# Block Value based Insertion Policy for High Performance Last-level Caches

Lingda Li, Junlin Lu, and Xu Cheng
Microprocessor Research and Development Center, Peking University, Beijing, China
{lilingda, lujunlin, chengxu}@mprc.pku.edu.cn

## ABSTRACT

Last-level cache performance has been proved to be crucial to the system performance. Essentially, any cache management policy improves performance by retaining blocks that it believes to have higher *values* preferentially. Most cache management policies use the access time or reuse distance of a block as its value to minimize total miss count. However, cache miss penalty is variable in modern systems due to i) variable memory access latency and ii) the disparity in latency toleration ability across different misses. Some recently proposed policies thus take into account the miss penalty as the block value. However, only considering miss penalty is not enough.

In fact, the value of a block includes not only the penalty on its misses, but also the reduction of processor stall cycles on its hits, i.e., *hit benefit*. Therefore, we propose a method to compute both miss penalty and hit benefit. Then, the value of a block is calculated by accumulating all the miss penalty and hit benefits of its requests. Using our notion of block value, we propose *Value based Insertion Policy (VIP)* which aims to reserve more blocks with higher values in the cache. VIP keeps track of a small number of incoming and victim block pairs to learn the relationship between the value of the incoming block and that of the victim. On a miss, if the value of the incoming block is learned to be lower than that of the victim block in the past, VIP will predict that the incoming block is valueless and insert it with a high eviction priority. The evaluation shows that VIP can improve cache performance significantly in both single-core and multi-core environment while requiring a low storage overhead.

## Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*cache memories*

## General Terms

Design, Performance

## Keywords

Value; Miss penalty; Hit benefit; Insertion; Last-level cache

## 1. INTRODUCTION

Cache performance, especially last-level cache performance, is crucial to the system performance due to the increasing memory access latency. In order to reduce the processor stall cycles on cache misses, a large amount of cache management policies have recently been proposed. Essentially, any cache management policy improves cache performance by retaining blocks which it believes to have higher *values*[1] longer, and replacing valueless blocks preferentially. Most of recent proposals treat the access time or reuse distance of a cache block as its *value* to reduce the absolute cache miss count [14, 25, 27, 29, 38, 40, 46, 48]. These proposals implicitly assume that all cache requests result in equal performance degradation when they miss in the cache. However, this assumption is inaccurate in modern systems for two reasons.

The first reason is that memory access latency is not uniform in modern systems. First, in modern DRAM systems, memory requests need additional serving time when request queue and bank conflicts occur, and memory requests that access the same row as previous requests are served faster. Second, the interconnect network introduces variable access latency in a non-uniform memory access system. Moreover, in future systems, multiple memory techniques can be used simultaneously (e.g., eDRAM, PCRAM, STT-RAM [47]). Such systems also tend to use hybrid memory hierarchies such as 3D-stacked DRAM caches [32]. Therefore, the disparity in memory latency will become even larger.

The other reason for the disparity in miss penalty is that modern processors make use of various techniques such as non-blocking caches [28] and prefetching [45] to serve multiple cache misses in parallel. Using these techniques makes the processor stall cycles on a cache miss depend not only on its memory access latency, but also on other concurrent misses. The memory access latency of cache misses which can be served in parallel can be partly hidden by that of other concurrent misses, and these misses thus have small performance impact. On the other hand, cache misses that occur in isolation can incur greater performance loss.

The variation of miss penalty suggests that the cache management policy should use the miss penalty as a part of block value. Some recently proposed cache management policies take into account the miss penalty on replacement [16, 19,

[1]In this paper, the value of a block refers to its worth or merit, not its data value.

22, 39, 43]. These proposals compute and record the miss penalty when a block is inserted into the cache, and preferentially replace blocks with small miss penalty.

However, only considering miss penalty is not enough. Let us assume that there are two blocks A and B: A has a miss penalty of 200 cycles on its insertion and receives no hit; B has a smaller miss penalty of 100 cycles, nevertheless, it receives 3 hits, and each hit prevents the processor from stalling for another 100 cycles. In such a scenario, reserving B will potentially save an aggregate penalty of 400 cycles, while it is 200 cycles for A. Thus, the cache should reserve B instead of A. However, if we only consider the penalty on cache misses, A will incorrectly be preferred.

Therefore, the *value* of a cache block depends on all its requests. It includes not only the increment of processor stall cycles on its misses (i.e., *miss penalty*), but also the reduction of processor stall cycles on its hits (i.e., *hit benefit*). To evaluate cache block values, we propose a simple hardware mechanism to compute both miss penalty and hit benefit for superscalar processors. Then, the value of a block can be calculated by accumulating all the miss penalty and hit benefits of its requests during a period of time.

Based on our notion of value, we propose *Value based Insertion Policy (VIP)* to improve cache performance by reserving more blocks with higher values. VIP learns and predicts the relationship between the value of the incoming block and that of the victim block selected by the baseline replacement policy on cache misses. On the insertion of an incoming block, if its value is predicted to be lower than that of the victim block, VIP will insert the incoming block with a high eviction priority, and thus that block will be replaced preferentially on following misses.

VIP can be applied to the current cache design with negligible modification, and it can cooperate with any baseline replacement policy. Moreover, VIP is both thread-aware and prefetch-aware.

We evaluate the performance of VIP with NRU, LRU, and SRRIP [14]. Our evaluation shows that VIP can improve cache performance significantly while requiring a low storage overhead. On average, VIP outperforms LRU by 8.2% and 5.9% for single-core and 4-core workloads respectively in the absence of prefetching, and it can also achieve significant performance improvement in the presence of prefetching. VIP also outperforms other state-of-the-art techniques, including SBAR [39], DIP [38], DRRIP [14], PIPP [48], UCP [40], and SHiP-PC [46].

The rest of this paper is organized as follows. Section 2 discusses some related work. Section 3 introduces the computation method for miss penalty and hit benefit and the notion of block value. Section 4 describes the design and implementation of VIP. Section 5 shows the experimental methodology, and then Section 6 analyzes the results. Finally, Section 7 concludes this paper.

## 2. RELATED WORK

Extensive research on cache management has been done to improve cache performance. Based on the goal, they can be classified into two categories: miss count based policies which aim to reduce the miss count, and miss penalty based policies which aim to reduce the miss penalty. We will introduce the primary work of these two types of policies respectively in this section.

### 2.1 Miss Count based Cache Management

A lot of studies propose to improve cache performance by reducing the miss count. DIP [38] dynamically inserts most incoming blocks into the LRU position to avoid thrashing when the working set is larger than the cache size. Pseudo-LIFO [3] uses a fill stack and prioritizes to replace blocks on the top of fill stack. Keramidas *et al.* proposed to explicitly predict the reuse distance to guide replacement [24]. RRIP [14] distinguishes reused blocks from no reused ones, and evicts no reused blocks preferentially. SHiP [46] can further improve the performance of RRIP with a signature-based re-reference interval predictor, and their signatures include memory region, PC, and instruction sequence. PDP [5] protects cache blocks within a predicted reuse distance. Jiménez proposed a low cost replacement policy based on tree-based Pseudo LRU [20].

Dead block prediction techniques try to identify and preferentially evict blocks that will not be accessed again (i.e., *dead blocks*). Dead block prediction can be classified into three categories based on how to identify dead blocks: trace based [29], time based [10], and counter based [27]. Cache burst predictor [31] improves dead block prediction accuracy by making prediction for continuous access sequences. SDBP [25] samples a part of sets to reduce conflicts in the predictor for low overhead and high prediction accuracy.

Bypass techniques improve cache performance by bypassing blocks with poor locality. Based on how to predict the locality, these techniques can be classified into address based [15, 21, 41] and PC based [4, 8, 44]. DSB [6] adjusts the bypass probability based on the reuse order of the incoming block and the victim block. Gaur *et al.* proposed a bypass and insertion algorithm for exclusive last-level caches [7]. OBM [30] learns and predicts the behavior of the optimal bypass to make bypass decisions.

To improve shared cache performance, some recent work proposed to partition shared caches to minimize the aggregate miss count of multi-core processors. UCP [40] collects the cache utility information for each core to allocate cache space for minimizing the total miss count. TADIP [13] extends DIP to select the best insertion policy for each core. PIPP [48] adjusts the insertion and promotion policy of different cores to partition shared caches implicitly. NUcache [33] improves shared cache performance by retaining blocks accessed by selected PCs longer. Vantage [42] partitions shared caches at cache block granularity to make it applicable in many-core systems. PriSM [34] adjusts the eviction probabilities of different cores to partition shared caches.

Miss count based cache management policies implicitly assume that all cache misses are equal. However, the penalty of different misses can change dramatically in modern systems. Thus, only reducing the miss count is not enough, and it is important for cache management policies to take into account the variation of miss penalty.

### 2.2 Miss Penalty based Cache Management

Jeong and Dubois first proposed to take into account the miss cost in cache management [16, 17, 18]. They extend LRU to distinguish the miss cost of local memory from that of remote memory in CC-NUMA multiprocessors. In uniprocessor environment, Jeong *et al.* proposed to distinguish the miss penalty between load and store misses [19].

Critical cache [22] and LACS [26, 43] both estimate the miss penalty using the number of issued instructions during

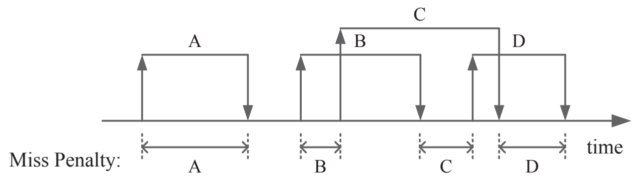


Figure 1: Example for miss penalty computation of demand misses.

**Algorithm 1** Compute miss penalty for demand misses. (called every cycle)

```
1: for each valid MSHR entry $X$ do
2:     if $X$ holds a demand miss then
3:         if $C_{\text{demand\_miss}} == 1$ then
4:             $X.T_{\text{p}}$ += 1;
5:         end if
6:     end if
7: end for
```

the cache miss. Critical cache dedicates a part of cache to preserve critical loads, and LACS replaces blocks with small miss penalty preferentially.

Qureshi *et al.* proposed to take into account the variation of miss penalty since modern processors can serve multiple misses in parallel, which is called the Memory Level Parallelism (MLP) cost [39]. They use the reciprocal of the in-flight miss number to represent the MLP cost. On replacement, the MLP-aware policy selects the block with the minimum weighted sum of MLP cost and LRU stack position as the victim. In case that the MLP-aware policy can hurt performance sometimes, SBAR is introduced to adaptively select the policy that has better performance between the MLP-aware policy and LRU. MLP-DCP [36] and MCFQ [23] both partition shared caches based on the MLP cost.

These previous miss penalty based policies only take into account the miss penalty on the insertion of cache blocks, and they do not consider the received benefits on the hits. On the other hand, our policy takes into account both miss penalty and hit benefit to represent cache block values.

## 3. BLOCK VALUE

In order to reduce the total processor stall cycles on cache misses, caches should retain blocks that can hurt the performance most when they are not resident in the cache. Therefore, we use the aggregate increasing processor stall cycles if a block is not reserved by the cache as its *value*, which depends on all requests of that block. The value of a cache block includes both the processor stall cycles on its misses (i.e., *miss penalty*) and the reduction of processor stall cycles due to its hits (i.e., *hit benefit*). We need to compute and accumulate all miss penalty and hit benefits of a block to get its value. Therefore, at first, we will introduce how to compute miss penalty and hit benefit respectively for superscalar processors.

### 3.1 Miss Penalty Computation

Miss penalty reflects the increment of processor stall cycles due to the cache miss. We describe how to compute miss penalty based on the request type.

**Demand misses.** After a long-latency demand miss, the processor will run out of resources and stall soon. Therefore, the number of processor stall cycles on a miss can be approximated by the number of cycles that the cache spends on waiting for the memory.

Figure 1 presents an example for miss penalty computation, where A, B, C, and D denote different miss requests. For an isolated miss such as A, the processor has to wait until the memory returns its data. Therefore, the miss penalty of isolated misses is their memory access latency. For parallel misses such as B and C, their miss penalty is the non-overlapping fraction of their memory access latency, as shown in Figure 1. The reason behind it is that if a parallel miss is removed, only its non-overlapping fraction of memory access latency is reduced from the total cache serving cycles.

To compute miss penalty, we extend MSHR (Miss Status Holding Register) [28]. MSHR is used to record in-flight cache misses. Each cache miss is allocated with an MSHR entry before it is sent to memory, and the MSHR entry is released when the memory completes the request. We append each MSHR entry with a counter $T_{\text{p}}$ for its miss penalty computation. When an MSHR entry is allocated, its $T_{\text{p}}$ is initialized to 0. Each cycle, if there is only one demand miss, the $T_{\text{p}}$ of the isolated miss is increased by 1 to indicate that this miss should be responsible for this cache serving cycle. Otherwise, it indicates that there are either multiple or no demand misses, and no miss needs to take responsibility in either case. $C_{\text{demand\_miss}}$ is used to count the number of in-flight demand misses. Algorithm 1 shows the algorithm for computing demand miss penalty.

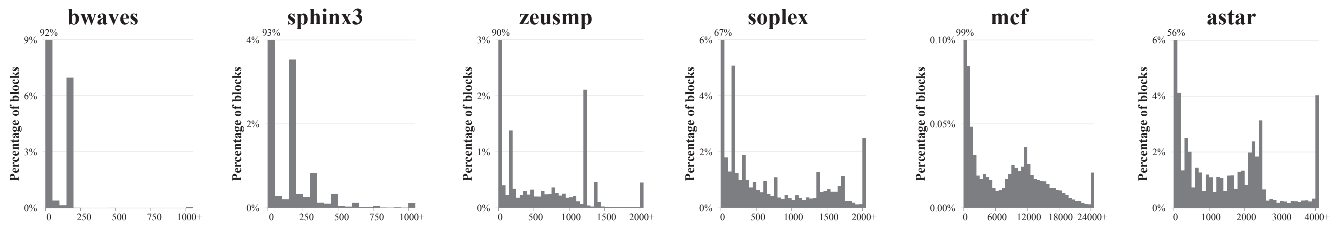
**Writeback and prefetch misses.** Writeback requests are generated by inner caches on replacement, and they do not affect the processor performance directly. Prefetch misses do not stall the processor, too. Thus, the miss penalty of writeback and prefetch requests is 0.

For demand misses, prefetch misses can affect their miss penalty in two ways. One is that for a prefetch request which is not timely, although the following demand request to the prefetched block cannot get the data immediately, its miss penalty becomes smaller. In such scenarios, we only need to reset the corresponding $T_{\text{p}}$ and restart the miss penalty counting when the address of an in-flight prefetch request recorded in an MSHR entry is found to be identical to that of the current demand request. The other way is that prefetch requests contend with demand requests for memory bandwidth, which will eventually affect the memory access latency of demand requests. However, our method for miss penalty computation takes into account variable memory latency. Consequently, our miss penalty computation method is applicable in the presence of prefetching.

### 3.2 Comparison with Other Miss Penalty Computation Methods

Jeong and Dubois proposed the optimal offline miss cost based policy, and it uses two static costs to distinguish the miss penalty of local memory from that of remote memory in CC-NUMA multiprocessors [16]. When the latency toleration ability of modern processors is taken into account, using two static costs is inaccurate since miss costs can change dynamically and have many different values.

Critical cache [22] and LACS [26, 43] use the number of issued instructions during a cache miss to represent its penalty. The more the issued instructions, the lower the miss penalty. However, when memory access latency is long enough, the number of issued instructions will show no d-



(a) Low values. (b) Middle values. (c) High values.

Figure 2: Value distribution for SPEC CPU2006 representative programs. The x-axis represents block values in cycles.

ifference, and thus using it to estimate miss penalty is not accurate. Besides, such information is difficult to obtain by last-level caches.

The MLP cost computation method proposed by Qureshi *et al.* [39] takes into account both variable memory latency and latency toleration ability. When multiple misses are served in parallel, unlike our method, their method divides the serving cycles equally for all in-flight misses, which indicates that all concurrent misses share the responsibility for these cycles. Since the MLP cost computation needs division operations, their method is more complex. While our method only needs plus-one operations, and at most one adder is used in one cycle. We evaluate the performance of our proposed policy using the MLP cost computation method instead of ours in Section 6, and its performance is similar to that of our original policy.

## 3.3 Hit Benefit Computation

**Demand hits.** Hit benefit represents the reduction of processor stall cycles on a cache hit. To compute it, we pretend the hit request to be a miss and then use the miss penalty computation method introduced above.

Unlike miss requests, cache hit requests do not need to access the memory and allocate MSHR entries. Thus, we propose a structure called *Hit Benefit Calculator (HBC)* to record cache hit requests for hit benefit computation. HBC is similar to MSHR. Each HBC entry includes a valid bit, the address of request, a latency counter to count the remaining hypothetic memory access latency, and $T_b$ for hit benefit computation. On a demand hit, an HBC entry is allocated with its latency counter initialized to its hypothetic memory access latency and $T_b$ initialized to 0. Since we model a desktop-like system in our experiments, where the variation of memory latency is not large, we use a static estimated memory access latency (150 cycles) for all hit requests in our experiments. For more complex systems where memory latency can change dramatically, a small address indexed latency predictor can be potentially employed, since cache block address usually determines its location in the memory system, and thus roughly determines its access latency. Each cycle, the latency counter is decreased by 1, and the $T_b$ is increased by 1 if $C_{\text{demand\_miss}}$ is equal to 0, since in such scenarios, the hypothetic miss converted from the hit request will be an isolated miss. When the latency counter of an HBC entry is reduced to 0, its $T_b$ represents the hit benefit.

Computing hit benefits for all hit requests will require a large HBC. Fortunately, we only need to compute hit benefits for a small fraction of cache hits as we will introduce later, and thus a small HBC is enough.

**Writeback and prefetch hits.** Their hit benefits are always 0 since there is no influence on the processor execution time when they are converted into misses.

## 3.4 Analysis of Block Value

Using the methods described above, the value of a cache block can be calculated by accumulating both miss penalty and hit benefits of its requests within a certain period of time. We analyze the values of cache blocks for SPEC CPU2006 benchmarks in an LRU-managed 2MB L2 cache without prefetching. To calculate the block value in a longer time, we double the associativity of the L2 cache tag array, so we can accumulate all miss penalty and hit benefits of a block as its value from the time when it is placed in the 2MB cache to the time when it is evicted from the virtual 4MB cache. Figure 2 presents the block value distribution for some representative programs.

First, we observe that a large amount of blocks have 0 values. The reason is that since L1 caches filter out most of the locality, L2 cache requests generally show poor locality and many blocks receive no cache hits. For `mcf`, although 99% of blocks have 0 values, the remaining blocks have extremely large values ($>$ 10000 cycles). Therefore, it is important to make sure that 99% of valueless blocks do not interfere 1% of valuable blocks for `mcf`.

Our second and more important observation is that the disparity in block values is large. The cache requests of some programs always occur in isolation, such as `sphinx3`, and thus most of their block values are multiples of memory access latency ($\approx$ 150X cycles). While for other programs such as `zeusmp` and `soplex`, there are more parallel requests.

Moreover, in multi-core system where the cache is shared, the variation of values exists not only between different blocks of the same core, but also between blocks of different cores. For instance, when `bwaves` and `astar` are executing in parallel, the cache management policy should allocate more cache space for `astar` since its blocks have larger values.

The disparity in block values motivates the need for a block value based cache management policy, which can be aware of the variation of cache block values and retain blocks with higher values preferentially.

## 3.5 Predictability of Block Value

To enable block value based cache management, block values must be predictable. A simple scheme to predict the value of a block is to use the last value when it is retained by the cache. Note that we only use this simple value prediction scheme to study the predictability of block value, and we do not use it in our proposed policy. We evaluate the feasibility of this scheme by measuring the difference between the values of a block during its two successive cache residence time. The *absolute difference* is calculated by subtracting the smaller value from the larger one, and the *relative difference* is calculated by dividing the absolute difference by the larger value. If either relative difference $< 10\%$ or absolute difference $< 15$ cycles (10% of memory access latency) for two successive values of the same block, the value prediction is considered to be accurate.

Among the 16 SPEC CPU2006 benchmarks used in our experiments, for most programs, more than 80% of their value prediction is accurate. Only the block values of `bzip2`, `zeusmp`, and `gobmk` show poor predictability. Therefore, we conclude that the block value is predictable, and it can be used to guide the cache management policy.

## 4. VALUE BASED INSERTION POLICY

Our goal is to design a cache management policy that focuses on increasing the aggregate values of blocks reserved by the cache to improve performance. In order to increase the aggregate values of cache blocks, on a cache miss, if the value of the incoming block is lower than that of the victim block, the incoming block should be inserted with a high eviction priority, so that it can be evicted from the cache quickly.

To that end, we propose *Value based Insertion Policy (VIP)*. VIP makes insertion decisions by learning and predicting the relationship between the value of the incoming block and that of the victim.

### 4.1 Overview

Figure 3 shows the structure of VIP, where grey modules are added for VIP. VIP uses *Replacement History Table (RHT)* to compare the value of the incoming block with that of the victim block selected by the baseline replacement policy. On a cache miss, an RHT entry is allocated to keep track of the current incoming block and the corresponding victim. Each RHT entry contains a *value comparer (VC)* to record the difference between the value of the incoming block and that of the victim, and the VC is initialized to 0 on the allocation of an RHT entry. On the following access to the incoming block of an RHT entry, the miss penalty computed by the MSHR or the hit benefit computed by the HBC is added to its VC. While on the following access to the victim of an RHT entry, the miss penalty or hit benefit is subtracted from its VC. If VC exceeds a positive threshold, it indicates that the incoming block has a higher value. Otherwise, if VC exceeds a negative threshold, it indicates that the value of victim block is higher.

Then, a PC indexed saturating counter table called *Relative Value Prediction Table (RVPT)* is employed to record value comparison results. The numerical value of a counter in the RVPT indicates whether the value of the incoming block generated by the corresponding PC is larger than that of the victim block recently. All counters of RVPT are initialized to 0. When a value comparison completes, the PC of the incoming block is used to index the RVPT to update its corresponding counter: If the incoming block is learned to have a higher value, the counter is increased by 1; if the victim block has a higher value, it is decreased by 1.

On the insertion of an incoming block, VIP consults the RVPT to predict whether the incoming block has a smaller value. If the counter indexed by its PC is less than 0, it indicates that the values of previous incoming blocks accessed

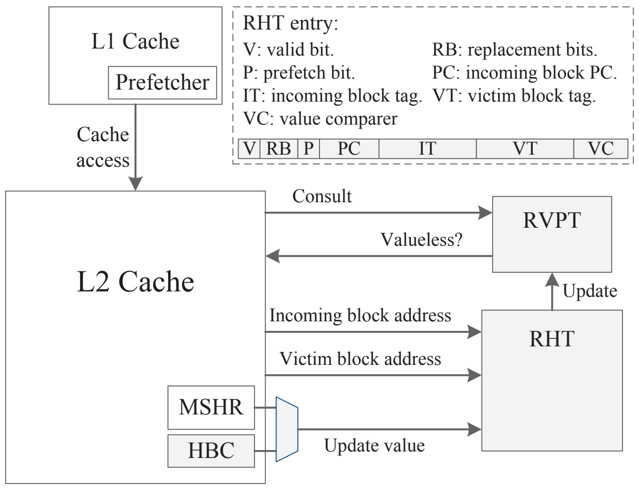


Figure 3: The structure of VIP.

by the same PC are smaller than those of the victim blocks. In such scenarios, VIP predicts that the current incoming block also has a smaller value. Otherwise, it indicates that the values of previous incoming blocks accessed by the same PC are larger than those of the victims, and thus VIP predicts the current incoming block is valuable and should be retained longer in the cache.

VIP associates each cache block with an *eviction bit (EB)* to indicate its eviction priority. If an incoming block is predicted to have a smaller value compared to the victim, its EB is set to 1 to indicate its high eviction priority. Otherwise, its EB is set to 0.

Upon replacement, blocks whose EBs are 1 are selected as the victim preferentially. If there are no such blocks, the baseline replacement policy is used to select the victim. In doing so, VIP prevents incoming blocks with smaller values from evicting blocks with higher values in the cache, and thus increases the aggregate values of cache blocks.

Algorithm 2 shows the detailed algorithm of VIP. The threshold of VC is chosen as the estimated memory access latency (150 cycles). The reason is that assuming the value of block A minus that of block B is larger than the memory latency, even though B shows a higher value in a long term, B can be brought into the cache on its next miss. Since the miss penalty on the next miss of B must be smaller than or equal to the memory latency, it is also smaller than the value difference between A and B, and thus the total values of cache blocks increase by reserving A first and then B next.

VIP can cooperate with any baseline replacement policy. In our experiments, we evaluate the performance of VIP with various replacement policies, including Not Recently Used (NRU) policy, LRU, and SRRIP [14].

### 4.2 Implementation Details

The RHT can be organized as fully associative, set associative, or direct mapped. Our experiments show that a 128-entry 8-way set-associative RHT is enough. Since the set number of RHT (16) is smaller than that of the cache (2048 for the 2MB cache in our experiments), the set index field of RHT (9th to 6th bits of address) is a subset of the cache set index field (16th to 6th bits of address for the 2MB cache). Hence, the incoming and victim blocks from the same cache set are also in the same RHT set. Each RHT

**Algorithm 2** The algorithm of VIP

```
On an access to block $X$:
if $X$ is a demand miss then
    benefit = ComputeMissPenalty();
else if $X$ is a demand hit then
    benefit = ComputeHitBenefit();
else
    benefit = 0;
end if
for each valid entry $A$ in the corresponding RHT set do
    if $X$.tag == $A$.IT then
        $A$.VC += benefit;
        if $A$.VC >= VALUE_THRESHOLD then
            RVPT[$A$.PC]++;
            Invalidate $A$;
        end if
    else if $X$.tag == $A$.VT then
        $A$.VC −= benefit;
        if $A$.VC <= −VALUE_THRESHOLD then
            RVPT[$A$.PC]−−;
            Invalidate $A$;
        end if
    end if
end for
if $X$ is a miss then
    $Y$ = Cache.SelectVictimEB1($X$);
    $Z$ = Cache.SelectVictimBaseline($X$);
    if $Y$ != NULL then
        Replace $Y$ with $X$;
    else
        Replace $Z$ with $X$;
    end if
    if RVPT[$X$.PC] >= 0 then
        $X$.EB = 0;
    else
        $X$.EB = 1;
    end if
    if RHT.Record($X$) == true then
        $B$ = RHT.SelectVictim($X$);
        $B$.PC = $X$.PC;
        $B$.IT = $X$.tag;
        $B$.VT = $Z$.tag;
        $B$.VC = 0;
    end if
end if
```

entry contains 7 fields: A valid bit indicates whether the entry is valid, and an entry is invalidated when its VC exceeds the threshold; replacement bits are used to select an RHT entry to record the current incoming block and victim block when there is no invalid entry, and the replacement policy of RHT is similar to LRU; a prefetch bit indicates whether the request is a prefetch request, and we will introduce how to use it later; PC bits keep the PC of incoming block; VC stores the value difference; IT and VT store the tags of incoming and victim blocks respectively, and we use partial tags to reduce the storage overhead, so that IT and VT only store the lower 18 bits of tags instead of the whole tags.

It is not necessary to record the incoming and victim block pairs of all misses in the RHT. We record the miss in the RHT only if there are invalid entries in the corresponding set of RHT or a small probability is satisfied. Our experiments show that for the 128-entry RHT and 2MB L2 cache, VIP performs well enough when the probability is 1/256. A small RHT can also reduce the number of HBC entries. We only need to compute hit benefits for hit requests that hit in the RHT. Our experiments show that a 4-entry HBC is enough. When there is no available HBC entry, the extra request for hit benefit computation is simply dropped.

Besides PC, other signatures of cache requests can also be used to index the RVPT, such as block address. Previous studies have shown that using PC to classify cache requests is more effective compared to other methods [46], and thus we use the instruction PC which causes the miss to classify incoming blocks in this paper. Our experiments show that a 256-entry RVPT with 3-bit counters can achieve enough performance gain. Therefore, the RHT entry only needs to keep the 9th to 2nd bits of PC, since the benchmarks are compiled to Alpha binaries, and the lower 2 bits of PC in Alpha instructions are always 0. The shortened 8-bit PC is delivered along with the cache request in the cache hierarchy like all prior cache policies using PC [25, 29, 30, 46].

## 4.3 Thread-awareness

In multi-core environment where the last-level cache is shared, the values of blocks from different cores can show significant disparity. Therefore, it is necessary for VIP to make insertion decisions for each individual core. Due to the PC based design of VIP, it can distinguish cache blocks from different cores. The only extension that VIP needs for multi-core environment is to assign a $C_{\mathrm{demand\_miss}}$ for each core to enable its miss penalty and hit benefit computation.

## 4.4 Prefetch-awareness

Prefetching is widely used in modern processors to enhance cache performance. However, incorrect prefetch requests cannot provide any performance improvement, and they can hurt performance by prematurely evicting valuable blocks from the cache. Therefore, it is necessary for VIP to make insertion decisions for demand and prefetch requests separately. To be prefetch-aware, each RHT entry needs a prefetch bit to indicate whether it is a perfetch request, and the PC bits are used to store the core number instead of PC when recording a prefetch request in the RHT entry. We also assign an additional saturating counter in the RVPT for each core which is dedicated to prefetch requests. Its value indicates whether the EB of the block prefetched by a specific core should be set to 1.

## 4.5 Design Issues for Inclusive Caches

In this paper, we evaluate the performance of VIP in an inclusive L2 cache. In an inclusive cache hierarchy, when a cache block is evicted from the L2 cache, the same block must be evicted by L1 caches to satisfy the inclusion property, and these evicted L1 cache blocks as a result of inclusion are called *inclusion victims*. Inclusion victims may show good locality in L1 caches, and thus evicting them early can hurt the performance [12].

To eliminate inclusion victims, the cache management policy should avoid replacing blocks resident in L1 caches. To that end, the L1 cache sends an *explicit eviction notification* to the L2 cache on the replacement of a clean block, so that the L2 cache can have the accurate information about which cores are caching the block [35]. Upon replacement, the L2 cache preferentially selects the victim among blocks which are not resident in L1 caches. If there are no such blocks, the replacement policy will select the victim block from all blocks in the cache set. This method is similar to a recently proposed inclusive cache management policy [1]. To make a fair comparison, we evaluate the performance of all techniques using this extension in our experiments. Our experiments show that the performance of all techniques can improve slightly with this extension, and the increasing traffic due to explicit eviction notifications is very small.

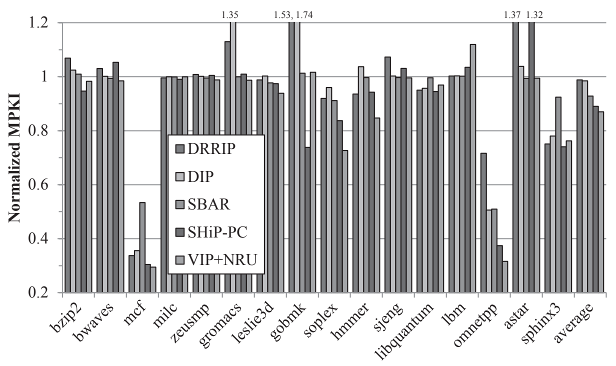


Figure 4: Normalized MPKI for various techniques.

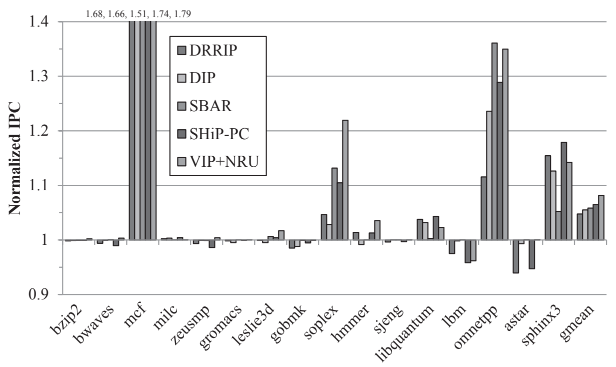


Figure 5: Normalized speedup for various techniques.

## 4.6 Bypass Instead of Insertion in Non-inclusive and Exclusive Caches

In non-inclusive and exclusive cache hierarchies, last-level caches do not have to keep the same blocks in L1 caches. In such scenarios, valueless blocks can be bypassed directly. Therefore, VIP converts to *Value based Bypass Policy (VBP)*. VBP does not need to append each block with an eviction bit, and thus it consumes less storage overhead. This paper focuses on the study of VIP, and we leave the study of VBP as part of our future work.

## 5. EXPERIMENTAL METHODOLOGY

The simulator we used is gem5 [2]. The microarchitecture parameters of the simulator are shown in Table 1, and the configuration of the processor is similar to that of the Intel Nehalem [11]. The simulator models a two-level inclusive cache hierarchy using the MESI coherence protocol. The L1 caches are private to each core, while the L2 cache is shared by all cores. In single-core configuration, the L2 cache is 16-way 2MB. In multi-core configuration, the L2 cache is 4MB for 4 cores and 8MB for 8 cores. The simulator also models a hardware stream prefetcher for each core, and prefetched blocks are inserted into both the L1 and L2 caches.

Table 1: Parameters of the simulator.

| Parameter | Configuration |
|---|---|
| Processor | 4-wide, 128-entry ROB, 48-entry Load Queue, 32-entry Store Queue |
| L1 ICache | 32KB, 64B block size, 4-way, 3-cycle hit latency, PLRU |
| L1 DCache | 32KB, 64B block size, 4-way, 3-cycle hit latency, PLRU, 32-entry MSHR |
| L2 Cache | 2MB/4MB/8MB, 64B block size, 16-way, 16-cycle hit latency, 64-entry MSHR |
| Memory | 1 channel, 2 dimms, 2 ranks per dimm, 8 banks per rank, bank conflicts modeled, 150-cycle minimal access latency |

We use SPEC CPU2006 benchmarks [9] with the first reference inputs to do evaluation. SimPoint [37] is used to obtain a single representative 200 million instructions for each benchmark. Among the benchmarks that can be addressed by our simulation infrastructure, `gamess`, `namd`, `povray`, `calculix`, `h264ref`, and `wrf` are not evaluated because their working sets are very small and their misses per thousand instructions (MPKI) are less than 0.1 in a 2MB L2 cache under LRU. The rest of 16 benchmarks are used in our experiments.

For the performance evaluation in multi-core environment, we choose several benchmarks out of 16 selected SPEC CPU2006 benchmarks at random to combine into a multi-core workload. Totally, we create 20 4-core workloads and 8 8-core workloads. We quantify the performance in multi-core environment with the following three widely used metrics: weighted speedup $= \sum_{i=1}^{n} \frac{IPC_i}{SingleIPC_i}$, throughput $= \sum_{i=1}^{n} IPC_i$, and fair speedup $= n / \sum_{i=1}^{n} \frac{SingleIPC_i}{IPC_i}$, where $n$ is the number of cores, and $SingleIPC_i$ is got when program $i$ runs alone.

## 6. RESULTS AND ANALYSIS

## 6.1 Performance on Single-core Workloads

At first, we evaluate the performance of VIP when NRU is used as the baseline replacement policy and prefetching is disabled. Besides VIP with NRU (VIP+NRU), we also investigate the performance of other state-of-the-art techniques, including miss count based DIP [38], DRRIP [14], and SHiP-PC [46], and miss penalty based SBAR [39].

Figure 4 and Figure 5 show MPKI and IPC both normalized to LRU for various techniques respectively. Compared to LRU, VIP+NRU reduces MPKI by 13.0% on average and achieves a geometric mean speedup of 8.2%, while the geometric mean speedup is 4.8% for DRRIP, 5.5% for DIP, 5.8% for SBAR, and 6.5% for SHiP-PC.

DIP, DRRIP, and SHiP-PC only consider the recency information of a block as its value, and thus VIP+NRU outperforms them since they are not aware of the variation of miss penalty. For instance, we observe that the MPKI of SHiP-PC for `bzip2` is lower than that of VIP+NRU, but since these reduced misses are at the expense of more costly ones, the IPC of VIP+NRU is slightly higher than that of SHiP-PC for `bzip2`. For `libquantum` and `sphinx3`, SHiP-PC reduces more misses and outperforms VIP+NRU because it uses a larger predictor. SBAR takes into account both miss penalty and recency information, but as we stated, it is also important to take into account hit benefit. VIP+NRU thus outperforms SBAR consistently except `omnetpp`, for which SBAR has slightly better performance since it uses a more complex miss penalty computation method as stated in Section 3.2. When VIP+NRU uses the same computation method as SBAR, they have similar performance for `omnetpp`. For `lbm`, a small number of blocks occasionally show huge values, and their behaviors are very difficult to

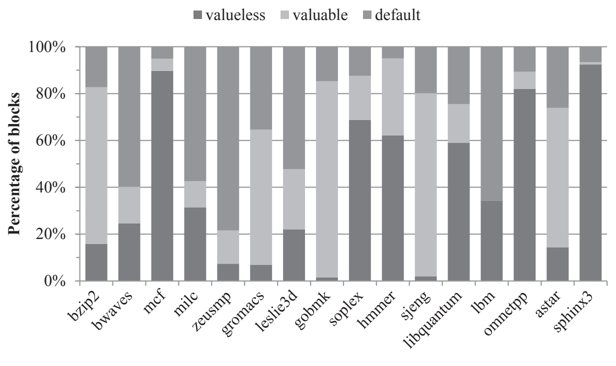


Figure 6: Prediction results of VIP+NRU.

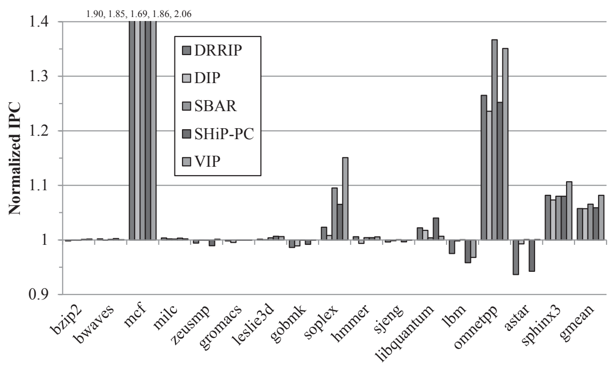
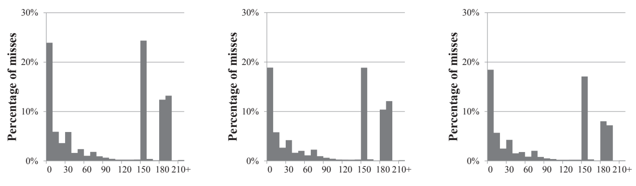


Figure 7: Normalized speedup in the presence of prefetching.

predict. Thus, only LRU and techniques that can dynamically switch to LRU (DIP and SBAR) perform well for `lbm`.

Figure 6 shows the prediction results of VIP+NRU. For programs such as `bwaves` and `lbm`, neither the incoming block nor the victim block is valuable in most cases. In such scenarios, VIP+NRU cannot learn their value relationship (the RVPT counter is 0) and does not set the eviction bit by default. For programs where majority of blocks have low values such as `mcf` and `sphinx3` in Figure 2, most of incoming blocks are predicted to be valueless and VIP+NRU performs well. While for programs which have higher block values such as `astar`, VIP+NRU correctly predicts most blocks to be valuable and thus does not hurt their performance.

To illustrate how VIP improves performance, Figure 8 compares the miss penalty distribution under LRU, SHiP-PC, and VIP+NRU for `soplex`. The miss numbers of SHiP-PC and VIP+NRU are both normalized to the total miss number of LRU. Compared to LRU, both SHiP-PC and VIP+NRU can reduce the aggregate miss count. While the miss counts with small penalty of SHiP-PC and VIP+NRU are almost the same, VIP+NRU reduces more misses with higher penalty ($\geq$ 150 cycles) by retaining more valuable blocks compared to SHiP-PC.



(a) LRU. (b) SHiP-PC. (c) VIP+NRU.

Figure 8: Miss penalty distribution for soplex.

Our evaluation also shows that considering hit benefits as parts of block values is essential. When only miss penalty is considered, the average performance improvement of VIP+NRU is merely 0.1% compared to LRU. Besides, the performance gain of VIP+NRU decreases to 7.3% when the hit benefit is fixed to a static number (150 cycles), which demonstrates that an accurate hit benefit computation method is necessary.

We also study the performance of VIP when the MLP cost computation method proposed by Qureshi *et al.* [39] is used to compute miss penalty and hit benefit. VIP+NRU using the MLP cost computation achieves a geometric mean speedup of 8.4%, which is similar to that of our original method. Since the MLP cost computation method requires complex calculation, our method is easier to implement.

Besides NRU, we also investigate the performance of VIP when LRU or SRRIP [14] is used as the baseline replacement policy. VIP+LRU and VIP+SRRIP outperform LRU by 8.3% and 8.2% on geometric mean respectively. Compared to SRRIP, VIP+SRRIP outperforms it by 5.8%. These results show that VIP can cooperate with various replacement policies. Since NRU needs the lowest storage overhead and is the simplest among these policies, and the performance of VIP+NRU is similar to that of VIP+LRU and VIP+SRRIP, we focus on the study of VIP+NRU. In the rest of this paper, VIP+NRU is referred to as VIP.

Next, we study the performance of VIP in the presence of prefetching. Figure 7 shows the speedup of various techniques when prefetching is enabled. The speedup is normalized to that of LRU in the presence of prefetching. VIP reduces average MPKI by 11.5% and achieves a performance gain of 8.2%, and it also outperforms other techniques. Compared to LRU without prefetching, VIP outperforms it by 23.9%, while LRU with prefetching outperforms it by 14.5%. These experiments show that VIP can also improve cache performance in the presence of prefetching.

## 6.2 Performance on Multi-core Workloads

For multi-core workloads, we compare the performance of VIP with other state-of-the-art shared cache management policies including TADIP [13], TADRRIP, UCP [40], PIPP [48], and SHiP-PC. TADIP and TADRRIP are the thread-aware versions of DIP and DRRIP respectively, and SHiP-PC itself is already thread-aware.

Figure 9 presents the s-curve of the weighted speedup normalized to LRU for 20 4-core workloads in the absence of prefetching. The s-curve is plotted by sorting the data from the lowest to the highest. Compared to LRU, VIP achieves a geometric mean weighted speedup improvement of 5.9%, while it is 2.0% for PIPP, 3.1% for UCP, 4.0% for TADIP, 4.3% for TADRRIP, and 4.2% for SHiP-PC. While other techniques incur performance loss for some workloads, VIP does not degrade the performance for any workload. Due to the variation of block values across different cores, the performance benefits of VIP for multi-core workloads are more significant than those for single-core workloads, especially for multi-core workloads where programs with high values and those with low values execute in parallel. On average, VIP predicts that 69.7% of incoming blocks are valueless for 20 4-core workloads.

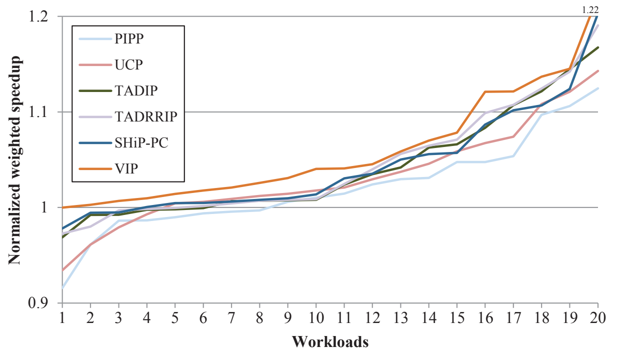


Figure 9: S-curve for normalized weighted speedup on 4-core workloads in the absence of prefetching.

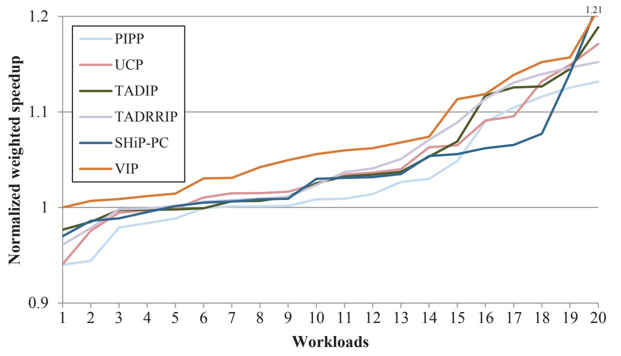


Figure 10: S-curve for normalized weighted speedup on 4-core workloads in the presence of prefetching.

We also study the results on throughput and fair speedup respectively. The throughput improvement of VIP is 5.7% on geometric mean, and the fair speedup improvement of VIP is 8.3%. VIP also outperforms other state-of-the-art techniques on the metrics of throughput and fair speedup. The performance improvement of VIP on fair speedup is the most, since VIP rarely causes performance degradation for any program in a multi-core workload.

Figure 10 shows the s-curve of the normalized weighted speedup on 4-core workloads when prefetching is enabled. Compared to LRU with prefetching, the weighted speedup improvement is 6.9% for VIP, while it is 4.7% for TADRRIP which performs best among the rest of techniques.

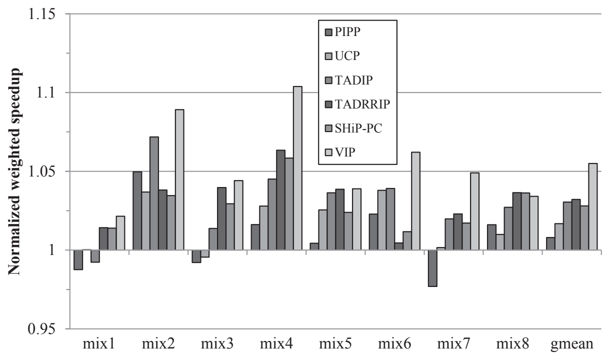


Figure 11: Normalized weighted speedup for 8-core workloads in the absence of prefetching.

To study the scalability of VIP with the number of cores, we evaluate the performance of VIP on 8-core workloads. Figure 11 presents the normalized weighted speedup for 8-core workloads in the absence of prefetching. Compared to LRU, the weighted speedup improvement of VIP is 5.5%, which is significantly better than the improvement of other techniques in which TADRRIP performs best and outperforms LRU by 3.2%.

## 6.3 Storage Overhead

Table 2 compares the storage overhead of various techniques for the 4MB L2 cache used in 4-core configuration. The storage overhead of VIP+NRU comes from per-block eviction bit and NRU bit, RHT, RVPT, and the structures for miss penalty and hit benefit computation. Each cache block needs 1 eviction bit and 1 NRU bit. For a 128-entry 8-way RHT, each RHT entry consists of 1 valid bit, 3 replacement bits, 1 prefetch bit, 8-bit PC, 18-bit IT, 18-bit VT, and 9-bit VC. Each RVPT counter needs 3 bits. For miss penalty and hit benefit computation, each core requires a 7-bit $C_{\text{demand\_miss}}$, and each MSHR entry requires an 8-bit $T_{\text{p}}$. Each HBC entry requires a valid bit, an 8-bit $T_{\text{b}}$, an 8-bit latency counter, and a 36-bit address. It totally consumes 17.09KB of extra storage to implement VIP, which is roughly 0.4% of the total storage of a 4MB L2 cache. Compared to other recent proposals, the storage overhead of VIP+NRU is close to that of TADRRIP and lower compared to that of others. The storage overhead of VIP+NRU in single-core configuration is also very low.

Table 2: Storage overhead of various techniques.

| | Storage per block | Extra storage | Total |
|---|---|---|---|
| LRU | 4 bits | 0 | 32KB |
| PIPP | 4 bits | 5KB | 37KB |
| UCP | 6 bits | 5KB | 53KB |
| TADIP | 4 bits | 40 bits | 32KB |
| TADRRIP | 2 bits | 40 bits | 16KB |
| SHiP-PC | 2 bits | 24.5KB | 40.5KB |
| VIP+NRU | 2 bits | 1.09KB | 17.09KB |

## 7. CONCLUSION

Every cache management policy tries to improve cache performance by retaining blocks with higher *values* longer. Traditionally, the access time or reuse distance of a cache block is treated as its value. However, cache miss latency is variable in modern systems. As a result, the cache management policy should also be able to take into account the performance impact of each cache request when it results in miss. This paper achieves this goal by making the following contributions:

1. We illustrate that the *value* of a cache block depends on all its requests. It includes not only the performance loss on its misses (i.e., *miss penalty*), but also the performance gain due to its hits (i.e., *hit benefit*).

2. We propose a hardware mechanism to compute miss penalty and hit benefit, which only requires negligible modification to the current cache design. Then the value of a block can be calculated by accumulating all its miss penalty and hit benefits.

3. We propose *Value based Insertion Policy (VIP)*. VIP learns and predicts the relationship between the value

of the incoming block and that of the victim block on a cache miss. If the value of the incoming block is predicted to be lower, it is assigned with a high eviction priority so that it can be evicted from the cache quickly.

Our evaluation shows that VIP can improve cache performance for both single-core and multi-core workloads. In future systems with heterogeneous multiprocessors and hybrid memory hierarchies, the values of different cache blocks can even change more dramatically. To apply VIP in such systems is part of our future work.

## 8. ACKNOWLEDGMENTS

This work is supported by the National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2009ZX01029-001-002-2. We would like to acknowledge Dong Tong, Zichao Xie, and the anonymous reviewers for their helpful comments.

## 9. REFERENCES

[1] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería. Exploiting reuse locality on inclusive shared last-level caches. *ACM Trans. Archit. Code Optim.*, 9(4):38:1–38:19, 2013.
[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, 2011.
[3] M. Chaudhuri. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *MICRO-42*, 2009.
[4] C.-H. Chi and H. Dietz. Improving cache performance by selective cache bypass. In *HICSS-22*, 1989.
[5] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *MICRO-45*, 2012.
[6] H. Gao and C. Wilkerson. A dueling segmented LRU replacement algorithm with adaptive bypassing. In *JWAC-1*, 2010.
[7] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *ISCA-38*, 2011.
[8] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS-9*, 1995.
[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, 2006.
[10] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *ISCA-29*, 2002.
[11] Intel. Intel Core i7 processor. http://www.intel.com/products/processor/corei7/.
[12] A. Jaleel, E. Borch, M. Bhandaru, S. Steely, and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *MICRO-43*, 2010.
[13] 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.
[14] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA-37*, 2010.
[15] J. Jalminger and P. Stenstrom. A novel approach to cache block reuse predictions. In *ICPP '03*, 2003.
[16] J. Jeong and M. Dubois. Optimal replacements in caches with two miss costs. In *SPAA-17*, 1999.
[17] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. In *HPCA-9*, 2003.
[18] J. Jeong and M. Dubois. Cache replacement algorithms with nonuniform miss costs. *Computers, IEEE Transactions on*, 55(4):353–365, 2006.
[19] J. Jeong, P. Stenström, and M. Dubois. Simple penalty-sensitive replacement policies for caches. In *CF-3*, 2006.
[20] D. A. Jiménez. Insertion and promotion for tree-based PseudoLRU last-level caches. In *MICRO-46*, 2013.
[21] T. Johnson, D. Connors, M. Merten, and W.-M. Hwu. Run-time cache bypassing. *Computers, IEEE Transactions on*, 48(12):1338 –1354, 1999.
[22] R. D.-c. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *ISCA-28*, 2001.
[23] D. Kaseridis, M. Iqbal, and L. John. Cache friendliness-aware management of shared last-level caches for high performance multi-core systems. *Computers, IEEE Transactions on*, 2013.
[24] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD-25*, 2007.
[25] S. M. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *MICRO-43*, 2010.
[26] M. Kharbutli and R. Sheikh. LACS: A locality-aware cost-sensitive cache replacement algorithm. *Computers, IEEE Transactions on*, 2013.
[27] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *Computers, IEEE Transactions on*, 57(4):433 –447, 2008.
[28] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.
[29] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *ISCA-28*, 2001.
[30] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng. Optimal bypass monitor for high performance last-level caches. In *PACT-21*, 2012.
[31] 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.
[32] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *MICRO-44*, 2011.
[33] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An efficient multicore cache organization based on next-use distance. In *HPCA-17*, 2011.
[34] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (PriSM). In *ISCA-39*, 2012.
[35] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.
[36] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero. MLP-aware dynamic cache partitioning. In *HiPEAC '08*. 2008.
[37] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31:318–319, 2003.
[38] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007.
[39] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA-33*, 2006.
[40] 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.
[41] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ICS-12*, 1998.
[42] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *ISCA-38*, 2011.
[43] R. Sheikh and M. Kharbutli. Improving cache performance by combining cost-sensitivity and locality principles in cache replacement algorithms. In *ICCD-28*, 2010.
[44] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO-28*, 1995.
[45] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.
[46] C. 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.
[47] Y. Xie. Modeling, architecture, and applications for emerging memory technologies. *Design Test of Computers, IEEE*, 28(1):44–51, 2011.
[48] Y. Xie and G. H. Loh. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA-36*, 2009.