Improving Cache Performance by Exploiting Read-Write Disparity

Samira Khan[†*]
samirakhan@cmu.edu

Alaa R. Alameldeen[*]
alaa.r.alameldeen@intel.com

Chris Wilkerson[*]
chris.wilkerson@intel.com

Onur Mutlu[†]
onur@cmu.edu

Daniel A. Jiménez[‡]
djimenez@cs.tamu.edu

[†]Carnegie Mellon University   [*]Intel Labs   [‡]Texas A&M University

Abstract

Cache read misses stall the processor if there are no independent instructions to execute. In contrast, most cache write misses are off the critical path of execution, since writes can be buffered in the cache or the store buffer. With few exceptions, cache lines that serve loads are more critical for performance than cache lines that serve only stores. Unfortunately, traditional cache management mechanisms do not take into account this disparity between read-write criticality. The key contribution of this paper is the new idea of distinguishing between lines that are reused by reads versus those that are reused only by writes to focus cache management policies on the more critical read lines. We propose a Read-Write Partitioning (RWP) policy that minimizes read misses by dynamically partitioning the cache into clean and dirty partitions, where partitions grow in size if they are more likely to receive future read requests. We show that exploiting the differences in read-write criticality provides better performance over prior cache management mechanisms. For a single-core system, RWP provides 5% average speedup across the entire SPEC CPU2006 suite, and 14% average speedup for cache-sensitive benchmarks, over the baseline LRU replacement policy. We also show that RWP can perform within 3% of a new yet complex instruction-address-based technique, Read Reference Predictor (RRP), that bypasses cache lines which are unlikely to receive any read requests, while requiring only 5.4% of RRP's state overhead. On a 4-core system, our RWP mechanism improves system throughput by 6% over the baseline and outperforms three other state-of-the-art mechanisms we evaluate.

1. Introduction

The performance gap between the processor and memory is a major bottleneck in microprocessor design. In today's high-performance systems, a memory request may take hundreds of cycles to complete. This performance gap motivates continued improvements in cache efficiency. Some prior works focused on improving cache efficiency by better identifying and exploiting the natural locality of the working set, relying on improved replacement/insertion algorithms to identify portions of the working set most likely to be reused [25, 9, 21, 11, 31]. Other works attempted to differentiate between "critical" and "non-critical" parts of the working set [32, 7, 28]: using memory-level-parallelism (MLP) or the likelihood of a processor stall as the main criteria, cache efficiency could be improved by prioritizing cache resources based on the criticality of the data.

The difference in criticality between loads and stores in the core is well-known. In the processor pipeline, loads and stores are treated very differently, with great attention paid to the latency of loads, while stores may be buffered for some time before finally committing to the cache or memory. Although the treatment of loads and stores in the processor pipeline is dramatically different, their associated read and write requests are not generally distinguished in the cache hierarchy. However, read and write requests have different characteristics. The latency of read requests is often more critical than the latency of write requests. In light of this fact, we argue that caches should be designed to favor critical read requests over less critical write requests.

In this paper, we propose cache management techniques that increase the probability of cache hits for critical read requests, potentially at the cost of causing less critical write requests to miss. To accomplish this, we must distinguish between cache lines that will be read in the future and those that will not. Prior works sought to distinguish between cache lines that are reused and those that are not reused, with the intent of filtering lines that were unlikely to be reused. However, prior work did not distinguish between reuse due to critical read requests and less critical write requests, hence potentially leading to additional read misses on the critical path. The key contribution of this paper is the new idea of distinguishing between lines that are reused by reads versus those that are reused *only by writes* to focus cache management policies on the more critical read lines. To our knowledge, this is the first work that uses whether or not a line is written to as an indicator of future criticality of the cache line.

We also make the following contributions:

- We present data highlighting the criticality and locality differences between reads and writes within the cache (i.e.,
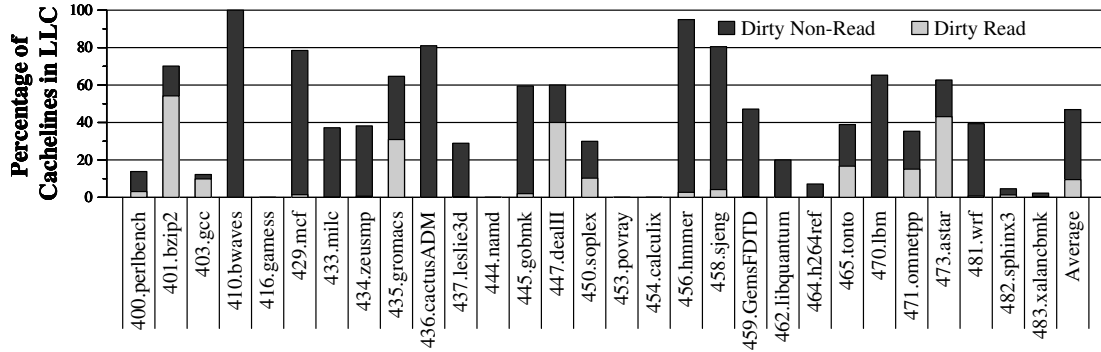
1

Figure 1: Percentage of dirty cachelines in LLC that are read vs. not read.

cache lines that are read versus cache lines that are only written).

- To exploit the disparity in read-write criticality, we propose Read Write Partitioning (RWP), a mechanism that divides the last-level cache into two logical partitions for clean and dirty lines. RWP predicts the best partition sizes to increase the likelihood of future read hits. This could lead to allocating more lines in the clean partition or the dirty partition, depending on which partition serves more read requests.

- To show the potential for favoring lines that serve reads, we discuss a complex PC-based predictor, Read Reference Predictor (RRP). RRP uses the PC of the first reference to identify cache lines likely to service future reads, and therefore avoids allocating write-only lines and lines that are unlikely to get reused by reads.

- We show that our RWP mechanism is close in performance to the more complex RRP with only 5.4% of RRP's state overhead. We also show that RWP outperforms prior state-of-the-art cache management policies.

Our experimental evaluation shows that our cache management mechanisms, which take into account read-write criticality, provide better performance by improving cache read hit rate. RRP provides an average 17.6% speedup while RWP provides an average 14.6% speedup for memory-intensive SPEC CPU2006 benchmarks over the baseline LRU replacement policy. Over the entire SPEC CPU2006 suite, RRP/RWP provide an average $6.2\%/5.1\%$ speedup. On a 4-core system, RRP/RWP improve system throughput by $8.4\%/6.2\%$ for a diverse set of multi-programmed workloads. In addition, we show that exploiting read-write criticality provides better performance than three previously proposed cache management mechanisms [28, 6, 25].

The organization of this paper is follows. Section 2 discusses the differences between reads and writes and articulates the reasons for distinguishing between them. Sections 3 and 4 highlight two different approaches (RWP, RRP) to exploit these differences. Sections 5 and 6 present our experimental methodology and results. Section 7 qualitatively compares our work to previous work, and Section 8 provides our conclusions.

2. Read Lines vs. Write-only Lines

Ideally, a cache that favors read requests over write requests could be designed by sorting cache lines into one of two categories. The first category would consist of lines that service read requests (read lines). The second category would consist of lines that only serve to buffer writes (write-only lines) and that will not be read. Leveraging this classification, the cache replacement algorithm would favor read lines over write-only lines when making replacement decisions. In practice, however, classifying cache lines into these two categories is challenging since it requires future knowledge. One approach might be to predict future access types based on past access types, e.g., predict that dirty lines will be written to and clean lines will be read. A replacement algorithm would exploit this by simply prioritizing clean lines over dirty lines. Unfortunately, dirty lines are often requested by reads, and a replacement algorithm that classifies dirty lines as non-critical and speeds up their eviction would degrade performance as the read requests to these lines that are evicted early would become cache misses. To avoid this problem, a cache management technique that exploits the criticality gap between reads and writes would distinguish between dirty lines and write-only lines. Dirty lines include all lines that have been written to, regardless of whether they are requested (or not) by future reads. Write-only lines are a subset of dirty lines that we anticipate not to be requested by future reads. To maximize the number of critical read requests that result in cache hits, we attempt to minimize the number of write-only lines in the cache and maximize the number of read lines (both clean and dirty). To the best of our knowledge, this paper describes the first cache management policy that draws this distinction and exploits it to improve cache performance.

We studied the differences between dirty lines and write-only lines in detail. Figure 1 shows the percentage of total cache lines that are dirty. On average, almost 46% of lines in a 4 MB last-level cache (LLC) are dirty lines across all SPEC CPU2006 benchmarks (the remaining 54% are clean lines). The dirty cache lines are divided into two sub-categories. The first sub-category consists of dirty cache lines that are written to with no subsequent reads before eviction (i.e., write-only lines), and is depicted by the darker bar at the top. On average, nearly 37% of all cache lines fit in this category. The second

sub-category, denoted by the gray bar at the bottom, depicts the subset of dirty lines that are read at least once in addition to being written to, constituting nearly 9% of all cache lines on average. In the next two subsections, we discuss both categories of dirty lines.

2.1. Write-Only Lines

Workloads with write-only lines fall into two categories: read-intensive, and write-intensive. 483.xalancbmk, and 482.sphinx, for example, are read-intensive benchmarks, while 481.wrf and 456.hmmer are write-intensive benchmarks.

Read-Intensive In Figure 1, we show that the majority of cache lines in the LLC are clean for 483.xalancbmk and 482.sphinx. In 483.xalancbmk, the majority of dirty lines come from the stack. In fact, 99% of all dynamic writes come from passing parameters through the stack, primarily in the function *elementAT*.[1] The vast majority of dirty lines are also write-only lines for these benchmarks.

Write-Intensive These workloads produce a large number of writes, and are distinguished by a large percentage of dirty lines in the LLC. 481.wrf and 456.hmmer are both good examples of this type of workloads. 481.wrf uses the Runge-Kutta numerical method, initializing a large number of 3D arrays to zero at each step of the numerical analysis. Since the arrays are very large, the lines are evicted and written back to memory without being read at all. In 481.wrf, very few read requests are serviced by dirty lines, and this appears to be true at every level in the cache hierarchy. 456.hmmer is a write-intensive workload that exhibits more complex behavior. Writes are often requested by subsequent reads while still residing in the L1 cache. After eviction from the L1, however, dirty cache lines are rarely reused. In the function P7viterbi in 456.hmmer, results (tentative pattern matches) are stored in 2D tables, written back and not reused. For the portions of 456.hmmer that we evaluated, we found that 90% of the lines in the LLC are write-only lines that will not be requested by future reads.

2.2. Dirty-Read Lines

A write no-allocate policy can address the problem of write-only lines by not allocating write lines in the cache. However, not all dirty lines are write-only lines. On average across all SPEC CPU2006 workloads, about 20% of all dirty lines (and 9% of all cache lines) are dirty-read lines. Some workloads, such as 401.bzip, 435.gromacs, 447.dealII, 471.omnetpp, 473.astar, and 450.soplex frequently read dirty data. 450.soplex, in particular, often writes intermediate data to be reused in the immediate future. 450.soplex uses the simplex algorithm to solve linear programs, essentially performing a series of transformations on a sparse matrix that largely resides in the LLC. Most reads and writes are produced as loop iterations examine and manipulate the matrix. Lines that are

written to in any particular iteration will be read in the next iteration as the next iteration re-examines the matrix. In fact, reads are more likely to request data in dirty lines than in clean lines. As a result, 450.soplex actually benefits from favoring those lines that have been written to over those that have not.

As we demonstrate in this section, different workloads exhibit different types of behavior and different mixes of write-only, dirty-read, and clean lines with different criticality. Simple mechanisms (such as bypassing one type of lines) would not lead to good performance across all workloads. We demonstrate the downside of such simple mechanisms in the motivating example we present in the next subsection. Therefore, maximizing performance across a range of workloads requires a more sophisticated approach capable of identifying write-only lines, or favoring one type of access over another based on the likelihood of future reads.

2.3. Motivating Example

Figure 2 describes an example to illustrate the benefit of using read-write information in the cache allocation and replacement policy. Figure 2(a) depicts a loop with a burst of memory references occurring at four different points in the execution of the loop. Memory operations in the loop reference a total of 6 cache lines, denoted as A, B, C, D, E, and F. Cache lines A, D, E, and F are only read, B is read and written to, and C is only written to. Figure 2(b) shows the state of an LRU replacement algorithm for a fully-associative cache with a total of 4 lines, after each of the 4 bursts of memory references. During each iteration of the loop, cache lines A-F are all referenced, however, the latest reference to D (in the third burst), is followed by 4 unique references to A, C, E, and F, causing D to be evicted at the end of each iteration. As a result, the processor stalls when a load re-references D during the first burst of the second iteration. In addition, the intermittent accesses to D and B periodically evict E and F, introducing two more stalls in each iteration. The problem is that the cache is capable of storing 4 lines but the loop has a total working set of 6 lines. As a result, LRU policy leads to thrashing as all 6 working set lines compete for available cache space, leading to three read misses (i.e., processor stalls) per iteration.

Our proposal to address this thrashing is to exclude the write-only-line (C) from allocation, and focus cache resources only on the 5 critical read-lines, A, B, D, E, and F. With this approach, we only incur one read miss (i.e., processor stall) in each loop iteration. Figure 2(c) depicts the same loop, this time in the context of a read-biased LRU replacement algorithm (i.e., favoring read-lines over write-only-lines). When the write-only line C is referenced, the replacement algorithm chooses not to allocate the line, leaving the handling of it to the store queue. Excluding the write-only line from allocation reduces the effective size of the working set (from 6 to 5 lines in this example), which increases read hit rate in the cache, leading to only one stall per iteration.

While a simple write no-allocate policy would avoid some of the problems in this example, it misses the impact of writes
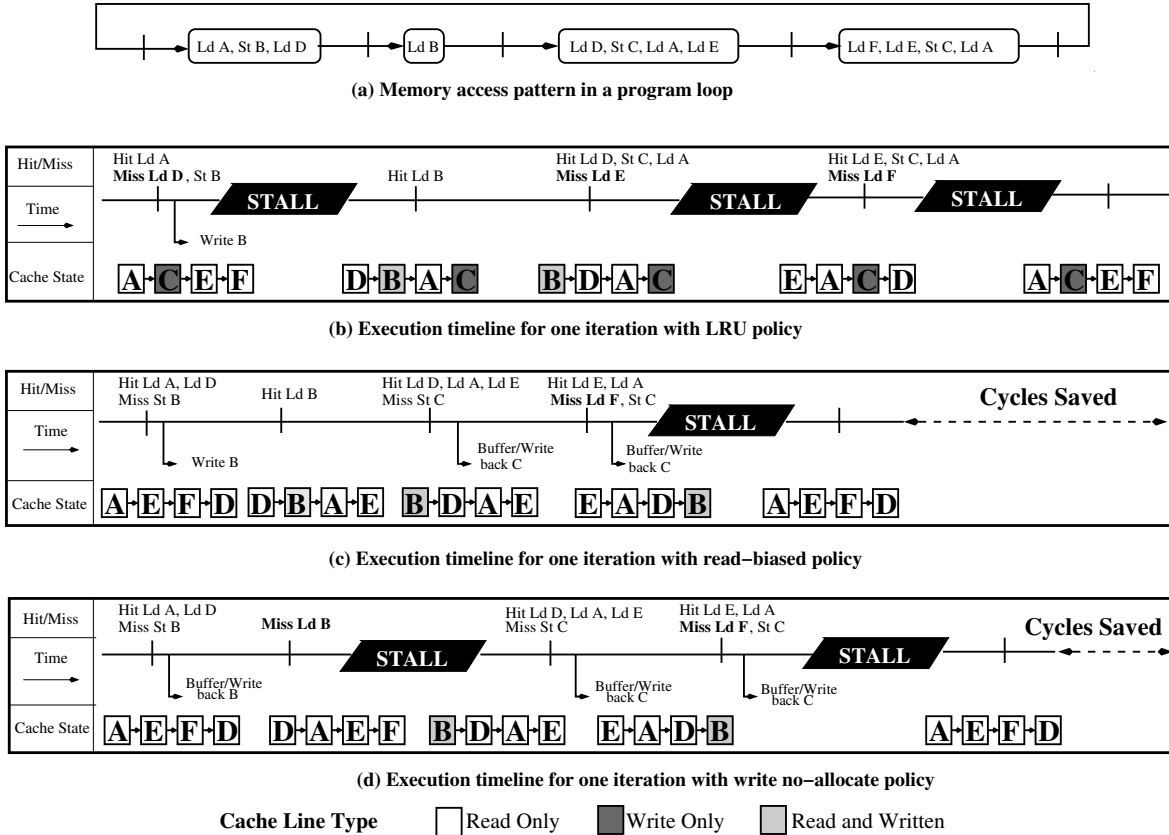
---

[1]While the stack experiences both reads and writes, this benchmark had most of the reads filtered by the higher level caches, so the last level cache got mostly write-only lines.

(a) Memory access pattern in a program loop

| Ld A, St B, Ld D | Ld B | Ld D, St C, Ld A, Ld E | Ld F, Ld E, St C, Ld A |

Hit/Miss: Hit Ld A / **Miss Ld D**, St B — Hit Ld B — Hit Ld D, St C, Ld A / **Miss Ld E** — Hit Ld E, St C, Ld A / **Miss Ld F**

Time → STALL ... STALL ... STALL

Write B

Cache State: A·C·E·F D·B·A·C B·D·A·C E·A·C·D A·C·E·F

(b) Execution timeline for one iteration with LRU policy

Hit/Miss: Hit Ld A, Ld D / Miss St B — Hit Ld B — Hit Ld D, Ld A, Ld E / Miss St C — Hit Ld E, Ld A / **Miss Ld F**, St C — **Cycles Saved**

Time → STALL

Write B — Buffer/Write back C — Buffer/Write back C

Cache State: A·E·F·D D·B·A·E B·D·A·E E·A·D·B A·E·F·D

(c) Execution timeline for one iteration with read−biased policy

Hit/Miss: Hit Ld A, Ld D / Miss St B — **Miss Ld B** — Hit Ld D, Ld A, Ld E / Miss St C — Hit Ld E, Ld A / **Miss Ld F**, St C — STALL — **Cycles Saved**

Time → STALL ... STALL

Buffer/Write back B — Buffer/Write back C — Buffer/Write back C

Cache State: A·E·F·D D·A·E·F B·D·A·E E·A·D·B A·E·F·D

(d) Execution timeline for one iteration with write no−allocate policy

Cache Line Type: ☐ Read Only ■ Write Only ▨ Read and Written

Figure 2: Motivating Example: Drawback of not taking read-write differences into account in cache management. (a) Memory accesses in a loop; (b) LRU replacement; (c) Read-biased LRU replacement; (d) Write no-allocate policy.

that are followed by reads to the same line. The behavior of a write no-allocate cache is shown in Figure 2(d). Like our read-biased replacement algorithm, the write no-allocate cache provides benefit by excluding the write-only-line C. The two policies differ in their treatment of line B, which is both read and written. B is written to in the first burst and read in the second burst. Our read-biased replacement algorithm, despite the first write to B, allocates the line, resulting in a cache hit for the read request to B. In contrast, the write no-allocate policy does not allocate B in the cache in the first burst. This causes the read request to B in the second burst to consistently miss the cache, leading to two processor stalls per iteration. This is better than the LRU policy (Figure 2(b)), but not as good as the read-biased policy that does not allocate write-only lines (Figure 2(c)).

As this example demonstrates, distinguishing between reads and writes in the cache can improve execution time (Figure 2(c)), but simple approaches (such as the one in Figure 2(d)) could cause undesired side effects that undermine performance or they do not effectively exploit the potential of differentiating reads and writes in the cache. We therefore need more sophisticated mechanisms to reduce write-only lines in the cache. In the next two sections, we describe two mechanisms that exploit the criticality gap between reads and writes by dynamically dividing up the cache between clean and dirty lines (Section 3), or by predicting future reads to a cache line based on the requesting instruction's PC (Section 4).

3. Read-Write Partitioning (RWP)

In this section, we propose a mechanism to exploit the read-write criticality gap to improve cache performance. The goal is to maximize the number of read hits in the cache so that critical read requests can be serviced earlier. To achieve this goal, we need to effectively identify cache lines that are likely to service future reads.

We make the observation that some applications have more read requests to clean lines, whereas other applications have more read requests to dirty lines. In Figure 3(a-b), we show the total number of reads in clean and dirty lines logged at every 100 million instructions, normalized to the number of reads in clean lines at 100 million instructions. Figure 3(a) shows that the benchmark 483.xalancbmk has more reads in clean lines than dirty lines. Figure 3(b) shows that the benchmark 450.soplex receives more read requests to dirty lines.

We exploit this behavior in our mechanism, Read-Write Partitioning (RWP). RWP logically divides the cache into two partitions, a read (i.e., clean) partition and a write (i.e., dirty) partition.[2] We dynamically adapt partition sizes for different

---

[2] We interchangeably use *read partition* and *clean partition*. Ditto for *write partition* and *dirty partition*.
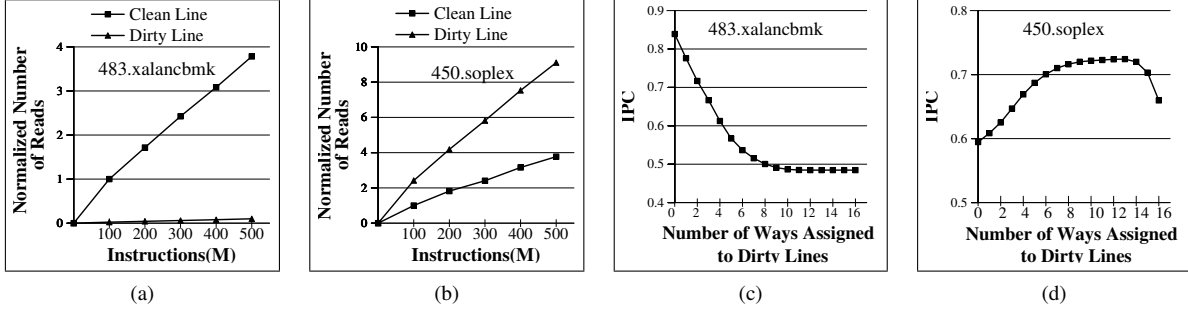
Figure 3: (a, b) Number of reads in clean and dirty lines during the execution; (c, d) IPC when the number of ways allocated to dirty lines is varied in a 16-way 4MB LLC.
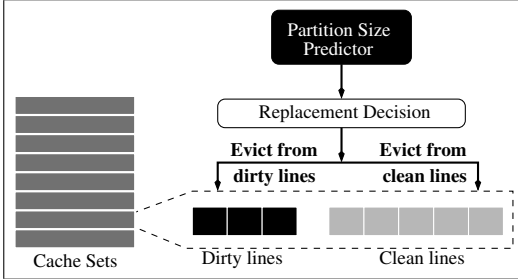

Figure 4: Structure of Read Write Partitioning (RWP).

workloads and phases to maximize read hits and thereby improve performance. RWP is confined to the LLC, and does not need any additional information from the processor, the L1 or L2 caches. This is in contrast to the more complex RRP mechanism we will describe in Section 4.

To motivate our dynamic RWP mechanism, we show that performance varies differently for different benchmarks based on partition sizes. Figure 3(c-d) shows the change in performance for 483.xalancbmk and 450.soplex when the number of ways allocated to dirty lines is varied from 0 to 16 in a 16-way 4MB LLC. 483.xalancbmk has more reads in clean lines. Figure 3(c) shows that it performs best when all cache lines are allocated to the clean partition. Conversely, 450.soplex has more read requests serviced from dirty lines than from clean lines. However, 450.soplex also has a significant number of read requests serviced from the clean lines, as shown in Figure 3(b). Figure 3(d) shows that, while having more dirty lines than clean lines is beneficial for 450.soplex, allocating more than 13 ways to dirty lines would hurt performance, so 13 is the best number of dirty lines per set. Such varying behavior per workload demonstrates the need for a dynamic cache partitioning technique that predicts the best (logical) number of ways that should be allocated to dirty vs. clean lines. We next describe the basic framework to implement RWP and explain the adaptive mechanism to predict best partition size.

3.1. RWP Framework

In this section, we provide an overview of the framework that support read-write partitioning. Each cache set is logically divided into two partitions for clean and dirty lines. The per-line dirty status bit determines if the line belongs to the read (clean) partition or the write (dirty) partition. Cache lines do not move between physical ways. On a write to a clean line,

its dirty bit is set and it is logically considered to be part of the write partition, but no further action is taken even if write partition size exceeds the predicted-best size. We do not strictly enforce partition sizes on regular cache accesses. Instead, partition sizes are adjusted when a new cache line is allocated.

When a new cache line is allocated, our mechanism first determines which partition to evict a cache line from to place the incoming line, and then inserts the new line in the correct partition based on the Dynamic Insertion Policy (DIP) [28]. This insertion policy decides to put incoming lines in the LRU position if it detects that the workload is thrashing the cache. There are three cases that we consider when we choose a replacement victim, based on the current number of dirty lines in a cache set:

- *Current number of dirty lines is greater than the predicted-best dirty partition size.* This means that the set currently has more dirty lines than it should have. In this case, RWP picks the LRU line from the dirty partition as the replacement victim.

- *Current number of dirty lines is smaller than the predicted-best dirty partition size.* This means that the set currently has more clean lines than it should have. In this case, RWP picks the LRU line from the clean partition as the replacement victim.

- *Current number of dirty lines is equal to the predicted-best dirty partition size.* In this case, the replacement victim depends on the cache access type. If the access is a read, RWP picks the replacement victim from the clean partition. Similarly, a write access triggers a replacement from the dirty partition.

RWP only enforces partition sizes when a new line is inserted, and not when a clean line is written to. We considered two strategies for eagerly enforcing the predicted partition sizes on writes to clean lines, but both were worse than our current mechanism. One option was to write back data when a clean line is written to. However, this significantly increases memory bandwidth and power consumption, as we observed that writes to a clean line are very often followed by other writes to the same line. Another option was to eagerly replace a dirty line with the newly modified line, and invalidate the clean version of the line that is being written to. However,
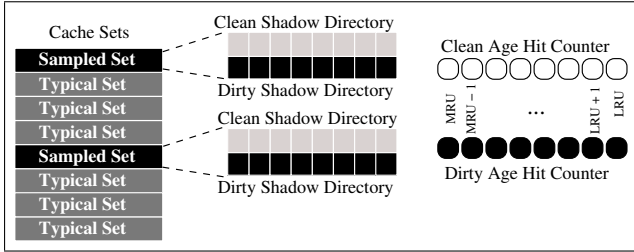
5

Figure 5: Sample sets vs. typical sets. Sample sets model all possible clean vs. dirty partition sizes.

this would hurt performance in cases where the replaced dirty line is re-referenced, since we are wasting the space used by the invalidated line. Our lazy partition enforcement mechanism outperformed both of these options.

3.2. Predicting Partition Sizes

Since clean and dirty lines exhibit different degrees of read reuse for different workloads, RWP adjusts the sizes of the partitions based on workload behavior. As growing one partition necessitates shrinking the other, RWP constantly re-evaluates the benefit of growing one partition against the cost of shrinking the other. To accomplish this, we use a variant of the utility monitors proposed by Qureshi *et al.* [27]. The key difference from [27] is that the monitors in [27] were used to adjust the number of cache ways allocated to different applications in a multi-core system, not to different partitions for clean and dirty lines.

To estimate the partition sizes, RWP compares the read reuse exhibited by the clean and dirty lines as if each were given exclusive access to the entire cache. To do this, RWP employs set sampling [26], extending a small subset (32) of the total sets with shadow directories as depicted in Figure 5. Each sampled set is augmented with two shadow directories: a shadow directory for clean lines (clean shadow directory) and a second dedicated to dirty lines (dirty shadow directory). Each read miss allocates a line to the clean shadow directory; each write miss allocates a line to the dirty shadow directory. A write request hitting in the clean shadow directory causes the clean line to become dirty and move to the dirty shadow directory. The shadow directories are maintained in traditional LRU (least recently used) order, with new allocations replacing the LRU element and being inserted at the MRU (most recently used) position.

Since the shadow directories are only intended to provide guidance for sizing the partitions, the shadow directories do not require data. Instead, each entry in the shadow directory merely consists of a tag to indicate the LRU position of the line had it remained in the cache. RWP maintains two global age hit counters: a clean age hit counter and a dirty age hit counter. These hit counters reflect the number of times cache lines were hit while at various LRU positions in the shadow directory. For example, each time a clean cache line is requested while in the MRU position of any of the 32 clean shadow directories, the MRU counter in the clean age hit counter is incremented. Similarly each time a dirty cache line is requested while in the

LRU position of the dirty shadow directory, the LRU position of the dirty age hit counter is incremented.

By comparing the values of the dirty and clean age hit counters, our mechanism can predict the number of additional hits and additional misses the cache will incur if one of the ways currently allocated tot a partition is re-allocated to the other. We use the same mechanisms proposed by Qureshi *et al.* [27] to determine the best partition size from the age hit counters. This allows RWP to determine the number of ways to allocate to the clean and dirty partitions.

3.3. Summary

An advantage of RWP is that it is confined to the last-level cache, and does not require any information from the processor or higher-level caches. However, RWP does not attempt to directly identify and deprioritize write-only cache lines. Write-only cache lines are expected to be eliminated as a by-product of changing the clean and dirty partition sizes. In the next section, we discuss a more complex mechanism that attempts to directly identify write-only lines. Our evaluations show that RWP achieves most of the benefit of such a mechanism at a fraction of the complexity and cost.

4. Read Reference Predictor

The key argument we make in earlier sections is that read-reuse, as opposed to write-reuse, is the critical value provided by the cache. An ideal mechanism could categorize cache lines before they are allocated based on their likelihood to be subsequently read. Write-only cache lines and those not reused at all should then bypass the cache to allow cache resources to be allocated to lines likely to service future reads. Prior work offers guidance on how we might accomplish this. Piquet *et al.* [25], building upon Tyson's past work [34], observed that some memory reference instructions request cache lines that would subsequently see no reuse, while others request cache lines that would be reused. They exploited this observation by using the PC of the memory instruction to predict the level of reuse its memory requests would exhibit.

We modify the approach in [25], using the PC of the memory instruction *not* to predict *general reuse* by all memory instructions, but instead focusing only on *reuse by subsequent reads*. In contrast to [25] which predicted general-reuse in cache lines to guide the replacement algorithm, we predict only read-reuse. Instead of allocating only those cache lines likely to service any reference (reads or writes), we allocate only those cache lines likely to service reads and bypass those likely to be only written to or not referenced at all. We note that our goal is to identify cache lines that are likely to be read as opposed to written to. Both load and store instructions may allocate lines that are likely to be read.

4.1. RRP Framework

Similar to RWP (Section 3), RRP relies on shadow directories added to a few sample sets to measure the read reuse of lines. Figure 6 illustrates how locality is tracked in a hypothetical 8-way set associative cache. A small number of the sets
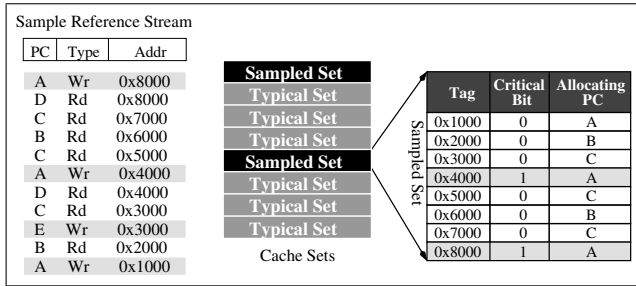
Figure 6: Sample sets in Read-Reference Predictor (RRP).

are augmented with an 8-way shadow directory. The shadow directory consists of a cache line tag, a critical bit (indicating read reuse), and a hashed value of the PC of the instruction responsible for allocating the cache line (hashed-PC).

On the left we show a sample reference stream consisting of a number of reads and writes, with the PC, the reference type, and the cache line tag address of the request. The reference stream begins with an instruction A writing to data address $0x8000$. The initial write allocates an entry for the cache line $0x8000$ in the shadow directory with the critical bit initially set to 0. A second reference, a read to $0x8000$ by instruction D causes the critical bit to be set. A subsequent write by A to the address $0x4000$ follows the same pattern. There are two key points we highlight with this example. First, the critical bits for both $0x4000$ and $0x8000$ are set, not because the allocating instruction was a read, but because both cache lines were reused by subsequent reads. The key behavior we would like to capture is not whether or not A itself is a read or write, but whether or not the cache lines A allocates are subsequently used by reads. Later in the reference stream we can see a contrasting example when instruction C reads cache line $0x3000$. Although cache line $0x3000$ is subsequently written by instruction E, the critical bit is not set because the critical bit is intended to capture only read reuse not write reuse. The distinction between read reuse and write reuse is what distinguishes our work from prior work on general reuse prediction, e.g., [34, 25]. While Piquet *et al.* [25] treated all references equally for the purposes of reuse prediction; RRP differentiates between read reuse and write reuse, effectively classifying write reuse as *no reuse*.

As cache lines are evicted from the shadow directories, the PC and critical bit of each evicted line are used to update a table of 2-bit saturating counters. As in [25], the table of 2-bit saturating counters, i.e., the RRP predictor table, lacks tags. The PC is used to index into the table and select the 2-bit counter, and the counter is incremented if the critical bit is set to 1 and decremented if the critical bit is set to 0. All subsequent memory instructions that perform allocations will first query the RRP predictor table. Instructions that map to a 2-bit counter set to 0 will bypass the cache, all other instructions will result in cache allocations.

4.2. RRP Implementation Issues

The key advantage of RRP over RWP is that it explicitly classifies memory requests as exhibiting read or write reuse.

In that sense, RRP is a more pure implementation of what RWP attempts to do conceptually. In practice, however, the implementation of RRP is quite complex in a last level cache.

The biggest issue that complicates RRP is that writeback requests are not associated with any program counter. However, we want to predict if a writeback would be followed by a read request to the same cache line. To achieve this goal, we record the first PC that brought each line into the L1 cache and supply this PC to the L2 and LLC when the line is written back from L1 and L2. This requires additional storage and logic complexity in core caches (L1, L2) to store and update the hashed-PC value with the cache line tags.

Providing PC information to the LLC has been proposed earlier (e.g., [36, 34, 15, 10]). This information is critical to provide good prediction accuracy at the LLC. However, providing PC information to the LLC consumes more on-die bandwidth, and would require the on-die interconnect to handle a different transaction type than just address and data transactions. As we show in our results, our simpler RWP mechanism, which is confined to the LLC, has competitive performance with the more complex RRP mechanism.

5. Evaluation Methodology

5.1. Baseline Configuration

We use CMP$im [5], a Pin-based [20] x86 simulator. We use a framework similar to the Cache Replacement Championship [4]. Our baseline processor is a 4-wide out-of-order processor with a 128-entry reorder buffer and a three-level cache hierarchy, similar to many of Intel®'s mainline cores. We use a 32KB, 4-way L1 instruction cache, a 32KB, 8-way L1 data cache, a 256KB 8-way L2 cache, and a 4MB, 16-way shared non-inclusive/non-exclusive last-level cache. All caches use 64-byte lines, and implement the LRU replacement policy as the baseline. The load-to-use latencies for the L1, L2, and LLC caches are 1, 10, and 30 cycles, respectively. In our model, the core does not stall for write requests to bring data into the cache. Only a subsequent load hitting on an unfilled line stalls if the store has not yet returned from the memory. We assume on-die interconnects can transfer 32-bytes per cycle between the L1 and L2, and between the L2 and the LLC. This increases network congestion and latency when writeback traffic contends with cache misses for the shared interconnect. We model a 200-cycle latency to main memory, and support a maximum of 32 outstanding misses to memory. We implement four memory controllers, each with a 6GB/sec bandwidth to memory, for a total of 24GB/sec memory bandwidth. Our simulation methodology is similar to prior cache studies [28, 6, 10, 11].

5.2. Benchmarks

We simulate benchmarks from the SPEC CPU2006 suite with the reference input set. For SPEC benchmarks, we use SimPoint [24] to identify a single characteristic interval (i.e., simpoint) of each benchmark. We first run 50 million instructions to warm up the internal structures and then run
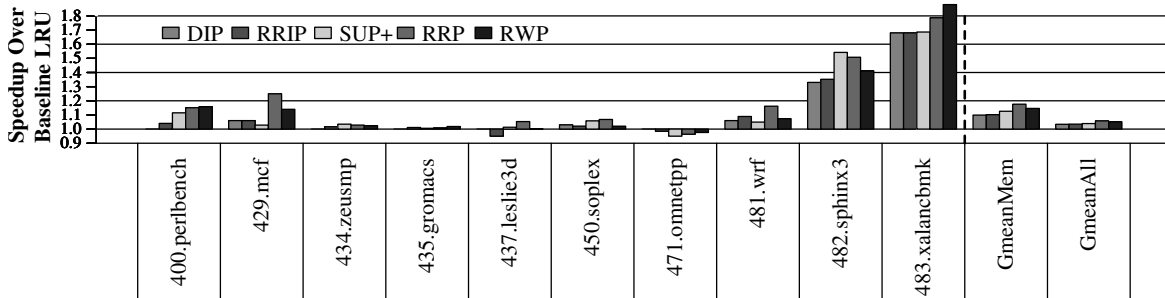
Figure 7: Speedup over baseline LRU.

the simulation for 500 million instructions. For most results in Section 6, we use a cache-sensitive subset of the SPEC CPU2006 benchmarks. The benchmarks that have more than 5% speedup when LLC size is increased from 4MB to 8MB are considered as cache-sensitive applications. We also report average results from the entire SPEC suite.

In our multi-core analysis, we use 35 randomly picked multi-programmed workloads from SPEC running on a 4-core configuration. Our workloads include five homogeneous workloads (with four instances from 429.mcf, 434.zeusmp, 456.hmmer, 462.libquantum and 482.sphinx3) while the remaining thirty were heterogeneous workloads. Each benchmark runs simultaneously with the others, restarting after 250 million instructions, until all of the benchmarks have executed at least 250 million instructions. We report the weighted speedup normalized to LRU. That is, for each thread i sharing the last-level cache, we compute IPC_i. Then we find $SingleIPC_i$ as the IPC of the same program running in isolation with LRU replacement. Then we compute the weighted IPC as $\sum IPC_i/SingleIPC_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

5.3. Simulated Cache Configurations

We compared RRP and RWP to the following configurations, none of which takes into account read-write criticality.
Baseline. We use LRU as our baseline policy.
Dynamic Insertion Policy (DIP). DIP is an insertion policy that can dynamically choose to insert cache lines either at MRU or LRU position [28] in order to eliminate cache thrashing. It protects a portion of the working set of a thrashing workload by inserting lines at the LRU position in the LRU stack, and protects recently-used lines in LRU-friendly workloads by inserting lines at the MRU position.
Re-Reference Interval Predictor (RRIP). RRIP is another insertion policy that avoids thrashing and scanning by inserting cache lines either at the end of the recency stack or the middle of the recency stack [6]. We compare with the best performing version of DRRIP in our simulation environment.
Single-Use Predictor (SUP). SUP uses a PC-based predictor to identify cache lines that are referenced only once [25]. However this work does not describe how to associate a writeback with a PC. Other PC-based predictors [10, 36] also bypass this issue by excluding writebacks from the prediction mechanism. SUP counts write accesses as references, and

does not take advantage of non-criticality of some dirty lines. In addition, SUP does not have a feedback mechanism to adapt to workload changes. For a fair comparison, we add the sampling based feedback loop to this mechanism. We call this mechanism **SUP+**.

6. Results

In this section, we compare our two mechanisms, RRP and RWP, to a variety of alternative configurations, enumerated in Section 5.3. We focus mainly on memory-intensive workloads, whose geometric mean performance is denoted as GmeanMem. The geometric mean performance of *all* SPEC benchmarks, denoted as GmeanAll, is included with our performance results to show the effect of our algorithms on all workloads.

6.1. Single Core

Speedup. Figure 7 shows the speedups for the cache sensitive workloads for a variety of replacement algorithms compared to a baseline cache with LRU. On average (Gmean-Mem), RRP improves performance by 17.6% and RWP improves performance by 14.6%, when compared to the baseline. Our worst-case performance occurs when running 471.omnetpp. We lose less than 3% performance, largely due to sampling error in our sampling-based predictors. Our best case performance comes from 483.xalancbmk, where we improve performance by 88% relative to the baseline. When considering all SPEC CPU2006 workloads (GmeanAll), RRP and RWP provide 6.2% and 5.1% speedup over the baseline. We note that RRP and RWP also deliver substantial performance gains when compared to three state-of-the-art replacement algorithms. For memory-intensive workloads, RRP achieves 7.9%, 7.4%, and 5% performance gains over DIP, RRIP, and SUP+ respectively; while RWP delivers speedups of 6.8%, 6.3%, and 3.9% over DIP, RRIP, and SUP+.

Effect on Load Misses and Write Traffic. One of the key attributes of our approach is that read misses will sometimes get reduced at the expense of increased write misses. In Figure 8, we show LLC load misses normalized to the LRU baseline. Our mechanisms reduce load misses for most benchmarks since they favor lines likely to be read. On average, RRP/RWP reduce load misses by 30%/29% relative to the baseline. RRP and RWP achieve 15%, 15%, 10% reduction in
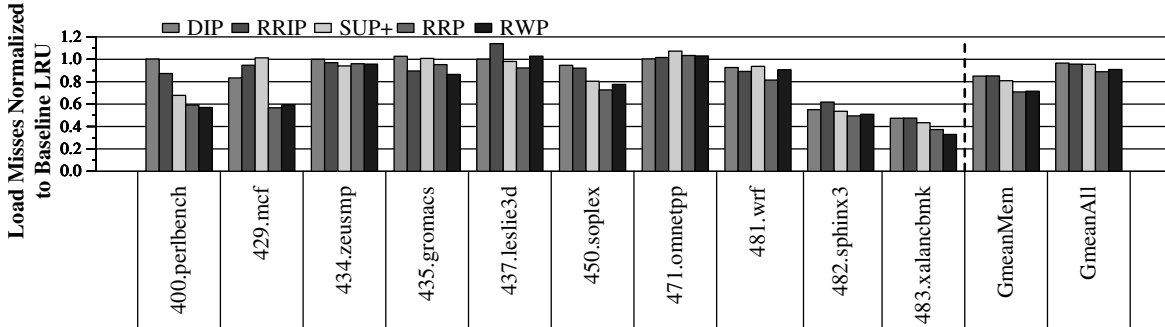
Figure 8: Normalized load miss count over baseline LRU.
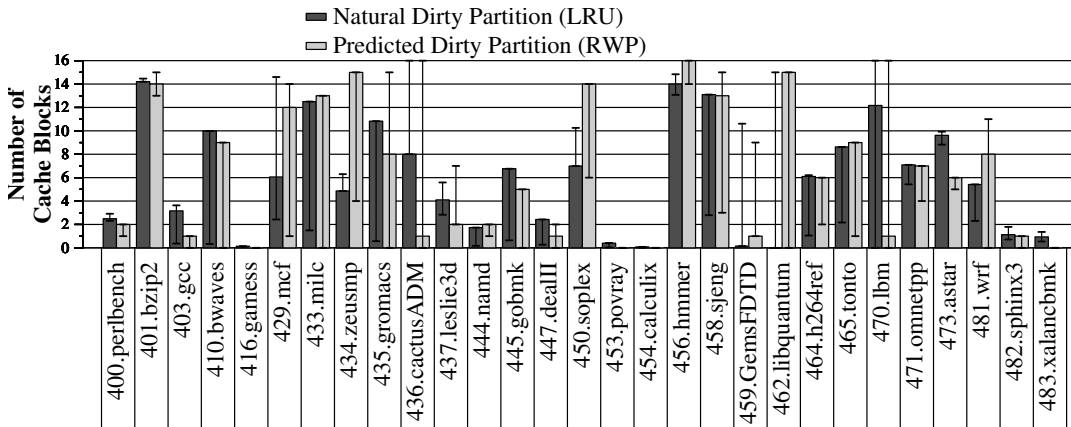

Figure 11: Comparing *natural* dirty partition size with LRU to the *predicted* size from RWP.
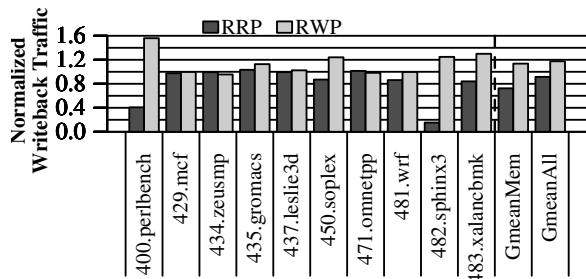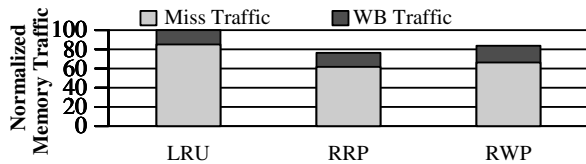

Figure 9: Write traffic over baseline LRU.

Figure 10: Memory traffic over baseline LRU.

load misses over DIP, RRIP, and SUP+. However, the reduction in load misses comes at the expense of an increase in write traffic. Figure 9 shows the writeback traffic to main memory for RWP and RRP normalized to the baseline LRU. The increase in write traffic is significant (17%) for RWP. RRP actually manages to reduce write traffic by 9% overall. This is because RRP bypasses any line that is unlikely to be read, resulting in bypassing clean lines that are unlikely to be reused at all, and provides extra space for read lines. Therefore, more read-

dirty lines can buffer writes in the cache, resulting in fewer writebacks. RWP, on the other hand, provides no facility to bypass lines, but relies on sizing the dirty partition to maximize read hits. If clean lines show more read utility, RWP evicts dirty lines earlier, increasing the number of writebacks from the LLC. This is the behavior demonstrated by 400.perlbench, 429.mcf, 450.soplex, 482.sphinx3, and 483.xalancbmk. On the other hand, if dirty lines exhibit more read utility, RWP protects dirty lines and decreases the number of writebacks. This is the case for 456.hmmer (not shown) and 434.zeusmp, both of which exhibit reductions in writeback traffic. Figure 10 shows that even though RWP increases writeback traffic, it still manages to reduce overall main memory traffic by an average of 16% over the baseline LRU due to the much higher reduction in load misses (29% on average).

Importance of the Dynamic Partitioning Policy. One key mechanism of RWP is that it dynamically resizes the clean and dirty partitions based on which is expected to get more read reuse. Figure 11 illustrates the impact of our dynamic partitioning policy. For each benchmark, we show two data points: (1) the *natural* dirty partition size at the end of benchmark run with traditional LRU replacement; (2) The *predicted* dirty partition size from RWP at the end of the simulation. While both data points are similar for some benchmarks, there are wide differences for a significant number of benchmarks. For example, 462.libquantum sees almost no dirty blocks in the cache
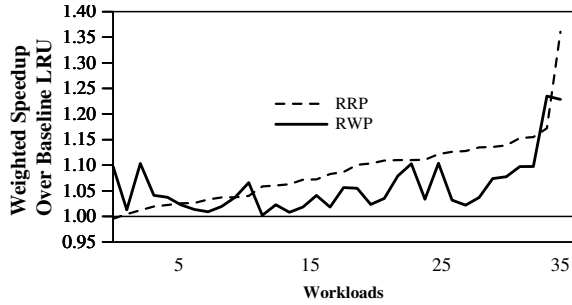
9

Figure 12: Normalized weighted speedup over LRU in 35 multi-programmed workloads.
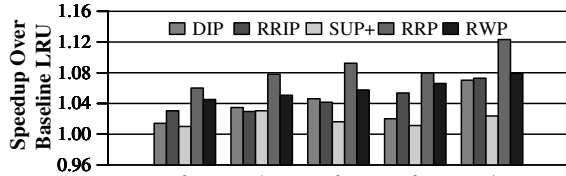

Figure 13: Normalized weighted speedup over LRU when the number of memory-intensive applications in a multi-programmed workload mix is varied.

with LRU, while the predicted dirty partition size is 15 out of 16 lines in each cache set. That is, RWP protects dirty lines from eviction in 462.libquantum since they get a large fraction of read hits. Conversely, 470.lbm has dirty lines filling three quarters of the cache with LRU, while the predicted dirty partition size is 1 out of 16 lines in each cache set. That is, RWP protects clean lines from eviction in 470.lbm since they get a large fraction of read hits. However, we need a dynamic policy since the best partition size changes significantly for many workloads during their execution. The ranges (drawn as error bars) shown for each bar presents the minimum and maximum dirty partition size out of all partition sizes logged every 100 million instructions. For example, 470.lbm's predicted dirty partition size varies between 0 and 16 (i.e., either no dirty lines or all lines are in the dirty partition). Most of the applications exhibit large differences in partition size during different phases of their execution (e.g., 429.mcf, 435.gromacs, 481.wrf, 433.milc, 350.soplex). However, a smaller number of workloads have less variation across their execution (e.g., 416.gamess, 483.xalancbmk). We conclude that RWP's partitioning scheme is effective at adjusting the partition size dynamically to maximize read hits.

6.2. Multi-Core

We evaluated a 4-core system and 35 multi-programmed workloads, each consisting of a random mix of SPEC workloads. These workloads include both memory-intensive and non-memory-intensive applications. They also include write-sensitive and non-write-sensitive applications. Figure 12 shows the weighted speedup improvement over LRU for RRP and RWP across all the workloads, denoted as a curve sorted based on the improvement obtained with RRP. On average, RRP improves system throughput by 8.4% and RWP improves system throughput by 6.2%. The performance improvement

of DIP, RRIP, and SUP+ (not shown in the figure) is 3.9%, 4.2%, and 2.0%, respectively over the baseline LRU. Although SUP+ performed well in our single core experiments compared to DIP and RRIP, it performs poorly in our multi-core configuration compared to these two schemes. SUP+ suffers from its inability to characterize and predict the behavior of writeback requests; and therefore writeback requests are always allocated in the LLC. In multi-core systems, writebacks have a greater impact on cache capacity due to increased contention in the cache.

We analyzed the sensitivity of performance gains to the number of memory-intensive workloads in 35 multi-programmed workload mixes. Figure 13 shows the average speedup (over LRU) for different mechanisms when the number of memory-intensive applications in a mix grows from zero to four. In general, both RRP and RWP outperform all other mechanisms in all configurations. The performance benefit for RWP and RRP grows when the number of memory-intensive applications in a workload increases. With no memory-intensive applications in the mix, RWP/RRP has a 4.5%/6% gain over LRU. However, with all memory-intensive applications in the mix, the performance advantage grows to 8%/12% for RWP/RRP. This is due to the success of RWP and RRP in eliminating less-critical write-only lines from the cache, leaving more room for critical read lines.

6.3. Storage Overhead

Table 1 compares the storage overhead of RRP and RWP mechanisms. The overhead can be categorized into three types: sampled sets, counters, and core cache overhead.

Sampled Sets. RRP/RWP has 512/32 sampled sets for training the predictor; there are 16 lines in each of the sampled sets. The number of sampled sets for RRP is higher than RWP, as PC-based sampling predictors require more sets to learn the behavior of accesses brought by a PC [10, 35]. Each line in the sampled sets in RRP has partial PC (15 bits), partial tag (15 bits), a critical bit, a valid bit, and LRU states (4 bits), a total of 36 bits. On the other hand, RWP does not require PC information, so the sampled lines require only 21 bits each (15 bits for tag, a valid bit, a dirty bit, and 4 bits for LRU states).

Counters. The predictor table in RRP has 2-bit saturating counters, where RWP age hit counters are 12 bits long. However, RWP has only 32 counters to track the read hits, but the predictor table of RRP consists of 16384 counters.

Core Cache Overhead. RRP keeps PC information in the L1 and L2 caches, which adds significant cost (15 bits for each line). However, there is no such overhead in RWP.

The RRP mechanism requires nearly 49KB of state overhead, where RWP uses only 2.67KB of state, amounting to only 5.4% of RRP's state overhead. RWP's additional state corresponds to only 0.06% of the total LLC capacity.

6.4. Summary

In this section, we showed that our RWP and RRP predictors outperform the baseline LRU and three state-of-the-art cache management mechanisms. RRP slightly outperforms

Type	RRP	RWP
Each line in sample sets	36 bits	21 bits
Number of sample sets	512	32
Types of sample sets	1	2
Sampled set overhead	36KB	2.62KB
Counter size	2 bits	12 bits
Number of counters	16384	32
Counter overhead	4KB	0.05KB
PC overhead in L1 and L2	8.43KB	0
Total overhead	48.43KB	2.67KB

Table 1: Storage overhead of RRP and RWP.

RWP for single-core and multi-core workloads. However, RRP requires changes to the core caches to store program counter information, passing information from the processor and L1 and L2 caches to the LLC. RWP, on the other hand, requires changes to only the LLC, while delivering close-enough performance to RRP. We conclude that RWP is an efficient mechanism that can exploit the read-write criticality differences to minimize read misses in caches.

7. Qualitative Comparison to Prior Work

7.1. Cache Management

There has been a significant amount of work on last-level cache replacement policies [23, 2, 6, 10, 28, 37, 26, 22]. However, none of these prior proposals takes into account the criticality difference between reads and writes. We briefly describe some of the prior works in cache replacement policies. Recent replacement policies can be broadly divided into two categories: insertion-based and partition-based. The insertion-based mechanisms dynamically insert incoming lines at different stack positions based on their reuse [28, 6, 31, 37]. Dynamic Insertion Policy (DIP) places incoming lines at LRU/MRU position [28]. Re-Reference Interval Prediction (RRIP) dynamically inserts lines near the end of the recency stack or at the end of the stack [6]. The Evicted-Address Filter predicts the reuse behavior of cache lines by keeping track of their recency or reuse after eviction and modifies the replacement and insertion policies based on that prediction [31]. The partition-based mechanisms partition the cache lines based on recency, frequency and/or locality [23, 2, 19, 8, 23]. Adaptive tuning policies like ARC [23] and CAR [1] divide the cache into non-referenced and re-referenced lines protecting the re-referenced lines. Other works propose to divide the cache lines into different partitions according to temporal locality to protect lines with higher locality [19, 8].

Another area of research predicts the last touch of cache lines [14, 15, 34]. Dead block predictors can detect when a block is accessed for the last time and evict them to make room for useful lines [10, 12, 36]. Some replacement algorithms take into account the reuse behavior of blocks [9, 21, 11], but these mechanisms do not differentiate between read and write-only lines. Piquet *et al.* proposed a PC-based prediction mechanism [25] that is similar to our read reference predic-

tor. However, they do not exploit the fact that write requests are not on the critical path, and consider both read and write requests as re-references.

Utility-based cache partitioning (UCP) partitions the cache ways among the cores according to the usage of each core [27]. We have used a similar technique to determine the best possible partitioning for reads and writes. Our mechanism also takes advantage of cache set sampling to approximate the cache behavior from a few sets [26]. Qureshi *et al.* [26] first showed data indicating that cache behavior is more or less uniform across all sets, on average.

7.2. Mechanisms Exploiting Writes

Though no work has explicitly exploited the differences between reads and writes in caches, reads and writes are treated differently in the memory controller due to the physical timing constraints of DRAM read, write and read-to-write turn around cycles [18, 17, 33, 35]. Recent works in phase-change memory (PCM) and spin-transfer torque memory (STT-RAM) also try to reduce the number of writes to tolerate the higher write latency [16, 29, 30, 38, 13].

Some mechanisms [17, 33, 35] evict dirty lines proactively from the cache to increase DRAM row buffer hit rate. These mechanisms focus on exploiting spatial locality in DRAM as opposed to improving cache performance.

7.3. Mechanisms Exploiting Critical Loads

There are prior works exploiting the criticality differences among loads. A prior work proposed to dynamically determine the criticality of loads and prioritize them over non-critical ones [32]. Cost-sensitive cache replacement policies try to take into account the cost associated with misses and protects cache lines with high cost [7]. Memory-level parallelism (MLP)-aware cache showed that isolated misses are costlier than parallel misses and proposed a mechanism that tries to minimize costly isolated misses [28]. Another work characterized the static stores that can indicate criticality of future loads and proposed a prefetching mechanism to exploit this behavior [3]. These works do not exploit the criticality gap between read and write-only lines in the cache hierarchy.

8. Conclusions and Future Directions

In this paper, we exploit the observation that cache read requests are frequently on the critical path whereas cache writes are not, to design better cache management techniques that improve system performance. We propose cache management mechanisms to increase the probability of cache hits for critical reads, potentially at the expense of less-critical write requests, by allocating more cache resources to lines that are reused by reads than to lines that are reused *only by writes*. The key contribution of this paper is the new idea of distinguishing between *read lines* vs. *write-only lines* to design cache management policies that favor read lines. Our Read-Write Partitioning (RWP) mechanism dynamically partitions the cache into clean and dirty partitions to maximize

read hits. It protects the partition that receives more read requests by evicting lines from the other partition that exhibits less read reuse. We discuss a more complex yet more effective mechanism, Read Reference Predictor (RRP), which uses program counter information to directly predict cache lines that are reused by reads. Our mechanisms provide better performance over several prior cache management mechanisms. For a single-core system, RWP has an average speedup of 14.6%/6.3% over LRU/RRIP for cache-sensitive workloads. On a 4-core system, RWP improves average system throughput by 6.2%/2% over LRU/RRIP, across a wide set of multi-programmed workloads. We show that RWP performs within 3% of RRP but requires only 5.4% of RRP's state overhead.

A limitation of our proposal that future work can address is that it does not specifically address multi-threaded workloads that share LLC lines or workloads where different threads have different requirements for read vs. write partition sizes. Another future research direction is analyzing the trade-off between read and write-only lines in systems with non-volatile cache or memory. Write operations in emerging non-volatile memory technologies have limited endurance and/or require high energy and latency [16, 13]. A read-write partitioning mechanism can dynamically adjust the write partition considering the write latency, energy, endurance, and performance. We hope that the distinction between read and write-only lines drawn in this paper will enable other future mechanisms that can exploit this distinction.

Acknowledgement

References

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

[2] M. Chaudhuri. Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. MICRO, 2009.

[3] A. Holloway and G. S. Sohi. Characterization of problem stores. *IEEE Comput. Archit. Lett.*, 2004.

[4] A. Jaleel et al. 1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship.

[5] A. Jaleel et al. CMP$im: A pin-based on-the-fly single/multi-core cache simulator. MoBS, 2008.

[6] A. Jaleel et al. High performance cache replacement using re-reference interval prediction. ISCA, 2010.

[7] J. Jeong and M. Dubois. Cost-sensitive cache replacement algorithms. HPCA, 2003.

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

[9] G. Keramidas et al. Cache replacement based on reuse-distance prediction. ICCD, 2007.

[10] S. Khan et al. Sampling dead block prediction for last-level caches. MICRO, 2010.

[11] S. Khan et al. Decoupled dynamic cache segmentation. HPCA, 2012.

[12] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. on Computers*, 2008.

[13] E. Kultursay et al. Evaluating STT-RAM as an energy-efficient main memory alternative. ISPASS, 2013.

[14] A.-C. Lai et al. Dead-block prediction & dead-block correlating prefetchers. ISCA, 2001.

[15] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. ISCA, 2000.

[16] B. C. Lee et al. Architecting phase change memory as a scalable DRAM alternative. ISCA, 2009.

[17] C. J. Lee et al. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. Technical Report TR-HPS-2010-002, 2010.

[18] H.-H. Lee et al. Eager writeback - A technique for improving bandwidth utilization. MICRO, 2000.

[19] G. H. Loh. Extending the effectiveness of 3D-stacked DRAM caches with an adaptive multi-queue policy. MICRO, 2009.

[20] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. PLDI, 2005.

[21] R. Manikantan et al. NUcache: An efficient multicore cache organization based on next-use distance. HPCA, 2011.

[22] R. Manikantan et al. Probabilistic shared cache management (PriSM). ISCA, 2012.

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

[24] E. Perelman et al. Using simpoint for accurate and efficient simulation. SIGMETRICS, 2003.

[25] T. Piquet et al. Exploiting single-usage for effective memory management. ACSAC, 2007.

[26] M. Qureshi et al. A case for MLP-aware cache replacement. ISCA, 2006.

[27] M. Qureshi et al. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. MICRO, 2006.

[28] M. Qureshi et al. Adaptive insertion policies for high performance caching. ISCA, 2007.

[29] M. Qureshi et al. Scalable high performance main memory system using phase-change memory technology. ISCA, 2009.

[30] M. Qureshi et al. Improving read performance of phase change memories via write cancellation and write pausing. HPCA, 2010.

[31] V. Seshadri et al. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. PACT, 2012.

[32] S. T. Srinivasan et al. Locality vs. criticality. ISCA, 2001.

[33] J. Stuecheli et al. The virtual write queue: Coordinating DRAM and last-level cache policies. ISCA, 2010.

[34] G. Tyson et al. A modified approach to data cache management. MICRO, 1995.

[35] Z. Wang et al. Improving writeback efficiency with decoupled last-write prediction. ISCA, 2012.

[36] C.-J. Wu et al. SHiP: Signature-based hit predictor for high performance caching. MICRO, 2011.

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

[38] P. Zhou et al. A durable and energy efficient main memory using phase change memory technology. ISCA, 2009.