, our mechanism inserts a block predicted to have high reuse at the MRU position. This ensures that the block does not get evicted from the cache even before it can receive a hit. On the other hand, blocks predicted to have low reuse are inserted at the LRU position with a high probability and at the MRU position with low probability [25]. By doing so, the cache prevents most of the blocks with low reuse from evicting more useful blocks from the cache. As we explain in Section 2, inserting a small fraction of low-reuse blocks at the MRU position improves the performance of applications with large working sets.

We show that the our proposed prediction scheme can be implemented in existing caches by keeping track of a addresses of blocks that are recently evicted from the cache. For this purpose, we augment a conventional cache with a structure called victim tag store. We show that the victim tag store can be compactly implemented using a Bloom filter [3] and a counter. Our mechanism requires no modifications to the cache. In addition, it operates only on a cache miss and hence, the cache hit operation remains unchanged.

We compare our proposed mechanism to two different groups of prior art. First, we compare to two mechanisms that choose the cache insertion policy on a per-application basis: adaptive insertion policy [10, 25], and dynamic re-reference interval prediction policy [11]. Second, we compare to three mechanisms that predict the reuse behavior of the missed block and then decide its insertion policy: single-usage block prediction [24], run-time cache bypassing [13], and adaptive replacement cache [21].

We make the following major **contributions**:

- We propose a new mechanism that predicts the reuse behavior of a missed block before inserting it into the cache. Our mechanism adapts its prediction to the behavior of the missed block rather than using program counters or memory region related information. We provide a low-overhead implementation of our mechanism using Bloom filters.
- We compare our approach with two state-of-the-art cache management schemes that are agnostic to reuse behavior of blocks at the time of insertion. Evaluations show that, compared to the best

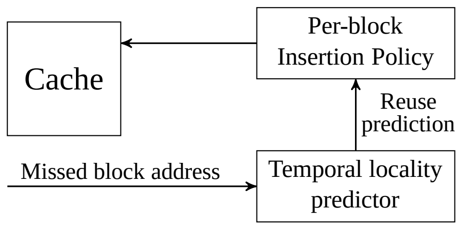
previous mechanism, it improves system throughput by 6% for 2-core systems (15% compared to LRU) and 8% for 4-core systems (21% compared to LRU). We also compare our mechanism to three prior proposals for block-level cache insertion policies and show that it provides better performance 8% compared to the best previous mechanism proposed for on-chip caches).

2 Our Approach: VTS-cache

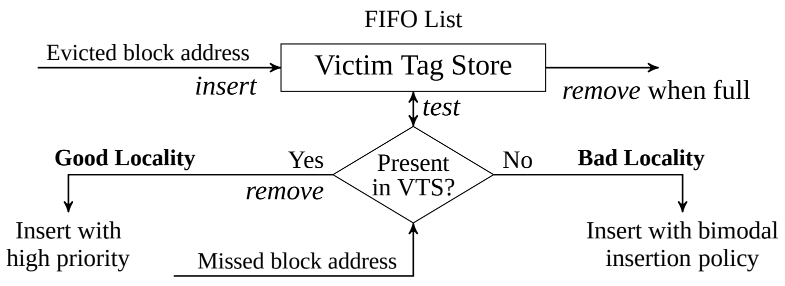
As we mentioned in the introduction, majority of the proposals for high-performance cache management have looked at techniques to improve cache replacement. As we will show in the paper, significant performance improvements can be achieved by using a cache insertion policy that considers the temporal locality of a missed block before inserting it into the cache. Starting first with what the ideal insertion policy would do, we describe our approach to address this problem.

The ideal insertion policy which attempts to maximize the overall hit rate of the system will work as follows. On a cache miss, it will determine the degree of temporal reuse of the missed block *before* inserting it into the cache. If the block has little or no temporal reuse compared to the blocks present in the cache, then it will insert the block with the *lowest* priority or bypass the cache if possible. This will force the replacement policy to evict the block from the cache immediately without affecting other useful blocks in the cache. On the other hand, if the block does have good temporal locality, then the ideal mechanism will insert the block with a higher priority that keeps the block in the cache long enough for it to receive a hit. Therefore, the ideal mechanism will follow *different* insertion policies for different blocks based on their temporal locality behavior.

Our goal in this work is to approximate the performance of this ideal mechanism with low hardware overhead. To achieve this, we propose a new cache management technique, VTS-cache, that has components as shown in Figure 1a. The first component is a *temporal locality predictor* that takes the address of a missed cache block and predicts whether the block has good temporal locality. The second component is a *per-block insertion policy* that decides the insertion policy for a missed block based on its temporal locality prediction.



a) Components



b) VTS Cache Operation

Figure 1: Conceptual design of the VTS-cache

2.1 VTS-cache: Conceptual Design

VTS-cache predicts the reuse behavior of a missed block based on how recently it was evicted from the cache the last time it was accessed. The key insight behind this scheme is as follows: if a block with good temporal locality gets prematurely evicted from the cache, then it will likely be accessed soon after eviction. On the other hand, a block with low temporal locality will not be accessed again for a long time after getting evicted.

Figure 1b shows the schematic representation of the operation of the VTS-cache, based on this insight. The cache is augmented with a structure that keeps track of *addresses* of blocks that were recently evicted from the cache. We call this structure the *victim tag store* (VTS). Conceptually, VTS is a first-in-first-out (FIFO) list. When a cache block gets evicted from the cache, its address is inserted at the head of the VTS. Once the VTS gets full, addresses are removed from the tail of the VTS to accommodate more insertions. On a cache miss, VTS-cache tests if the missed block address is present in the VTS. If it is, VTS-cache assumes that the block was prematurely evicted from the cache and predicts that it has good temporal locality. If the block address is not present in the VTS, then VTS-cache predicts the block to have bad temporal locality.

Intuitively, the insertion policy should ensure two things: 1) blocks predicted to have good temporal locality (i.e., predicted-good-locality blocks) should not get evicted from the cache before they are accessed again, 2) blocks predicted to have bad temporal locality (i.e., predicted-bad-locality blocks) should get evicted from the cache immediately, without affecting other useful blocks. Therefore, one insertion policy could be to insert a predicted-good-locality block with high priority and a predicted-bad-locality block with the *lowest* priority[1].

However, for an application with a working set that does not fit in the cache, always inserting a predicted-bad-locality block with the lowest priority does not provide good performance. Most blocks of such an application will be predicted-bad-locality because they have large reuse distances and, hence, are unlikely to be accessed immediately after getting evicted from the cache. For such an application, we would like to keep a fraction of the working set in the cache, to ensure cache hits at least for that fraction. As previous work has shown [25], the bimodal insertion policy (BIP) achieves this by determining the insertion priority of a block probabilistically: high priority with very low probability ($\frac{1}{64}$ in our design), lowest priority with very high probability. Therefore, to ensure good performance for a wide variety of applications, including those with large working sets, VTS-cache inserts a predicted-bad-locality block using the bimodal insertion policy.

In summary, VTS-cache keeps track of addresses of blocks that are recently evicted from the cache in the victim tag store. On a cache miss, if the missed block address is present in the VTS, VTS-cache predicts the block to have good temporal locality and inserts the block with high priority. Otherwise, VTS-cache predicts the block to have bad temporal locality and inserts the block using the bimodal insertion policy (i.e., high priority with a low probability, lowest priority with a high probability).

3 Qualitative Comparison to Previous Approaches

Prior research [25, 10, 11] has proposed mechanisms to vary the insertion policy based on the application behavior. The primary aim of these mechanisms is to ensure good performance for applications with large working sets and applications with multiple access patterns. In this section, we emphasize the need for a per-block insertion policy by qualitatively comparing VTS-cache to two prior proposals: thread-aware dynamic insertion policy [10] and thread-aware dynamic re-reference interval prediction policy [11]. Before proceeding with our case study, we provide an overview of these two mechanisms.

[1]Bypassing bad-locality blocks is an option. We find that using bypassing improves the performance of VTS-Cache by 1% but do not include these results due to space constraints.

3.1 Overview of Prior Approaches

Thread-aware dynamic insertion policy [10] (TA-DIP): The conventional LRU policy inserts all new blocks at the most-recently-used position and evicts blocks from the least-recently-used position. Certain applications benefit from this policy. On the other hand, this policy provides sub-optimal performance for applications with working sets bigger than the cache size, as blocks of such an application will keep evicting each other from the cache. These applications benefit from a bimodal insertion policy [25] (BIP) which inserts majority of the blocks at the least-recently-used position, thereby retaining a fraction of the working set in the cache. The key idea behind TA-DIP is to determine the best insertion policy for each application dynamically in the presence of other applications. For this purpose, TA-DIP uses set dueling [25] for each application to identify which policy provides fewer misses (using a small sample of sets) and uses that policy for all the blocks of that application.
Thread-aware dynamic re-reference interval prediction [11] (TA-DRRIP): The LRU policy inserts all new blocks at the MRU position. Therefore, blocks that do not have any temporal locality stay in the cache for a long time before getting evicted, occupying precious cache space. To mitigate this problem, Jaleel et al. propose the re-reference interval prediction (RRIP) policy which prioritizes the cached blocks based on a prediction of how far into the future they will be accessed (the farther the re-reference, the lower the priority). All blocks are inserted at the next-to-lowest priority, preventing blocks with poor temporal locality from occupying cache space for a long time. A block is elevated to the highest priority on a hit. On a request for replacement, the priorities of *all* the blocks are decremented until some block reaches the lowest possible priority. To benefit applications with large working sets, the paper also uses a bimodal insertion policy, bimodal RRIP (similar to BIP [25]), which inserts most of the blocks at the lowest priority. Similar to TA-DIP, TA-DRRIP dynamically determines the best policy between the static RRIP policy and the bimodal RRIP policy for each application in the system, using set dueling [25].

3.2 Case Study

Prior work (e.g., [11, 21, 25]) has identified three common memory access patterns exhibited by various applications. The first access pattern consists of repeated references to a small set of blocks, that fits into the cache, referred to as the *cache fitting* access pattern. The second access pattern consists of repeated references to a large sequence of blocks, that does not fit in the cache, referred to as the *cyclic reference pattern*. The last access pattern, called *scan*, consists of references to a large set of blocks with no temporal locality. With respect to a fully-associative cache with 4 blocks, the sequence $(f_1, f_2, f_1, f_2, f_2, f_1)$ would be a cache fitting pattern, $(c_1, c_2, ..., c_6, c_1, c_2, ..., c_6)$ would be a cyclic reference pattern, and the sequence $(s_1, s_2, ..., s_{10})$ would be a scan.

We illustrate the performance benefits of using a per-block insertion policy (compared to the mechanisms described before) using a case study with two applications, A and B, whose references contain these access patterns. Figure 2 shows the sequence of blocks referenced by applications A and B in one iteration of their respective loops. As indicated in the figure, application A's references consist of a *cache fitting* access pattern, to a sequence of blocks F, followed by a *scan* to a sequence of blocks S (different for each iteration of the loop). On the other hand, application B's references consists of a cyclic reference pattern to a large sequence of blocks C. The figure also shows the interleaved access sequence when the two applications are running concurrently in a system with a shared, fully-associative cache. For ease of understanding, we assume that the loops of the two applications are synchronized and hence, the interleaved access sequence also repeats in a loop.

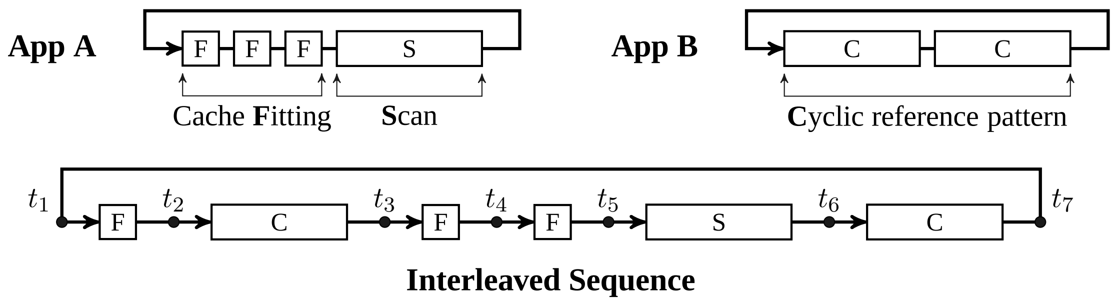


Figure 2: Blocks referenced by applications A and B in one iteration of their respective loops. Blocks in the scan S are different for each iteration of the loop. But F and C denote the same sequence of blocks for all iterations. The interleaved sequence shows the access sequence as seen by the shared cache when the applications are running concurrently. The interleaved sequence is also assumed to repeat.

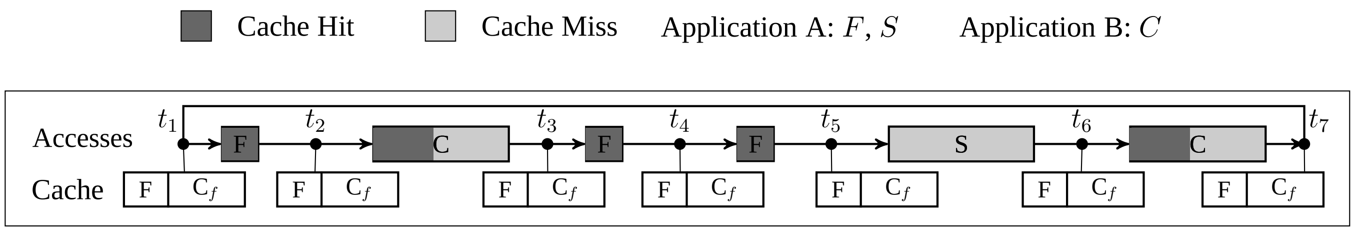
Figures 3a through 3e show the steady state cache performance of various mechanisms on this interleaved access sequence. The ideal policy to maximize hit rate (figure 3a) for this access sequence is to cache A's working set, i.e. F, and a subset of blocks from B's working set, i.e. C. This is because, in every iteration, the blocks of F are accessed the most number of times (3 times) and hence, they should be cached completely. Although all the blocks of C are accessed twice in every iteration, only a portion of them can be cached in the remaining available cache space. All the other blocks, especially those belonging to the scan S, should be inserted with the lowest priority and evicted from the cache immediately.

The commonly used application-unaware LRU policy inserts all the blocks at the most recently used (MRU) position. This has two negative effects. One, blocks of the scan S (accessed between t_5 and t_6, figure 3b), which have no reuse, evict useful blocks of both the applications. Two, blocks of C, due to their large reuse interval, evict each other and also blocks of F from the cache (between t_2 and t_3). As a result, the LRU policy results in cache hits only for one set of accesses to blocks of F in every iteration.

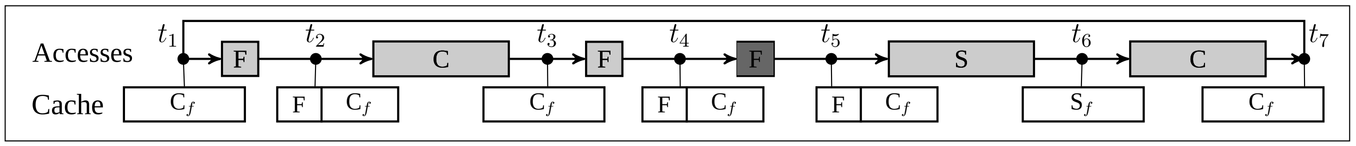
Application A, due to its cache fitting access pattern, incurs more misses with the bimodal insertion policy. This is because blocks of F might repeatedly get inserted at the LRU position and evicted immediately, before they can be accessed again. Therefore, TA-DIP will follow the conventional LRU policy for A. For the same reason, TA-DRRIP will follow the static RRIP policy for A. On the other hand, application B, because of its large working set, will benefit from the bimodal insertion policy as the alternative policies (LRU or static RRIP) will cause B's blocks to evict each other. Therefore, both TA-DIP and TA-DRRIP will follow the bimodal insertion policy for application B.

By following the bimodal insertion policy for application B, both TA-DIP and TA-DRRIP improve the hit rate for application B compared to the LRU policy. However, since they follow a single insertion policy for application A, blocks of S are treated similarly to blocks of F. In the case of TA-DIP, this causes accesses to S to evict the complete working set from the cache as all its blocks are inserted at the MRU position (figure 3c). TA-DRRIP, on the other hand, mitigates this effect by following a more robust policy. However, for every block of S evicted from the cache, the priorities of all the other blocks are reduced. Therefore, blocks of S will start polluting the cache, evicting useful blocks. As a result, even though TA-DRRIP improves performance, it is still far from achieving the ideal hit rate (figure 3d).

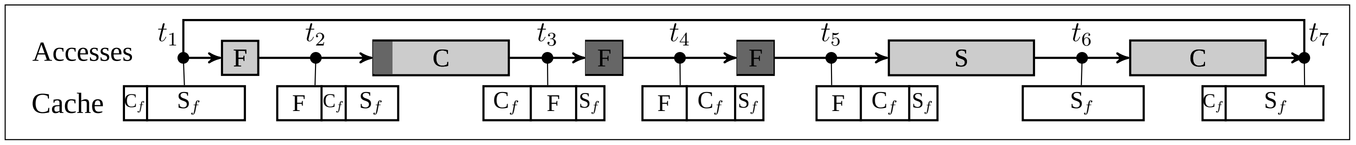
Finally, figure 3e shows the cache performance of our proposed mechanism, VTS-cache. Unlike prior approaches, VTS-cache chooses the insertion policy based on the block behavior. In this example, there are two situations where VTS-cache makes a better decision compared to prior approaches. First, it chooses the bimodal insertion policy for blocks of S as they will be accessed for the first time and hence, will not be present in the VTS. By doing so, it prevents these blocks from polluting the cache by inserting a


Cache Hit
Cache Miss
Application A: F, S
Application B: C
Accesses
Cache
t_1 F t_2 C t_3 F t_4 F t_5 S t_6 C t_7
F C$_f$ | F C$_f$ | F C$_f$ | F C$_f$ | F C$_f$ | F C$_f$ | F C$_f$

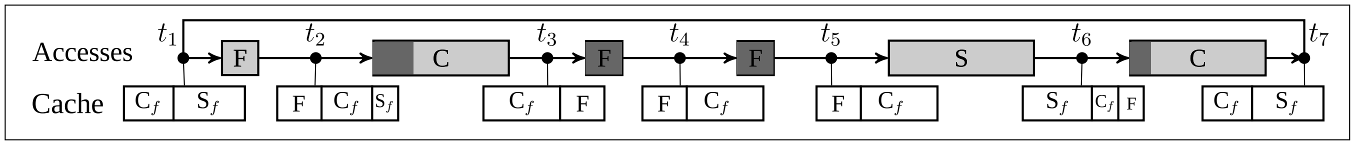
(a) **IDEAL** policy: Always keeps F and a fraction of C in the cache. All other blocks are filtered at the time of insertion.


Accesses
Cache
t_1 F t_2 C t_3 F t_4 F t_5 S t_6 C t_7
C$_f$ | F C$_f$ | C$_f$ | F C$_f$ | F C$_f$ | S$_f$ | C$_f$

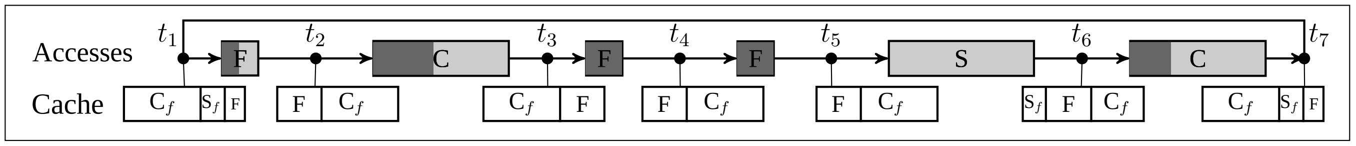
(b) Application-unaware **LRU** policy: All blocks of both A and B are inserted at the MRU position (left end of the cache).


Accesses
Cache
t_1 F t_2 C t_3 F t_4 F t_5 S t_6 C t_7
C$_f$ S$_f$ | F C$_f$ S$_f$ | C$_f$ F S$_f$ | F C$_f$ S$_f$ | F C$_f$ S$_f$ | S$_f$ | C$_f$ S$_f$

(c) **Thread-aware DIP**: TA-DIP chooses the conventional LRU policy for A, i.e. all of A's blocks (F & S) are inserted at the MRU position, and the bimodal insertion policy for B, i.e. only a small fraction of B's blocks (C) are inserted at the MRU position and the rest at LRU position. Without loss of generality, we assume this fraction to be $\frac{1}{8}$ in this example.


Accesses
Cache
t_1 F t_2 C t_3 F t_4 F t_5 S t_6 C t_7
C$_f$ S$_f$ | F C$_f$ S$_f$ | C$_f$ F | F C$_f$ | F C$_f$ | S$_f$ C$_f$ F | C$_f$ S$_f$

(d) **Thread-aware DRRIP**: TA-DRRIP chooses the static RRIP policy for A, i.e. all of A's blocks (F & S) are inserted with the next-to-lowest priority. For B, it chooses the bimodal RRIP policy, i.e. only a fraction ($\frac{1}{8}$) of its blocks (C) are inserted with the next-to-lowest priority. The remaining blocks of B are inserted with the lowest priority.


Accesses
Cache
t_1 F t_2 C t_3 F t_4 F t_5 S t_6 C t_7
C$_f$ S$_f$ F | F C$_f$ | C$_f$ F | F C$_f$ | F C$_f$ | S$_f$ F C$_f$ | C$_f$ S$_f$ F

(e) **VTS-cache**: VTS-cache chooses the insertion policy on a per-block basis. Blocks of S are always inserted with the bimodal policy (between t_5 and t_6) as it will be their first access. Blocks of C are also inserted with the bimodal policy due to their large reuse distance. However, between t_1 and t_2, blocks of F that were evicted towards the end of the previous iteration (between t_6 and t_7), will be inserted at the MRU position as they were recently evicted from the cache.

Figure 3: Benefits of using VTS-cache: Each sub-figure shows the steady state cache performance of the corresponding cache management mechanism on the interleaved sequence (the first row of boxes). The dark gray portion indicates the fraction of blocks that hit in the cache. The cache state is indicated in the second row of boxes at the corresponding time step. The labels represent the set of blocks that are cached at that time. The subscript f indicates that only a fraction of those blocks are present in the cache.

majority of them with the *lowest* priority. Second, at the end of each iteration some blocks of F get evicted from the cache and are accessed *immediately* at the beginning of the next iteration, while they are in the VTS. Therefore, VTS-cache predicts these blocks to have good temporal locality and inserts *all* of them with a high priority. Blocks of C, due to their large reuse interval, are unlikely to be present in the VTS. Therefore, VTS-cache will insert them with the bimodal insertion policy. As a result, it retains a portion of C in the cache.

Therefore, by following a per-block insertion policy based on different access behavior of blocks even within an application (in the example, for F and S of application A), VTS-cache is able to prevent blocks with little or no temporal locality from polluting the cache. As a result, it is able to perform better than other approaches which are agnostic to the temporal locality of missed cache blocks.

4 VTS-cache: Design Choices & Improvements

In section 2.1, we described the conceptual design and operation of the VTS-cache. In this section, we discuss some design choices and improvements to VTS-cache. Specifically, we discuss the impact of the size of the VTS, effect of VTS-cache on LRU-friendly applications, and the possibility of incorporating thread-awareness into the VTS design.

4.1 Size of the Victim Tag Store

The size of the VTS, i.e., the number of block addresses it can keep track of, determines the boundary between blocks that are classified as recently evicted and those that are not. Intuitively, the VTS size should be neither too small nor too large. Having too small a VTS will lead to mispredictions for a lot of blocks with good temporal locality. As a result, many such blocks will get inserted at the lowest priority and evicted from the cache, thereby increasing the miss rate. Conversely, having too large a VTS will lead to a good-temporal-locality prediction even for a block with a large reuse distance. Hence, these blocks will get inserted with high priority and pollute the cache by evicting more useful blocks.

In our evaluations, we find that VTS-cache provides the best performance when the size of the VTS is same as the number of blocks in the cache. The reason behind this could be that, a VTS with size smaller than the number of blocks in the cache will lead to poor performance for an application whose working set just fits the cache as a majority of its useful blocks will inserted with the bimodal insertion policy. On the other hand, a VTS with size bigger than the cache size will lead to poor performance for an application whose working set is just larger than the cache size as most of its blocks will be inserted with high priority causing them to evict each other from the cache. In all our evaluations (except the one that studies the effect of the VTS size), we set the VTS size to be same as the number of blocks in the cache.

4.2 Improving Robustness of VTS-cache

When a block with good temporal locality is accessed for the first time, VTS-cache will falsely predict that it has poor temporal locality. Hence, it likely inserts the block with the *lowest* priority, forcing the replacement policy to evict the block immediately on a set conflict. Therefore, for an application which is LRU-friendly, i.e., blocks that are just accessed have good temporal locality, VTS-cache incurs one additional miss for a majority of blocks, by not inserting them with high priority on their first access. In most of the workloads with LRU-friendly applications, we find that this misprediction does not impact performance, as the cache is already filled with useful blocks. However, when *all* applications in a workload are LRU-friendly, we find that VTS-cache performs worse than prior approaches.

To increase the robustness of VTS-cache, we propose a dynamic scheme which uses set dueling [25] to determine if *all* the applications in the system will benefit from a always-high-priority insertion policy. If so, then the cache ignores the VTS and inserts all blocks of all applications with high priority. We call this enhancement *D-VTS* and evaluate it in Section 8. Our results indicate that using this enhancement mitigates the performance loss incurred by VTS-cache for workloads with all LRU-friendly applications and does not affect performance for other workloads.

4.3 Incorporating Thread-awareness into VTS

When a shared cache is augmented with the victim tag store, the VTS is also shared by concurrently running applications. Therefore, it is possible that applications interfere with each other in the VTS too, i.e., evicted blocks of one application can remove addresses of blocks of another application from the VTS. This can cause suboptimal or incorrect temporal locality predictions for the applications sharing the VTS: due to interference in the VTS, a block with good reuse behavior can actually get evicted early and thus be predicted as having bad temporal locality. One way to solve this problem is to partition the VTS equally among multiple hardware threads sharing it. We call such a partitioned VTS as a *thread-aware VTS*. We found in our evaluations that a thread-aware VTS only provides minor performance improvements (around 1% on an average) compared to a thread-unaware VTS design. Although this could be an artifact of the applications and the system configuration used in our evaluations, we do not extensively evaluate this design choice due to space limitations.

5 Practical Implementation & Storage Overhead

One of the most important strengths of VTS-cache is that its implementation does not require *any* modifications to the existing cache structure. This is because VTS-cache simply augments a conventional cache with the victim tag store. The victim tag store only decides the insertion policy for a *missed* cache block. Therefore, any in-cache monitoring mechanism that is used to improve performance, including the cache replacement policy, is left unchanged. The main source of hardware overhead in the VTS-cache comes from the VTS itself. In this section, we describe a practical implementation of the VTS using Bloom filters [3] and evaluate its storage overhead.

5.1 Practical Implementation

A naive implementation of the VTS would be to implement it as a set-associative structure and keep track of evicted blocks on a per-set basis. However, such an implementation will have a huge storage overhead and also consume a lot of static and dynamic power. For a practical, low-overhead implementation of VTS-cache, we modify the design of the VTS to make it implementable using Bloom filters [3].

A Bloom filter is a probabilistic data structure used as a compact representation of a large set. New elements can be inserted into the filter and elements can be tested if they are present in the filter. However, the test operation can have false positives, i.e., it can falsely declare an element as being present in the set. Also, once inserted, the only means of removing elements from a Bloom filter is to clear the filter completely. Since the VTS is only used as a prediction mechanism, the false positives do not lead to any correctness issues. However, for implementing VTS using a Bloom filter, we need to eliminate the *remove* operations (as shown in Figure 1b) from the VTS design.

There are two cases when a block address is removed from the VTS. One, when a missed block is present in the VTS, it is removed from the VTS. We get rid of this delete operation by simply leaving the block address is the VTS. The second case is when the VTS becomes full and block addresses have to be removed from the tail to accommodate more insertions. To avoid this remove operation, we propose to clear the VTS completely when it becomes full. Since the VTS only keeps track of block addresses, neither modification leads to any consistency issues.

With these changes, the VTS can be implemented using a Bloom filter and a counter that keeps track of the number of addresses currently present in the VTS. When a block gets evicted from the cache, its address is inserted into the filter and the counter is incremented. On a cache miss, the cache tests if the missed block address is present in the filter. When the counter reaches a maximum (size of the VTS), the filter and the counter are both cleared.

It is worth mentioning that Bloom filters are widely used in hardware [5, 20], especially in low-power devices to filter away costly operations. Therefore, implementing and verifying VTS in hardware should be straight forward. Also, since VTS-cache does not introduce any modifications to the cache itself, it further reduces the design complexity.

For our VTS implementation, we use a Bloom filter which uses an average of 8-bits per address. We use the state-of-the-art multiply-shift hashing technique [7] which can be easily implemented in hardware and is also less expensive in terms of latency and dynamic energy. Our implementation has considerably low false positive rate ($< 0.5\%$).

5.2 Storage overhead

The main source of storage overhead in VTS Cache is the Bloom filter that implements the VTS. The size of the Bloom filter depends on the maximum number of elements that it has to hold (M) and the average number of bits used per element stored in the filter (α). For our evaluations, the value of M, i.e., the size of the VTS, is same as the number of blocks in the cache, N. Therefore, the percentage storage overhead of the VTS compared to the cache size in terms of α, M, N, the cache block size (B), and the average tag entry size per block (T) is given by,

$$\text{\% Storage overhead of VTS compared to cache size} = \frac{\text{Bloom filter size}}{\text{Cache Size}}100\% = \frac{\alpha M}{(T+B)N}100\% = \frac{\alpha}{T+B}100\%$$

Thus, the percentage storage overhead of VTS is independent of the cache size itself. Rather, it depends only on α, the cache block size (B) and the average tag entry size (T). In our evaluations, we use $\alpha = 8$ bits, $B = 64$ bytes and $T > 2$ bytes. For this configuration, the percentage storage overhead of VTS compared to the cache size is less than 1.5%.

6 Prior Work on Block-level Insertion Policies

Prior research has identified and studied the significance of choosing the insertion policy on a per-block basis. In this section, we describe three such proposals to which we quantitatively compare VTS-cache. One of them is a instruction-pointer based approach called single-usage block prediction [24]. The other two work based on block addresses similar to VTS-cache: run-time cache bypassing [13] and adaptive replacement cache [21]. As we will show in our evaluations (Section 8), VTS-cache performs better than these three approaches.

6.1 Single-usage Block Prediction SUB-P

Piquet et al. [24] make the observation that a majority of blocks that are evicted from the last-level cache without being reused at all (called single-usage blocks) are loaded by a few instructions. Based on this observation, they propose a mechanism, single-usage block prediction (SUB-P), that identifies such instructions and predicts that blocks loaded by them will never be accessed again. SUB-P *marks* such blocks at the time of insertion and forces the replacement policy to evict such marked blocks first. To account for phase changes, a small fraction of single usage blocks are inserted without marking.

VTS-cache is more general than SUB-P because it can reduce the harmful performance effects of not only single-usage blocks but also blocks that exhibit very low temporal locality. In fact, VTS-cache will predict all single-usage blocks to have low temporal locality as they will not be present in the VTS the only time they are accessed.

6.2 Run-time Cache Bypassing (RTB)

Johnson et al. [13] propose a mechanism to compare the temporal locality of a missed block to that of the block about to be replaced. Based on the result, the missed block is either inserted normally into the cache or bypassed. The key observation behind their mechanism is that there is a spatial correlation in the reuse behavior of blocks, i.e., blocks that are close to each other in memory tend to show similar reuse behaviors. RTB keeps track of reuse counts of macro blocks (1KB regions in memory) in a table called memory address table (MAT) on chip. On a cache miss, the counter values for the regions corresponding to the missed block and the to-be-evicted block are compared. If the counter for the missed block is lower than that of the to-be-evicted block, then the missed block bypasses the cache.

One main disadvantage of RTB over VTS-cache is that within a macro block, it cannot distinguish between a single block accessed k times and k blocks accessed once each. This can lead to mispredictions which can cause blocks with bad locality getting inserted into the cache with high priority. Also, RTB requires a MAT access on every cache access (hit/miss). On the other hand, VTS-cache only accesses the VTS on a cache miss. It does not modify the cache hit operation at all.

6.3 Adaptive Replacement Cache (ARC)

Adaptive replacement cache is a self-tuning page replacement policy proposed for DRAM memory management. ARC adapts to different phases within an application that benefit from caching either recently used pages or frequently used pages. ARC achieves this by dividing the set of in-memory pages into two lists, one for recency and another for frequency, and maintaining a precise history of recently evicted pages. The sizes of the two lists are controlled by a self-tuning parameter based on how often missed pages hit in the history.

Although ARC is proposed as a page replacement policy, it can be easily evaluated in a on-chip setting. However, since ARC considerably modifies the cache structure and also requires precise history for tuning its parameters, its hardware implementation incurs high storage overhead and design complexity. For this reason, most prior works have dismissed ARC as an on-chip cache management mechanism. But we

Mechanism	**Storage overhead**	**Changes to cache?**	**Modifies hit behavior?**
SUB-P [24]	14KB for instruction tags to tag store + prediction table* (1 KB)	Requires additional information in the tag store.	Updates to the prediction table.
RTB [13]	≈ 3KB for a 1024 entry MAT*	No changes to cache	Updates to the memory access table.
ARC [21]	≈ 32KB for per-set history	Separates a cache set into two lists (one for frequency & one for recency)	Possible movement from one list to another.
VTS-Cache	8KB for Bloom filter	No changes to cache	No changes to cache hits.

Table 1: Overhead and Design Complexity of Different Mechanisms. *In our evaluations for SUB-P and RTB, we use an infinite sized table.

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

Table 2: Main configuration parameters used for simulation

compare VTS-cache to ARC for completeness. Table 1 presents a comparison of the storage overhead and design complexity of the different mechanisms for a 16-way associative 1MB cache using 64 byte blocks.

7 Evaluation Methodology

We use an event-driven 32-bit x86 simulator that models in-order cores. All systems use a three level cache hierarchy. The L1 and L2 caches are private to individual cores and the L3 cache is shared across all the cores. We don't enforce inclusion in any level of the hierarchy. All caches uniformly use a 64B cache block size. Writebacks do not update the replacement policy state. Other major simulation parameters are provided in Table 2.

For evaluations, we use benchmarks from SPEC CPU2000 and CPU2006 suites, three TPC-H queries, a TPC-C server and an Apache web server. All results are collected by running a representative portion of the benchmarks for 500 million instructions. We classify benchmarks into nine categories based on their cache sensitivity (low, medium or high) and 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 the performance degradation as a metric that determines sensitivity. We define a benchmark's intensity as the number of L2 cache misses per 1000 instructions (L2-MPKI). Benchmarks with L2-MPKI less than one are not evaluated in our studies as they do not exert any pressure on the last-level cache. Table 3 shows the intensity (under the L2-MPKI column) and cache sensitivity (under the Sens. column) of different benchmarks used in our evaluation.

We evaluate single-core systems and multi-programmed workloads running on 2-core and 4-core

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

Table 3: Classification of benchmarks based in intensity and cache sensitivity (L - Low, M - Medium, H - High). `L2-MPKI` is the number of L2 misses per kilo instructions and `Sens.` (sensitivity) is the % degradation in performance going from a 1 MB L3 to a 256 KB L3.

Mechanism	Label	Implementation
Thread-aware DIP [10]	TA-DIP	Feedback based set-dueling, 32 dueling sets
Thread-aware DRRIP [11]	TA-DRRIP	$\text{RRPV}_{\text{max}} = 7$, Hit priority, feedback based set dueling
Single usage block prediction [24]	SUB-P	Infinite size predictor table, RRIP replacement policy
Run-time cache bypassing [13]	RTB	Infinite size memory address table, RRIP replacement policy
Adaptive Replacement Cache [21]	ARC	Per-set history of evicted blocks, RRIP replacement policy
VTS Cache	VTS	Bloom filter (8 bits per element), RRIP replacement policy
VTS Cache with set dueling	D-VTS	VTS Cache + set dueling to determine all-LRU workload

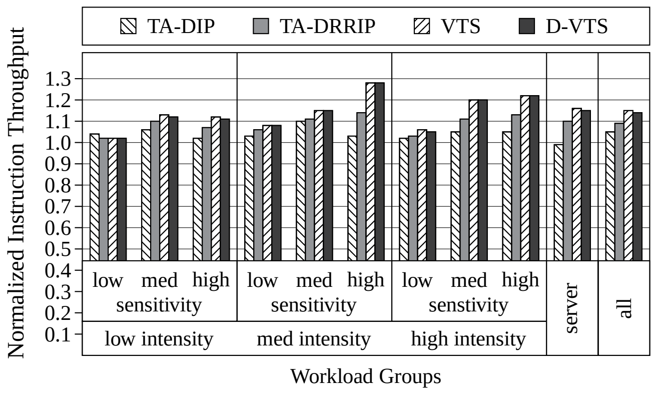
Table 4: List of evaluated mechanisms along with their implementation.

CMPs. We generate our multi-programmed workloads with nine different levels of aggregate intensity (low, medium or high) and aggregate sensitivity (low, medium or high). For 2-core simulations, we generate approximately 20 workloads in each category. For 4-core simulations, we generate between 10 to 15 workloads in each category. The server benchmarks are evaluated separately with ten 2-core and five 4-core workload combinations. In all, we present results for 208 2-core workloads and 135 4-core workloads.

Metrics: We compare performance using two metrics: weighted speedup [35] and instruction throughput. For evaluating fairness, we use the maximum slowdown metric. A lower maximum slowdown indicates better fairness.

$$\begin{aligned}\text{Instruction Throughput} &= \sum_i \text{IPC}_i \\ \text{Weighted Speedup} &= \sum_i \frac{\text{IPC}_i^{\text{shared}}}{\text{IPC}_i^{\text{alone}}} \\ \text{Maximum Slowdown} &= \max_i \frac{\text{IPC}_i^{\text{alone}}}{\text{IPC}_i^{\text{shared}}}\end{aligned}$$

Mechanisms: Table 4 provides the references to the five different mechanisms to which we compare VTS-cache to. We also mention the specific implementation parameters for each of those mechanisms.



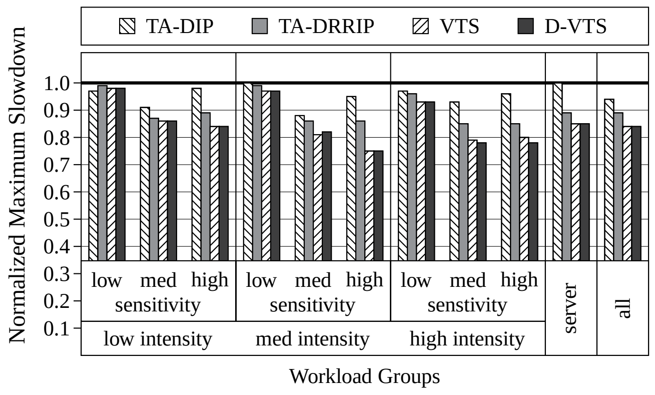


Figure 4: VTS-cache vs other mechanisms for 2-core systems. Left: Performance. Right: Unfairness

8 Results & Observations

In this section, we present and discuss the results of our evaluations comparing VTS-cache with the other prior mechanisms. We initially present the case for the block-level insertion policy approach by comparing VTS-cache with TA-DIP [10, 25] and TA-DRRIP [11] across a variety of system configurations. To show the effectiveness of our proposed approach, we present results comparing VTS-cache to single-usage block prediction [24], run-time cache bypassing [13] and adaptive replacement cache [21].

2-Core Results: [htp]

Figure 4 compares the system performance of VTS-cache with TA-DIP and TA-DRRIP on a 2-core system. The results are classified based on workload category. Averaged over all 208 2-core workloads, VTS-cache improves system throughput by 15% compared to baseline and 6% compared to TA-DRRIP. It also reduces unfairness by 16% compared to baseline and 5% compared to TA-DRRIP.

One major trend is that, for a given aggregate intensity, the performance improvements of all the mechanisms increase with increasing aggregate sensitivity. VTS-cache outperforms other mechanisms for all categories except the low-intensity low-sensitivity category. Workloads in this category rarely access the L3 cache and also benefit less from more cache space. In fact, none of the prior approaches improve performance significantly compared to the baseline, indicating that there is little scope for improvement.

For the server workloads, VTS-cache and TA-DRRIP drastically improve performance over the baseline (16% and 10% respectively). This is because these workloads have a lot of *scans* that can evict useful blocks from the cache. Both VTS-cache and TA-DRRIP are designed to mitigate the effect of such scans. However, TA-DRRIP does this by monitoring blocks *after* inserting them into the cache. On the other hand, VTS-cache identifies that these blocks have low temporal locality *before* insertion and inserts most of them with the lowest priority, thereby providing better performance than TA-DRRIP.

4-Core Results:

Figure 5 shows the corresponding results for 4-core systems. The observations are similar to those made for 2-core systems. VTS-cache significantly improves system throughput (21% over baseline and 8% over TA-DRRIP) and reduces unfairness (31% over baseline and 12% over TA-DRRIP). Again for the server workloads, VTS-cache and TA-DRRIP improve performance significantly over the baseline (17% and 11% respectively).

Figure 6 plots the weighted speedup improvements compared to LRU for all the 2-core and 4-core workloads, sorted based on improvements due to VTS. Two observations can be made from the plots. One,

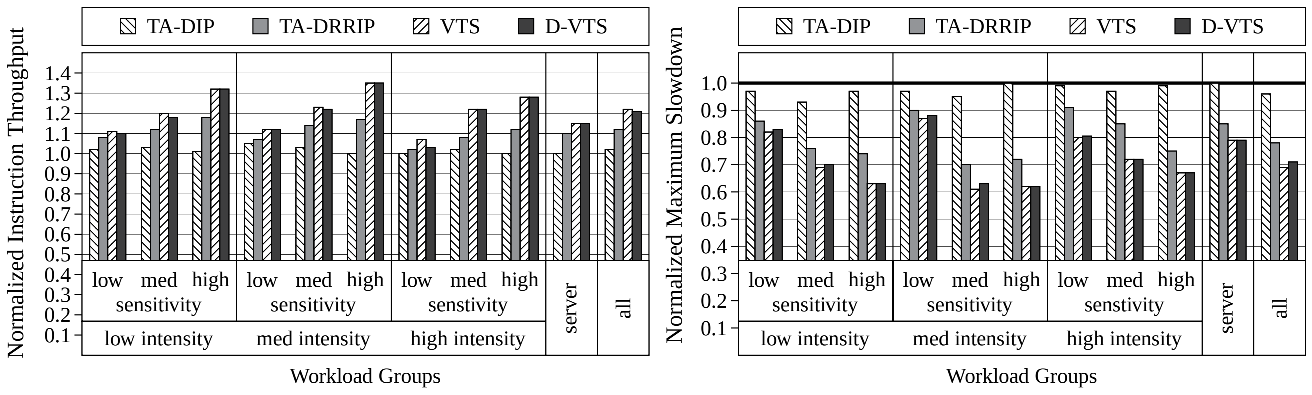


Figure 5: VTS-cache vs other mechanisms for 4-core systems. Left: Performance. Right: Unfairness

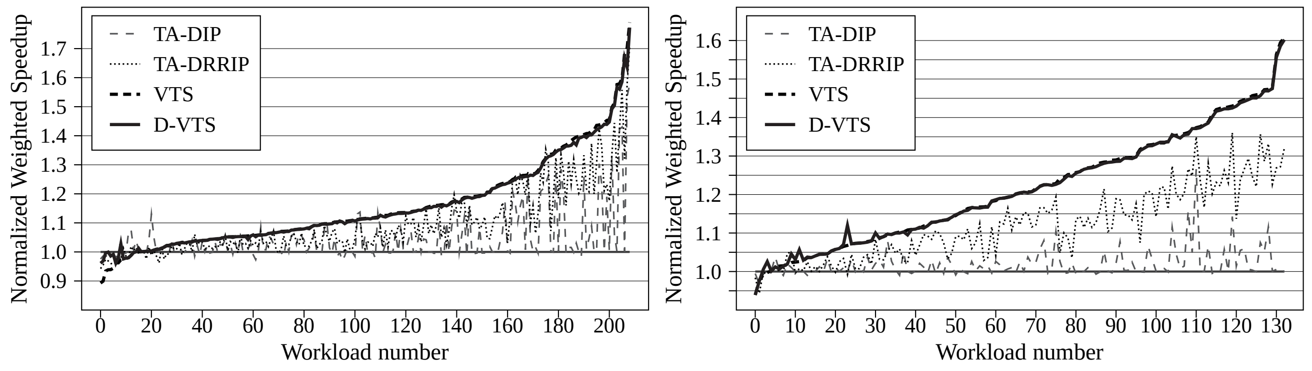


Figure 6: Normalized weighted speedup improvements over LRU for all workloads. Left: 2-core. Right: 4-core

the average improvements of VTS-cache are not due to a small number workloads. Rather, VTS-cache consistently outperforms prior approaches for most workloads. Two, as we described in section 4.2, VTS-cache can significantly affect performance for an LRU-friendly workload. This can be seen towards the left end of the curves where LRU outperforms all prior mechanisms. D-VTS mitigates the performance loss due to VTS-cache for these workloads by ignoring the VTS. For all other workloads, there is no significant difference between VTS and D-VTS making the latter a more robust mechanism.

Varying the Size of VTS:

Figure 7 shows the effect of varying the VTS size as a fraction of the number of blocks in the cache. As the figure indicates (and as discussed in section 4.1), VTS-cache provides maximum performance when the size of the VTS is same as the number of blocks in the cache.

Interaction with Replacement Policy:

VTS-cache can be used with any cache replacement policy. Figure 8 shows the performance improvement of adding VTS to a cache following the LRU replacement policy and one following the RRIP [11] replacement policy for 2-core workloads. As the figure shows, VTS-cache consistently improves performance in both cases for all workload categories (11% on an average for LRU and 12% for the RRIP policy). In fact, VTS-cache has the potential to be combined with any mechanism that works with blocks that are already present in the cache. This is because VTS-cache filters away blocks with low temporal locality and allows such mechanisms to work with potentially useful blocks.

Varying the Cache Size:

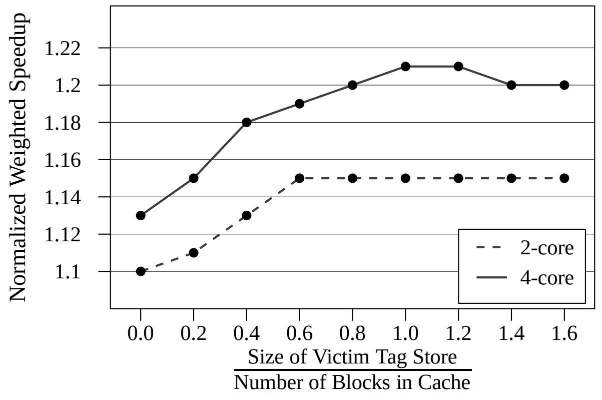


Figure 7: System Throughput vs Size of VTS

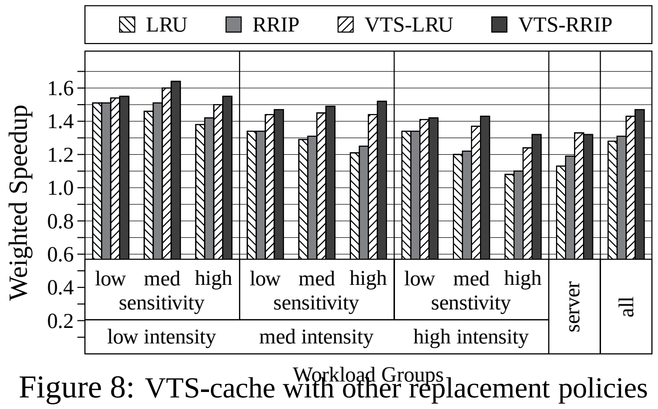


Figure 8: VTS-cache with other replacement policies

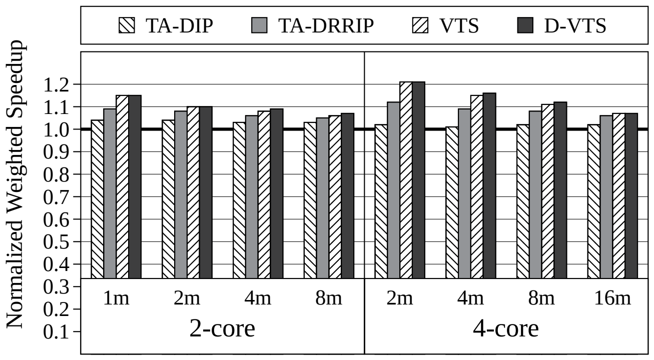


Figure 9: VTS-cache with different cache sizes

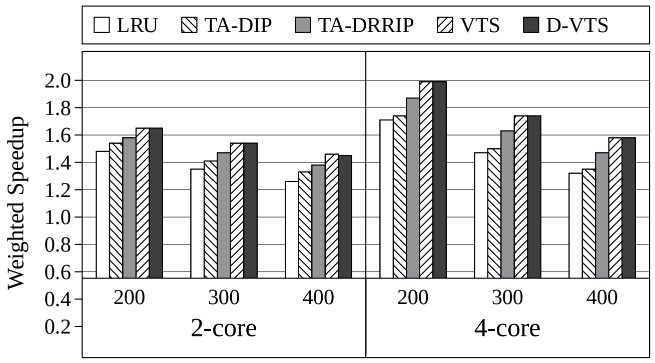
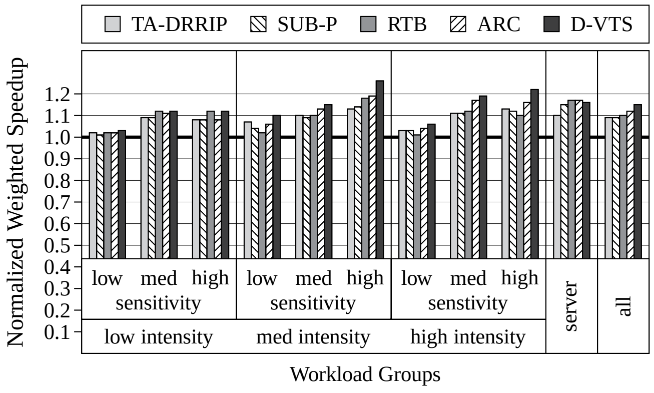


Figure 10: VTS-cache with different memory latencies



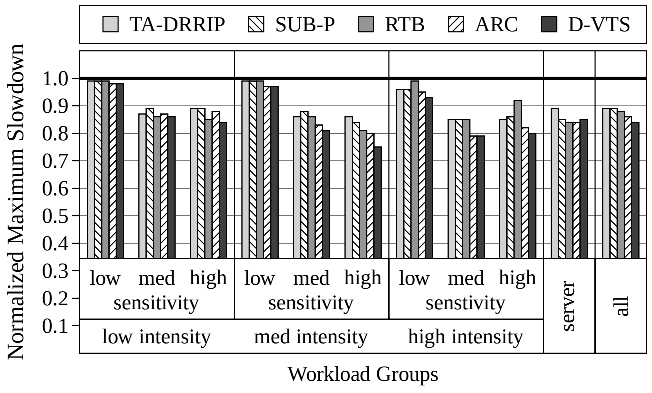
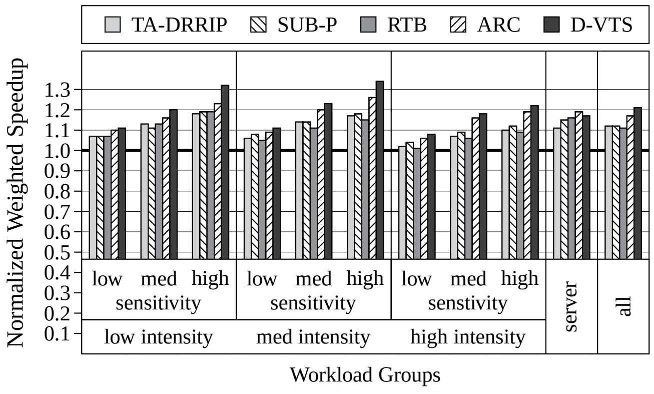


Figure 11: VTS-cache vs SUB-P, RTB & ARC for 2-core systems. Left: System throughput. Right: Unfairness



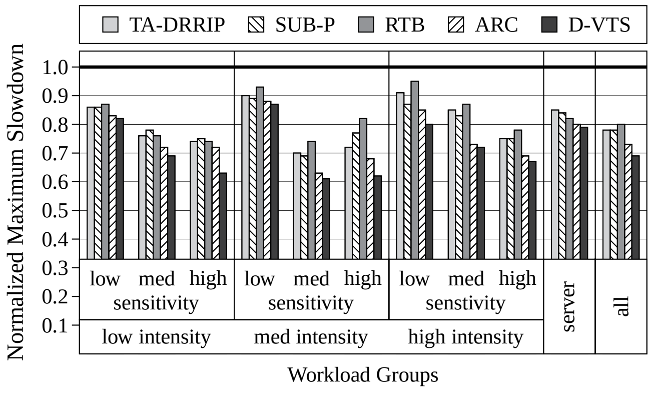


Figure 12: VTS-cache vs SUB-P, RTB & ARC for 4-core systems. Left: System throughput. Right: Unfairness

Figure 9 shows the effect varying the cache size on system throughput improvement using different mechanisms. As expected, the improvements due to different mechanisms decreases as the cache size increases. However, VTS-cache consistently outperforms other mechanisms. We conclude that VTS-cache is effective even with large cache sizes.

Sensitivity to Memory Latency:

Figure 10 shows the effect of varying the memory latency. For these experiments, we use a fixed latency for all memory requests. As expected, system throughput decreases as the memory latency increases. However, the performance benefit of VTS-cache over other mechanisms increases with increasing memory latency. In future multi-core systems, bandwidth constraints will lead to increase in average memory latency, a trend that is favorable for VTS-cache.

Comparison to RTB, SUB-P and ARC:

Figures 11 and 12 compares VTS-cache to single usage block prediction, run-time cache bypassing and adaptive replacement cache on 2-core and 4-core systems respectively. The figure also shows the results for TA-DRRIP for reference. The results indicate that overall VTS-cache performs better than all the other approaches. The plots also show that mechanisms that employ a per-block insertion policy based on their reuse behavior perform comparably or better than TA-DRRIP which chooses the insertion policy on an application granularity.

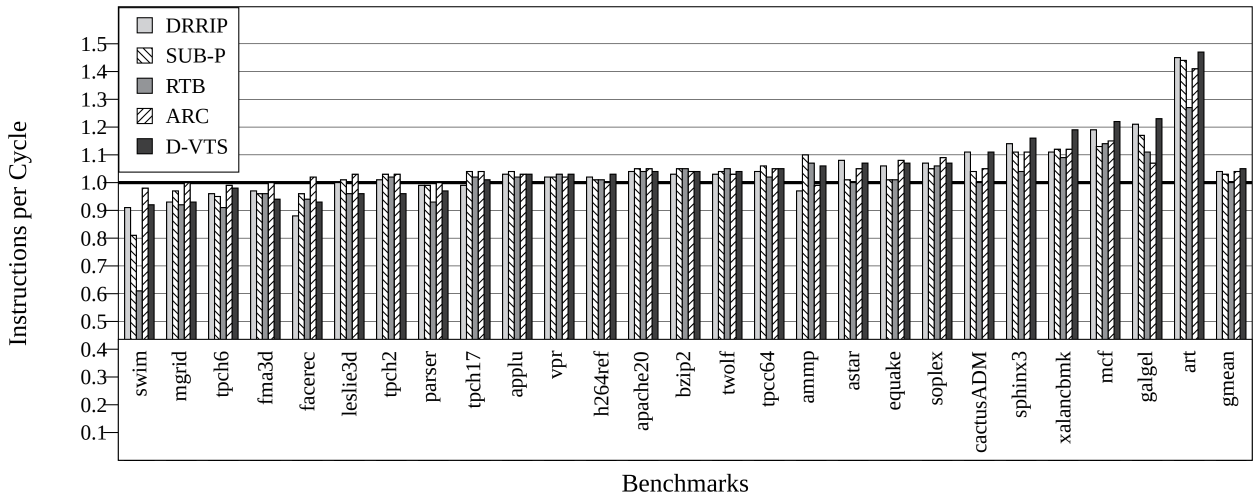


Figure 13: VTS-cache vs other mechanisms for the single core configuration. Benchmarks are sorted based on performance improvements due to VTS

Among the prior approaches, ARC outperforms other mechanisms for most workloads. However, VTS-cache, which is simpler and easier-to-implement mechanism performs better than ARC for most multi-core workloads (3.5% on an average) except the server workloads.This is because, for the server workloads, *scans* are the major source of performance degradation. ARC completely discards these scans whereas VTS-cache will insert a small fraction of such blocks with high priority due to the underlying bimodal insertion policy it uses for blocks with predicted low temporal locality.

Single-core Results:

Figure 13 compares the performance (normalized to LRU) of D-VTS with DRRIP, SUB-P, RTB and ARC. D-VTS improves performance by 5% compared to the baseline. It also provides better overall performance than other approaches. Although the performance improvements due to VTS-cache for single core are not as significant compared to that for multi-core systems, we present the results to show that VTS-cache does not significantly degrade single core performance. The plot indicates that VTS-cache loses performance mainly for LRU-friendly applications. Barring ARC, VTS-cache performs comparably or better than the other approaches for most of the applications.

9 Related Work

The main contribution of this paper is a low-complexity cache management mechanism, VTS-cache, that dynamically estimates the temporal locality of a cache block before insertion (using a structure called victim tag store) and decides the insertion priority on a per-block basis. We have already provided extensive qualitative and quantitative comparisons to the most closely related work in cache management [10, 11, 13, 24, 21], showing that VTS-cache outperforms all these approaches. In this section, we present other related work.

Block-level Insertion Policies: Tyson et al. [38] propose a mechanism to tag load instructions as cacheable or non-allocatable based on which blocks loaded by these instructions are either cached or bypassed. The mechanism itself was proposed for L1 caches and is very similar in approach to single usage block prediction [24]. Rivers et al. [29] propose a block-based approach similar to run-time cache bypassing [13] to improve the performance of direct mapped caches. [30] compares some of the above mentioned approaches and shows that block address based approaches work better than instruction pointer based approaches for secondary caches. We show in this paper that VTS-cache performs better than the run-time cache bypassing mechanism.

Cache Replacement Policies: Much prior research [2, 8, 11, 16, 19, 28, 33, 34] has focused on improving hardware cache replacement policies. Researchers have also paid attention to improving cache utilization [27, 31, 32, 41] by addressing the set imbalance problem. The insertion policy using VTS, proposed in this paper, can be easily coupled with any of these mechanisms to further improve cache performance.

Virtual Memory Page Replacement Policies: A number of page replacement policies [1, 12, 14, 18, 21, 23] have been proposed to improve the performance of the virtual memory subsystem. As these mechanisms were designed for software-based DRAM buffer management, they usually employ sophisticated algorithms and can use large amounts of storage. As a result, extending them to hardware caches incurs high storage overhead and implementation complexity in contrast to our low-cost VTS-cache design.

Victim Cache: Jouppi proposed victim caches [15] to improve the performance of direct mapped caches by reducing conflict misses. The key idea is to cache some of the recently evicted blocks in a fully-associative buffer. Even though the ideas might sound similar, the goal of VTS-cache is completely different. VTS-cache aims at preventing blocks with poor temporal locality from polluting the cache. Also, VTS-cache stores only the tags and not the data blocks themselves.

Shared Cache Management Mechanisms: With the advent of multi-cores, a number of mechanisms to improve the performance and fairness of on-chip shared caches have been proposed. Cache partitioning [26, 36, 37, 40] is one technique that has been effectively used to improve performance. VTS-cache can be coupled with many of these strategies by allowing them to work with blocks with potentially good temporal locality. The same applies to mechanisms for improving fairness and QoS in multi-core systems with shared caches [4, 9, 17, 22]. These approaches use soft partitioning to ensure that applications are guaranteed some amount of cache space. Since blocks with low temporal locality contribute neither to system performance nor to fairness, these QoS mechanisms can be employed in conjunction with VTS-cache. While VTS-cache can improve performance by retaining only the most useful blocks in the cache, these mechanisms can ensure fairness among different applications.

10 Conclusion

We presented VTS-cache, a cache management mechanism that determines the cache insertion policy on a per-block basis. The key idea is to predict a missed block's temporal locality before inserting it into the cache and choose the appropriate insertion policy for the block based on that temporal locality prediction. We present a new technique for estimating the temporal locality of a block before it is inserted into the cache by monitoring the addresses of recently evicted blocks. We provide a practical, low-overhead implementation of VTS-cache using Bloom filters.

Based on our evaluations, we conclude that VTS-cache provides significantly better system performance compared to other similar approaches on a wide variety of workloads and system configurations. It also improves fairness for the multi-core systems we evaluated. Our future work will include developing and analyzing other temporal locality prediction schemes and also investigating the interaction of our mechanism with prefetching.

References

[1] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *FAST-3*, 2004. 19

[2] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *MICRO-40*, 2007. 19

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM Communications*, 13:422–426, July 1970. 2, 9

[4] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS-21*, 2007. 19

[5] Y. Chen, A. Kumar, and J. Xu. A new design of bloom filter for packet inspection speedup. In *GLOBECOM*, 2007. 10

[6] J. Collins and D. M. Tullsen. Hardware identification of cache conflict misses. In *MICRO-32*, 1999. 1, 2

[7] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25:19–51, 1997. 10

[8] E. G. Hallnor and S. K. Reinhardt. A fully associative software managed cache design. In *ISCA*, 2000. 19

[9] R. Iyer. CQoS: a framework for enabling qos in shared caches of cmp platforms. In *ICS-18*, 2004. 19

[10] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT-17*, 2008. 2, 4, 5, 13, 14, 18

[11] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010. 2, 4, 5, 13, 14, 15, 18, 19

[12] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS*, 2002. 19

[13] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12), dec 1999. 1, 2, 10, 11, 12, 13, 14, 18

[14] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB-20*, 1994. 19

[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-17*, 1990. 19

[16] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD-25*, 2007. 19

[17] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT-13*, 2004. 19

[18] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transanctions on Computers*, 50:1352–1361, December 2001. 19

[19] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *MICRO-41*, 2008. 19

[20] M. J. Lyons and D. Brooks. The design of a bloom filter hardware accelerator for ultra low power systems. In *ISLPED*, 2009. 10

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST-2*, 2003. 2, 5, 11, 12, 13, 14, 18, 19

[22] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA-34*, 2007. 19

[23] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, 1993. 19

[24] T. Piquet, O. Rochecouste, and A. Seznec. Exploiting single-usage for effective memory management. In *ACSAC-12*, 2007. 1, 2, 10, 11, 12, 13, 14, 18

[25] M. K. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007. 2, 4, 5, 9, 14

[26] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO-39*, 2006. 19

[27] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way cache: Demand based associativity via global

replacement. In *ISCA-32*, 2005. 19

[28] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *MICRO*, '07. 19

[29] J. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *ICPP*, 1996. 1, 2, 18

[30] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ICS*, 1998. 19

[31] D. Rolan, B. Fraguela, and R. Doallo. Reducing capacity and conflict misses using set saturation levels. In *HiPC*, 2010. 19

[32] D. Rolán, B. B. Fraguela, and R. Doallo. Adaptive line placement with the set balancing cache. In *MICRO*, 2009. 19

[33] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *MICRO-43*, 2010. 19

[34] A. Seznec. A case for two-way skewed-associative caches. In *ISCA-20*, 1993. 19

[35] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGOPS Oper. Syst. Rev.*, 34:234–244, November 2000. 13

[36] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41:1054–1068, September 1992. 19

[37] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA-8*, 2002. 19

[38] G. S. Tyson, M. K. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO*, 1995. 1, 18

[39] 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-44*, 2011. 1, 2

[40] Y. Xie and G. H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA-36*, 2009. 19

[41] D. Zhan, H. Jiang, and S. C. Seth. STEM: Spatiotemporal management of capacity for intra-core last level caches. In *MICRO*, 2010. 19