Bandwidth-aware Last-level Caching: Efficiently Coordinating Off-chip Read and Write Bandwidth

Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
mainakc@cse.iitk.ac.in

Jayesh Gaur Sreenivas Subramoney
Processor Architecture Research Lab
Intel Corporation
jayesh.gaur@intel.com, sreenivas.subramoney@intel.com

***Abstract*— The last two decades have witnessed a large number of proposals on the last-level cache (LLC) replacement policy aiming to minimize the number of LLC read misses. Another independent large body of work has explored mechanisms to address the inefficiencies arising from the DRAM writes introduced by the LLC replacement policy. These DRAM scheduling proposals, however, leave the LLC replacement policy unchanged and, as a result, miss the opportunity of synergistically shaping and scheduling the DRAM write bandwidth demand. In this paper, we argue that DRAM read and write bandwidth demands must be coordinated carefully from the LLC side and hence, introduce bandwidth-awareness in the LLC policy. Our *bandwidth-aware LLC policy* proposal enables long uninterrupted stretches of DRAM reads while maintaining the efficiency of the last-level cache and controlling precisely when and for how long writes can demand DRAM bandwidth. Our proposal comfortably outperforms the state-of-the-art eager DRAM write scheduling proposals and bridges 75% of the performance gap between the baseline and a hypothetical system that deploys an unbounded DRAM write buffer.**

***Index Terms*—DRAM bandwidth, Last-level caching, DRAM writes**

I. Introduction

The last-level cache (LLC) policies have come a long way progressively improving the fraction of code and data reuses captured by the LLC. Driven by the traditional goals of cache design, the LLC replacement policy selects the victims to minimize the number of read misses. Dirty victims generate write traffic to the main memory DRAM. To reduce the impact on read throughput, DRAM writes are drained periodically in bunches from small write buffers. As a result, a steady stream of dirty victims periodically interrupts DRAM reads making it impossible to offer the much-needed long stretches of exclusive DRAM read bandwidth. DRAM writes introduce three types of inefficiencies: (i) channel turn-around delay when switching between reads and writes, (ii) additional delay of precharging DRAM banks and activating new DRAM rows due to lack of row locality in the write stream, and (iii) the DRAM bandwidth taken away by the writes. To reduce channel turn-around delay and improve row locality of writes, DRAM writes are drained in bunches from a per-channel write buffer. Proposals focusing specifically on write scheduling further improve the DRAM efficiency by identifying row-local groups of dirty blocks from the LLC [36], [64], [73] and scheduling them eagerly for writing back to DRAM whenever some DRAM resource falls idle [37], [64], [73], [75]. Prior research has also explored ways to overlap reads with writes [6]. All these proposals leave the LLC replacement policy unaltered and as a result, miss the opportunity of precisely coordinating the read and write bandwidth demands to the off-chip DRAM.

In this paper, we quantitatively establish that smart coordination between off-chip read and write bandwidth demands holds the key to improving DRAM efficiency. Based on this observation, we present bandwidth-aware LLC policies that maximize the length of uninterrupted exclusive DRAM read bandwidth stretches and control exactly when the writes can interrupt the read flow. We list our contributions in the following.

1. We motivate our bandwidth coordination-driven design principles by establishing quantitatively that data channel turn-arounds and poor locality of writes have much less performance impact than the bandwidth taken away by DRAM writes (Section III).
2. We present bandwidth-aware LLC policies that control when and for how long DRAM bandwidth can be demanded for reading and writing by learning the run-time read/write characteristics of workloads (Section IV).
3. Our proposal outperforms a well-tuned eight-core baseline chip-multiprocessor model (Section VI) by 12% and comes close to a hypothetical baseline that deploys an unbounded DRAM write buffer (Section VII).
4. We verify through RTL synthesis that the design additions of our proposal have low area overhead per LLC bank and that our design meets the necessary timing constraints to operate at 4 GHz frequency.

II. Background

On evicting a dirty block, the multi-banked shared LLC generates a write request to one of the DRAM channels. If the channel controller has space to enqueue an arriving write into the channel's write buffer, it sends a positive acknowledgment to the sender LLC bank. A negatively acknowledged write must be retried by the LLC bank from its local buffer which maintains each of the in-flight writes sent from the bank until its receipt is positively acknowledged by the memory/channel controller. If the local buffer of an LLC bank is full, the bank stops sending read misses to memory.

The DRAM data bits on a channel are organized into a number of ranks, each having several banks. Figure 1 shows the physical address mapping used in this study. A DRAM access needs to activate or open a row in the target bank and move the row contents to a per-bank row buffer. Subsequently, the appropriate columns can be read out from or written to the row buffer honoring the row-to-column delay (tRCD). In this paper, we use the first-ready-first-come-first-serve (FR-FCFS) DRAM access scheduling algorithm, which schedules accesses to already open rows first and breaks ties among them by the arrival order of the accesses. To open a new row in a bank, the bank must be precharged to a state that allows activation of a new row after the precharge delay (tRP). In this paper, the write scheduling algorithm drains writes from the channel's write buffer when either there are no pending reads on the channel (will be referred to as opportunistic minor write draining mode) or the write buffer is full (will be referred to as major write draining mode) [36], [73]. Minor write draining terminates when a read arrives on the channel. Major write draining terminates when the write buffer is empty.

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00022

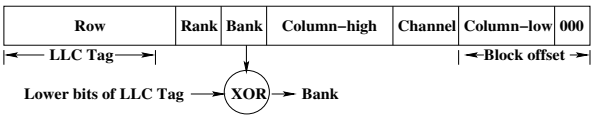

Fig. 1. DRAM address mapping scheme.

III. Motivation

The studies in this section first show that the performance improvement achieved by addressing the commonly known DRAM write overheads, namely channel turn-arounds and lack of row locality in the write stream, is not a sizable fraction of what is achievable optimally. Next, a set of studies brings out the necessity of proactively coordinating the DRAM read and write bandwidth demands for achieving significantly higher performance benefits. All studies presented in this section are conducted on a simulated eight-core chip-multiprocessor having an 8 MB 16-way shared LLC and a dual-channel DDR3-1600 11-11-11-28 DRAM system.

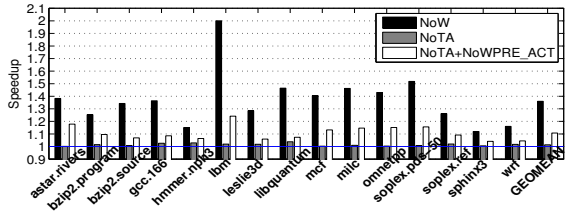

Fig. 2. Perf. impact of turn-around, write locality, and write bandwidth.

Figure 2 quantifies the relative importance of channel turn-around, write locality, and write bandwidth through comparison of the speedup achieved by discarding all writes (NoW in the leftmost bar)[1], by nullifying channel turn-around penalty (NoTA in the middle bar), and by nullifying channel turn-around as well as precharge/activate penalty during write draining modes (NoTA+NoWPRE_ACT in the rightmost bar). Each group of bars represents an eight-way multiprogrammed workload consisting of eight copies of the mentioned SPEC CPU 2006 benchmark application. In this study, each DRAM channel has a 32-entry write buffer. NoTA and NoTA+NoWPRE_ACT achieve average speedups of 1.4% and 11% respectively, while NoW achieves 36% average speedup indicating that addressing turn-around and precharge/activate penalty covers less than one-third of the potential speedup achievable through write optimization. The remaining about two-third of the speedup gap arises from DRAM write bandwidth demand and can be narrowed down through better coordination between DRAM read and write bandwidth demands.

One naïve way of achieving bandwidth coordination and long uninterrupted DRAM read stretches is by having larger per-channel write buffers. Figure 3 shows that the average speedup progressively increases with per-channel write buffer capacity. However, since an incoming read on a channel must search the write buffer for the youngest matching block address (if any), the write buffer needs to have an address CAM (content associative memory). Therefore, practically implementable write buffer sizes are limited to a few tens of entries. Nonetheless, an analysis of the performance improvement with larger write buffers can offer useful insight into the dynamics of read/write bandwidth interaction. Larger write buffers offer better performance due to three reasons: (A) fewer channel turn-arounds, (B) improved row locality of writes arising from better write bunching, and (C) long uninterrupted stretches of sustained high read throughput. Figure 4 isolates the impact of these three

[1] Modeled by not sending any write to the simulated DRAM arrays.

when the number of write buffer entries per channel is increased from 32 in the baseline to 256 (upper panel) and 8192 (lower panel). The WB256 and WB8K bars are same as the 256-entry and 8K-entry bars in Figure 3. The middle bar quantifies the combined speedup due to (B) and (C) by nullifying the turn-around penalty (WBn_NoTA) in both baseline and enlarged write buffer configurations. The rightmost bar quantifies the speedup due to (C) by nullifying the turn-around penalty as well as precharge and activation penalty during write draining modes in both baseline and with enlarged write buffers (WBn_NoTA+NoWPRE_ACT). Note that the baseline in the middle bar has zero turn-around penalty and that in the rightmost bar has zero turn-around penalty as well as zero precharge and activation penalty during write draining modes, while the baseline in the leftmost bar has both these penalties enabled. The speedup due to (A) alone is the gap between the leftmost bar and the middle bar. The speedup due to (B) alone is the gap between the middle bar and the rightmost bar. Therefore, the speedup achieved with 256 write buffer entries arises primarily due to (B), while the speedup for 8192 write buffer entries arises from a combination of (B) and (C). The speedup due to (C) is mostly non-existent for 256 entries, but becomes reasonably important across the board for 8192 entries indicating that very large write buffers are needed for read/write bandwidth coordination and optimization beyond (A) and (B). Our proposal realizes this coordination with bandwidth-aware LLC policies.

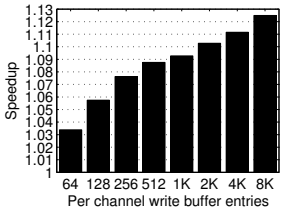

Fig. 3. Increasing number of write buffer entries.

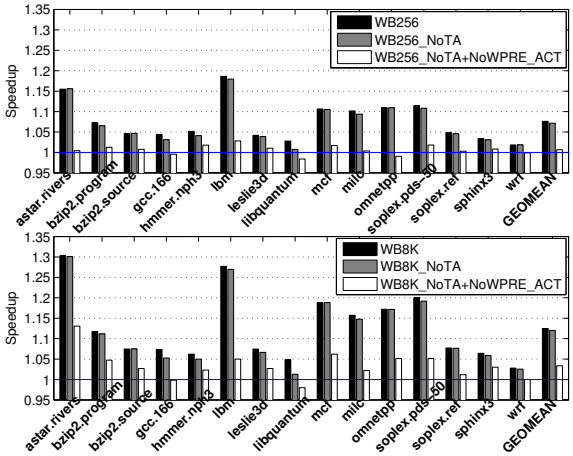

Fig. 4. Analysis for write buffers with 256 (upper) and 8192 (lower) entries.

Figure 5 studies a configuration with an unbounded write buffer as the extreme case of configurations with large write buffers. Since an unbounded write buffer never gets full, writes drain only through the minor write draining mode when there are no reads (WnoR). The leftmost bar (Unbounded_WnoR) shows an average speedup of 26% in this case, while the second bar (Unbounded_WnoR+DrainAtEnd) brings down the achievable speedup bound to 13% on average when the cycles needed to drain the residual buffered writes at the end are

included. In the latter case, the percentage of writes drained during the terminal phase for each workload is also shown on top of the bars. These writes drained during the terminal phase will be referred to as terminal writes. High read bandwidth demand throughout offers little scope for opportunistic write draining leading to high percentages of terminal writes in lbm, libquantum, milc, soplex.pds-50. For astar.rivers, bzip2.program, hmmer.nph3, and omnetpp, most writes can be drained through the minor draining modes alone leaving small percentages of terminal writes. As the rightmost two bars of Figure 5 shows, these applications, however, perform very poorly when the unbounded write buffer is replaced by 32-entry (WB32_WnoR) and 8192-entry (WB8K_WnoR) write buffers per channel while keeping only the minor write draining mode enabled. In this configuration, when the write buffers fill up, the LLC controller eventually stops sending read misses due to back-pressure and the DRAM controller is forced to enter a minor write draining mode. However, the minor draining mode lasts for a short while because as soon as the write buffers have space, the LLC controller starts sending read misses again. The results of this configuration indicate that extremely large write buffers are needed (more than even 8192 entries) for these workloads to keep buffering the writes between two large consecutive read bandwidth holes (periods with no read requests). The buffered writes can then be drained during the read bandwidth holes using the minor write draining mode.

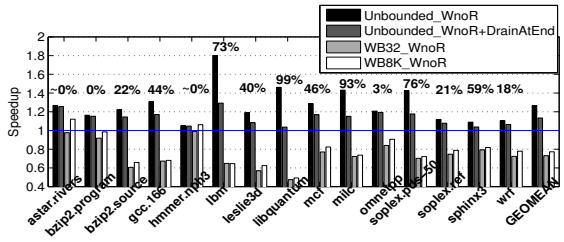

Fig. 5. Speedup with unbounded write buffer.

In summary, these studies establish the importance of off-chip bandwidth coordination. Our proposal discussed next incorporates such coordination through bandwidth-aware LLC policies.

IV. Bandwidth-aware LLC Policies

Our LLC policy proposal attempts to offer long uninterrupted stretches of exclusive DRAM bandwidth to the reads as much as possible. We initially assume the true LRU replacement policy and later extend our proposal to other more practical LLC replacement policies. We assume an inclusive LLC. However, our proposal works seamlessly in non-inclusive and exclusive LLCs as well.

A. Exclusive DRAM Read Bandwidth

Our proposal implements the clean LRU policy to delay replacement of dirty blocks from the LLC and guarantee long uninterrupted stretches of exclusive DRAM bandwidth to reads. This policy evicts the clean block closest to the LRU position within the target LLC set; the LRU block is evicted if the set is completely dirty. The clean LRU policy has been used for reducing off-chip write traffic [13], [79] and improving non-volatile memory and cache [14], [25], [48], but our primary contributions discussed next involve the mechanisms to maximize the stretch of clean block eviction while bounding the loss in LLC hits.

B. Dynamic Write Buffering Capacity

The clean LRU policy tends to deviate from the baseline LRU policy as the number of dirty blocks buffered toward the tail of the LRU stack of the LLC grows. This deviation may lead to loss in LLC hits depending on the sensitivity of the workload toward LLC capacity. The dirty block population toward the tail of the LRU stack acts as an in-LLC write buffer (Figure 6) holding the dirty blocks. Bounding the fraction of sacrificed LLC hits requires restricting the number of LLC ways the in-LLC write buffer can occupy. In the following, we discuss how the maximum allowable write buffer width is computed dynamically.

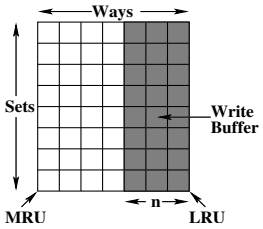

Fig. 6. Last n ways of the LRU stack are marked as the in-LLC write buffer.

Let us suppose that the last n positions of the LRU stack in an LLC set are occupied by dirty blocks. We would like to derive an upper bound on n. A clean LRU victim in such a set would sacrifice the read hits that it could have enjoyed in the LRU stack's last n positions. To estimate the fraction of sacrificed read hits, each LLC bank maintains a read hit histogram (RHH) array of length equal to the associativity, A, of the LLC (Figure 7). The array index i records the proportion of read hits at LRU stack position i. Therefore, the fraction of sacrificed read hits in the aforementioned case is $\sum_{i=0}^{n-1} RHH(B,i)/T_{rh}(B)$, where $i=0$ represents the LRU position and $T_{rh}(B)$ is the total number of read hits in LLC bank B. We bound this fraction by a given design parameter τ_R. Therefore, the desired width of the in-LLC write buffer in bank B is the largest n such that $\sum_{i=0}^{n-1} RHH(B,i) \leq \tau_R.T_{rh}(B)$. We refer to this width computed periodically based on RHH as $n_R(B)$. Every time $n_R(B)$ is computed, the RHH and T_{rh} counters of bank B are halved. In general, τ_R should be small. We set τ_R to 1/16.

The aforementioned clean LRU victim may have a dirty copy in the inner levels of the cache hierarchy. In an inclusive LLC, the dirty copy must be evicted to maintain inclusion. The probability of such dirty inclusion victims (DIVs) must be minimized because they lead to dirty LLC victims. If the clean victim stayed longer in the LLC, it would have experienced a write hit because the dirty copy would have been evicted from the inner levels with high likelihood. Therefore, the DIV probability can be bounded by restricting n such that the fraction of write hits experienced by LLC blocks in the last n LRU stack positions is bounded by a design parameter τ_W. Let this new bound on n be $n_D(B)$. To estimate the number of write hits, we maintain a write hit histogram (WHH) array in each LLC bank (Figure 7). If LLC bank B observes a total of $T_{wh}(B)$ write hits, $n_D(B)$ is the maximum value of n such that $\sum_{i=0}^{n-1} WHH(B,i) \leq \tau_W.T_{wh}(B)$, where $i=0$ is LRU. Now, the probability of DIV (P_{DIV}) is bounded by the product of τ_W and the capacity ratio of the inner cache level(s) to the LLC (roughly 1/4 to 1/8 in our configurations). We set τ_W to 1/2 for a P_{DIV} bound of 1/8. In an LLC bank, the WHH and T_{wh} counters are halved along with the RHH and T_{rh} counters. Finally, the dynamic width of the write buffer in LLC bank B is set to $n(B) = \max(3, \min(n_R(B), n_D(B)))$ so that it is at least three ways (decided empirically for a 16-way LLC).

Figure 8 shows the steps involved in updating the RHH and WHH arrays. Figure 9 shows how $n_R(B)$ is computed iteratively. The RHH index register is initially zero and it contains $n_R(B)$ when the comparator outputs zero at the end of the iterations. A similar piece of

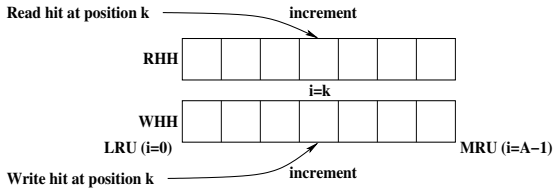

Fig. 7. The read and write hit histograms.

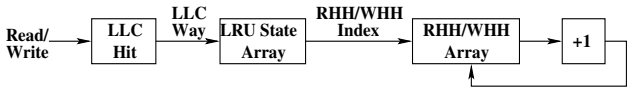

Fig. 8. Logic for updating RHH/WHH array on an LLC hit.

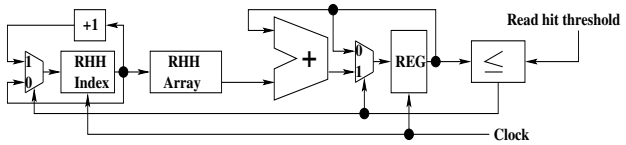

Fig. 9. Logic for computing $n_R(B)$. The read hit threshold is set to $T_{rh}(B) >> \log_2(1/\tau_R)$, where $1/\tau_R$ is always a positive power of two.

logic can be used for computing $n_D(B)$ in parallel with $n_R(B)$. We, however, note that the logic for computing $n_D(B)$ is unnecessary in non-inclusive and exclusive LLCs because LLC victims in such configurations do not generate inclusion victims.

C. Adaptive Write Draining Algorithm

In this section, we discuss how the dirty data are periodically scrubbed out of the LLC so that the in-LLC write buffer width $n(B)$, as computed above, is maintained.

1) Opportunistic Minor Write Draining: When a DRAM channel has no pending reads and the channel's write buffer occupancy is below a fractional threshold τ_{opp} (i.e., number of pending writes $< \tau_{opp} \times$number of write buffer entries per channel), the channel controller enters the minor write draining mode by selecting an LLC bank and requesting it to send dirty data to the channel. The LLC bank selection is done through a *turn* register per channel which records the id of the next round-robin LLC bank to send dirty data to the channel. During the minor draining mode, the LLC bank sends writes to only the requesting channel and attempts to keep the channel's write buffer only partially filled up to a fraction τ_{opp}. This lowers the chance of accidentally entering a major write draining mode by filling up the write buffer due to a burst of writes from the selected LLC bank. Every write acknowledgment coming back to the LLC bank carries a bit indicating whether the target channel's write buffer occupancy has crossed τ_{opp}. We set τ_{opp} to 1/2. The *turn* register is updated when a read arrives on the channel terminating the minor draining mode.

2) Major Write Draining: Most of the dirty data are scrubbed through the major write draining mode. Each LLC bank B can independently force a DRAM channel to enter this mode by sending a large volume of writes to the channel. The maximum number of dirty blocks in the write buffer of LLC bank B is $N.n(B)$ where N is the number of sets in the bank and $n(B)$ is the number of ways allocated to the write buffer. An LLC set is said to be full if all its lowest $n(B)$ LRU stack positions are filled with dirty blocks. An LLC bank B triggers a major draining mode if (i) the bank's write buffer has at least $\tau_{HWM}.N.n(B)$ dirty blocks where $\tau_{HWM} \leq 1$ is a design parameter defining a high watermark, or (ii) the number of full sets in the bank is at least $N.\tau_F$ and the bank's write buffer has at least $\tau_{LB}.N.n(B)$ dirty blocks ($\tau_F \leq 1$, $\tau_{LB} \leq 1$ are design parameters). Rule (i) triggers on reaching a high watermark, while Rule (ii) triggers when the number of full sets reaches a threshold in the bank. Rule (ii) is necessary because the clean LRU policy can perform very poorly in the presence of too many full sets. In Rule (ii), the parameter τ_{LB} is chosen such that $\tau_{LB}.N.n(B)$ is at least N (one way worth blocks) offering enough flexibility to the scrubber in the choice of scrub candidates. Since minimum $n(B)$ is three, we set τ_{LB} to 3/8 (less complex to implement than 1/3). We set τ_{HWM} to 3/4 (which is double of τ_{LB}). A workload with a high LLC read hit count per fill can trade some LLC read hits for extra write buffering and hence, can have a large τ_F. Let F_B be the fill counter and $T_{rh}(B)$ be the read hit counter in LLC bank B. Table I lists the values of τ_F used by our design. Each table entry is seven-bit wide (numerator: 3 bits, log of denominator: 4 bits).

TABLE I
LOOK-UP TABLE FOR DECIDING τ_F

Range of $T_{rh}(B)/F_B$	τ_F	Range of $T_{rh}(B)/F_B$	τ_F
[0, 0.1)	1/256	[0.4, 0.5)	3/8
[0.1, 0.2)	1/128	[0.5, 0.7)	4/8
[0.2, 0.3)	1/8	[0.7, 1.0)	5/8
[0.3, 0.4)	2/8	$\geq$ 1.0	6/8

Having decided the trigger rules for the write draining modes, we now turn to discuss the process of scrubbing dirty LLC blocks. The scrubber takes two passes over the sets in an LLC bank during a major draining mode and in each pass selects at most one scrub candidate from each set. In each idle cycle of the LLC bank when there is no pending read or write requests, the scrubber initiates a pipelined lookup to a set. To simplify the control over write bandwidth demand to a channel, all scrub candidates from an LLC bank map to the same channel. To improve bandwidth coordination, after sending out every 32 scrubbed writes (size of per-channel write buffer), the scrubber checks whether the number of pending reads in the target channel has reached a threshold K_r and if yes, the major draining mode terminates. K_r is set to the number of miss status holding registers (MSHRs) per LLC bank times the number of LLC banks that can send requests to a channel.

Each major write draining mode in an LLC bank B marks the beginning of a new epoch. Each epoch has the following three phases: (i) recomputation of the write buffer width $n(B)$ followed by halving of RHH, WHH, $T_{rh}(B)$, $T_{wh}(B)$, and F_B counters, (ii) scrubbing operations of the major write draining mode, (iii) large read stretches interleaved with minor opportunistic write stretches.

3) Efficient Selection of Scrub Candidates: We confine the search for a scrub candidate within a set to the lowest portion of the LRU stack so that premature scrubs are minimized. More specifically, let $n_{scrub}(B)$ be the maximum value of k such that $\sum_{i=0}^{k-1} WHH(B,i) \leq \tau_{scrub}.T_{wh}(B)$, where $T_{wh}(B)$ is the total number of write hits in LLC bank B and $\tau_{scrub} \leq 1$ is a design parameter. By confining the scrub candidates to the lowest $n_{scrub}(B)$ ways of the LRU stack, we bound the probability that a scrubbed block receives a write hit (meaning premature scrub) by τ_{scrub}. We set τ_{scrub} to 1/32. A logic similar to the one in Figure 9 computes $n_{scrub}(B)$ when an epoch starts. Minimum $n_{scrub}(B)$ is set to three to offer enough flexibility to the scrubber.

We devise a set traversal order for the scrubber to improve DRAM locality and parallelism among the scrub candidates. The 1024 sets in an LLC bank are divided into partitions $p0, \ldots, p7$ each having 128 consecutive sets. Each partition is divided into 32 segments $s0, \ldots, s31$ each having four consecutive sets ($set0, \ldots, set3$). The scrubber traverses the sets of an LLC bank in the following order (one

set at a time in an idle cycle): ($p0$, $s0$, $set0$), ..., ($p0$, $s0$, $set3$), ($p1$, $s0$, $set0$), ..., ($p1$, $s0$, $set3$), ..., ($p7$, $s0$, $set0$), ..., ($p7$, $s0$, $set3$), ($p0$, $s1$, $set0$), ..., ($p0$, $s1$, $set3$), and so on. During a minor draining mode, the traversal starts from where it left off last time. During a major draining mode, the traversal starts from ($p0$, $s0$, $set0$) and runs through the traversal order twice unless terminated early due to read bandwidth demand on target channel. As the scrubber moves from one partition to another, the chance of bank- and rank-parallelism increases (please refer to Figure 1). Within the four sets of a segment of a particular partition, the scrubber tries to improve row locality using a simple greedy row matching algorithm, in which it looks for a scrub candidate in the currently visited set having the same (rank, bank, row) tuple as the last scrub candidate. If no such candidate is found within the last $n_{scrub}(B)$ LRU positions, it looks for a candidate having DRAM bank or rank different from the last scrub candidate to improve bank-/rank-parallelism. If unsuccessful, it picks the LRU dirty block (if any) from the last $n_{scrub}(B)$ LRU positions. Our proposed scrubber walk obviates the need for an auxiliary structure such as the set state vector (SSV) [64] or organizing the dirty bits separately as in the dirty block index (DBI) [60]. In Section VII, we show that our scrubber walk is as effective as the SSV.

Table II summarizes all the tunable parameters of our design. The overall storage overhead of our proposal is slightly over 1Kbits per LLC bank arising from the RHH and WHH arrays (32 bits $\times$ 16 ways each), the T_{rh}, T_{wh}, and F_B counters (32 bits each), the full set counter (10 bits), the dirty block counter for the in-LLC write buffer (14 bits), and the look-up table for τ_F (8 entries $\times$ 7 bits). The logic overhead is discussed in Section IV-E.

TABLE II
TUNABLE DESIGN PARAMETERS

Parameter	Section	Value
τ_R, τ_W, $\min n(B)$	IV-B	1/16, 1/2, 3
τ_{opp}	IV-C1	1/2
τ_{LB}, τ_{HWM}, K_r	IV-C2	3/8, 3/4, 64
τ_F	IV-C2	Table I
τ_{scrub}, $\min n_{scrub}(B)$	IV-C3	1/32, 3
(#partitions, #segments)	IV-C3	(8, 32)

D. Application to Non-LRU LLC Policies

We now extend our proposal to not-recently-used (NRU), static and dynamic re-reference interval prediction (SRRIP and DRRIP) [21], and signature-based hit prediction (SHiP) [76] policies.

The NRU policy maintains a reference bit with each block. This bit is set to one on an access to a block. When all blocks in a set have this bit equal to one, all but the one accessed most recently are reset to zero. The replacement policy victimizes the block in a set that has its reference bit reset. A tie is broken by evicting the victim candidate with the lowest physical way id. Figure 10 illustrates how our proposal deterministically orders the ways from MRU to LRU based on reference bit and way id. The clean NRU policy exercised by our proposal invokes the baseline NRU policy in the absence of clean blocks in the target set; otherwise it selects the clean block closest to the LRU end in the order shown in Figure 10. If an evicted clean block has reference bit equal to one, the reference bits of the remaining blocks in the set are made zero to protect the newly filled clean block.

The SRRIP policy attaches a two-bit re-reference prediction value (RRPV) with each block. A newly filled block is assigned RRPV 2. On a hit, the block's RRPV is upgraded to zero. The replacement policy increments the RRPVs of all blocks in the target set until a block is found with RRPV 3. A tie is broken by evicting the victim candidate with the lowest physical way id. Figure 10 shows how our proposal deterministically orders the ways from MRU to LRU based on RRPV and way id. The clean SRRIP policy invokes the baseline SRRIP policy if there is no clean block in the target set; otherwise it increments the RRPVs of all blocks in the target set until a *clean* block is found with RRPV 3. The thread-aware DRRIP (TADRRIP) policy dynamically chooses between insertion RRPVs of 2 and 3 on a per-thread basis.

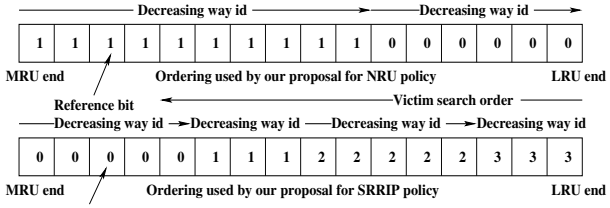

Fig. 10. Way ordering for NRU and SRRIP.

The SHiP policy uses two-bit RRPV per block and employs the correlation between program counter (PC) of an LLC fill and reuse behavior of the filled block to decide the RRPV of a newly filled block. For use in SHiP, a prefetch fill inherits the PC of the trigger demand access. We implement SRRIP+SHiP, which employs a set-duel to choose the best of SHiP and SRRIP dynamically. The clean replacement policy is same as clean SRRIP.

E. Design Synthesis and Area Estimates

We synthesize three logic blocks needed by our proposal in each LLC bank: updation of the RHH/WHH array (Figure 8), computation of n_R, n_D, n_{scrub} (Figure 9), and the scrubber logic. We use the 45 nm TSMC technology library and assume a 16-way LLC. A three-stage pipelined implementation of the logic after LLC hit detection shown in Figure 8 meets the target 4 GHz frequency and has an area of 0.00453 mm^2. A fully combinational implementation of the logic in Figure 9 requires two cycles at 4 GHz to complete one iteration and consumes 0.00307 mm^2 area. On average, n_R computation requires six iterations. This latency is off the critical path and observed only when n_R is recomputed. The scrubber logic in each LLC bank implements the greedy row matching algorithm discussed in Section IV-C3. A fully combinational implementation of this logic meets the cycle time of the target 4 GHz clock and consumes 0.00244 mm^2 area. The total area overhead of one instance of the logic in Figure 8, three instances (for $n_R(B)$, $n_D(B)$, $n_{scrub}(B)$) of the logic in Figure 9, and one instance of the scrubber logic is only 0.01618 mm^2 for an LLC bank B.

The LRU ordering is not explicitly stored for the non-LRU policies (unlike LRU). Updating the appropriate location of the RHH/WHH array on an LLC hit in these cases uses the logic shown in Figure 11 (shown for NRU; support for other policies is similar) in place of the logic in Figures 8. We synthesize this logic as a three-stage pipeline for meeting the target 4 GHz frequency. The first stage uses the combinational functions `f0` to `f15` to compute the indices `SI[i]` of the LLC ways in the shuffled array of `ref` bits reflecting the MRU to LRU ordering as shown in Figure 10. `SI[i]` is the index of LLC way `i` in the shuffled array. The LRU end has index zero and the MRU end has index 15. If `ref[i]` is zero, `SI[i]` would be computed as `(i-j)` if there are `j` bits set to one among `ref[0]` to `ref[i-1]`. If `ref[i]` is one, `SI[i]` would be computed as `(i+j)` if there are `j` zero bits among `ref[i+1]` to `ref[15]`. The functions `f0` to `f15` employ small lookup tables. The second stage (i) selects `SI[way]` as the RHH/WHH index where `way` is the way id of LLC hit and (ii) reads the value `v` at that index of RHH/WHH.

The third stage increments `v` and writes it to the RHH/WHH index location (this stage requires a bypass path). The entire pipeline consumes only 0.00504 mm^2 area including the RHH/WHH array. The logic necessary for supporting two-bit RRIP policies employs similar design techniques. For example, `SI[i]` is `(i-j+k)` if there are `j` RRPVs bigger than `RRPV[i]` among `RRPV[0]` to `RRPV[i-1]` and there are `k` RRPVs smaller than `RRPV[i]` among `RRPV[i+1]` to `RRPV[A-1]` where `A` (=16) is the associativity. A lookup table-based four-stage pipelined implementation clocked at 4 GHz consumes only 0.00938 mm^2 area.

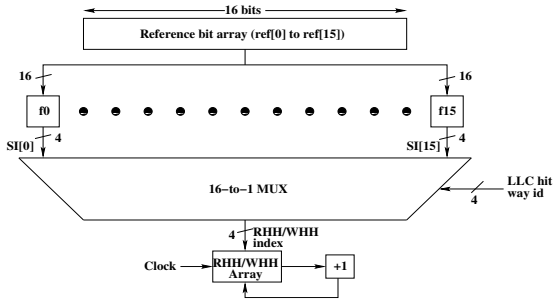

Fig. 11. RHH/WHH update logic for NRU policy.

V. Related Work

The existing LLC policy proposals optimize the number of read misses [2]–[5], [7]–[9], [12], [15], [20]–[24], [26]–[33], [38]–[40], [43], [49]–[54], [58], [59], [61], [68]–[70], [72], [76], [77]. DRAM rank-aware LLC policies have been proposed for optimizing DRAM energy [1]. The clean LRU policy and policies with higher replacement priority for clean blocks have been used in the LLC to reduce the volume of writes to the next-level non-volatile memory [13], [14], [79]. The clean LRU policy has also been used in non-volatile caches to improve intra-set wear-leveling [25] and to improve page cache replacement in systems with FLASH memory [48]. The enhanced second-chance page replacement algorithm also prioritizes clean page eviction [63]. LLC policies that offer additional protection to dirty blocks with write reuses have been explored with the goal of reducing write traffic to the next-level non-volatile memory [74], [78]. LLC policies to balance write traffic to the write queues of non-volatile memory [81] and LLC writeback-aware techniques to partition non-volatile memory bandwidth among applications [82] have been explored. Although our proposal relies on the clean LRU policy in the LLC, the primary novelty of our proposal arises from the mechanisms that maximize the DRAM read bandwidth stretches while controlling write bandwidth.

Majority of the DRAM access scheduling proposals focus on read scheduling [10], [11], [16], [18], [19], [34], [35], [42], [44]–[47], [55], [56], [65]–[67]. These proposals drain writes based on high and low watermarks on the channels' write buffer occupancy. Among the proposals on write scheduling, eager writeback generates writes from the LLC whenever some DRAM resource falls idle [37]. Since the write scheduler's visibility is limited to the small write buffers of the memory controllers, write locality is poor as in the baseline. A subsequent proposal has employed prediction mechanisms to identify and use large idle periods of a rank for sending writes from the LLC [75]. Eager writeback in the context of non-volatile memory has also been explored [80].

Virtual write queue (VWQ) expands the visibility of the write scheduler by using the last two ways of the LRU stack of the LLC as a statically sized write buffer [64]. It cleans the dirty blocks from the write buffer based on static pre-defined high and low watermarks or during idle periods of DRAM resources. It takes help of a structure called set state vector (SSV) to improve write locality. However, VWQ continues to use LRU replacement policy offering no guarantee of long stretches of exclusive DRAM read bandwidth. Our proposal also employs an in-LLC write buffer, but sizes this buffer dynamically based on run-time behavior. Dynamic sizing of this buffer enables maximization of the DRAM read stretches. DRAM-aware writeback (DAWB) proposal writes back dirty LLC blocks in row-local groups improving write locality while employing eager writeback and opportunistic scheduling [36]. Dirty block index (DBI) further eases the search for row-local groups of dirty blocks and can be combined with aggressive writeback (AWB) that employs eager and opportunistic write scheduling [60]. Last write prediction-guided (LWPG) writeback proposal removes the reliance of VWQ on LRU policy and employs a last-write predictor to identify the LLC blocks that are eligible to be written back to DRAM [73]. In Section VII, we present a quantitative comparison of our proposal against VWQ, DAWB, LWPG writeback, and DBI+AWB proposals for different LLC policies. The MRU to LRU ordering technique discussed in Section IV-D is used to expand the scope of VWQ to the non-LRU policies.

VI. Simulation Environment

We use the Multi2Sim infrastructure [71] to model eight dynamically scheduled out-of-order-issue x86 cores clocked at 4 GHz. Each core has private L1 instruction and data caches, a unified L2 cache, and a highly tuned prefetcher. Each L1 cache is 32 KB 8-way with two-cycle access latency. The L2 cache is 256 KB 8-way with five-cycle access latency. The shared LLC is 8 MB 16-way associative and 8-way banked with seven-cycle bank access latency. We also consider a 16 MB 16-way LLC with 8 banks and eight-cycle bank access latency. Each LLC bank has 16 MSHRs supporting 16 outstanding read misses from a bank. An LLC bank's local write buffer has 32 entries.

The dual-channel DDR3-1600 11-11-11-28 DRAM system is modeled using DRAMSim2 [57]. The DRAM system has 64-bit channels, burst length of 8, two ranks per channel, 8 banks per rank, and uses x8 chips with 1 KB row buffer per bank per chip. Each channel controller has a 32-entry write buffer.

The fifty eight-way multiprogrammed workloads used in this study are prepared by mixing fifteen SPEC CPU 2006 applications that were shown in Figure 2. These fifteen applications are chosen to represent a wide range of DRAM reads per kilo instructions (2.2 to 35.8) and DRAM writes per kilo instructions (1.2 to 18.5) in the eight-way rate mode. Out of the fifty workloads, fifteen are homogeneous, each of which has eight copies of the same application. The remaining 35 heterogeneous workloads are prepared by evenly mixing all the fifteen applications. Each application in a mix retires 500M representative dynamic instructions chosen using SimPoint [62]. Early finishing applications continue to run until all applications retire the representative segment of instructions. Only the representative segment is used for reporting speedup.

VII. Simulation Results

We first analyze the performance of our proposal in detail on the fifteen homogeneous workload mixes (Section VII-A). Next, we discuss the results for all the fifty mixes (Section VII-B).

A. Performance of Homogeneous Mixes

Figure 12 quantifies the speedup of our bandwidth-aware LLC (BALLC) policy proposal relative to the baseline. For each

workload, the leftmost bar shows the speedup of our proposal. The speedup ranges from 2% (libquantum and wrf) to 48% (astar.rivers) with an average of 11%. Several workloads enjoy more than 10% speedup: lbm (22%), mcf (12%), milc (12%), omnetpp (11%), soplex.pds-50 (14%), and sphinx3 (18%). For each workload, the middle bar shows the effect of replacing our scrubber walk order (Section IV-C3) by the SSV [64]. Our proposed scrubber walk order achieves nearly the same performance as the SSV. The rightmost bar shows the effect of replacing clean LRU by the baseline LRU policy in our proposal; the average speedup declines to only 3%. This large loss in performance arises from the disturbance caused by the dirty LLC victims. Subsequent analyses will discuss this further. These results clearly bring out the sources of performance improvement in BALLC: the clean LRU policy enabled by adaptive in-LLC write buffering and intelligent periodic scrubbing of LLC dirty blocks. The end-result is long uninterrupted stretches of DRAM reads leading to significantly accelerated execution.

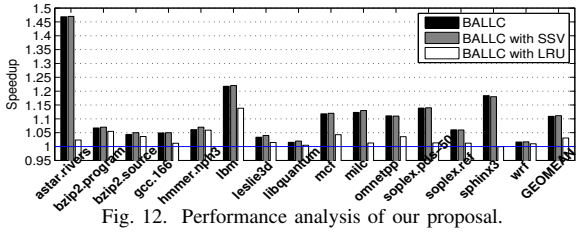

Fig. 12. Performance analysis of our proposal.

Figure 13 presents the DRAM read and write counts of our proposal normalized to the baseline. These counts remain unchanged for almost all workloads. Therefore, our proposal does not generate any extra LLC misses and nor does it scrub prematurely. This is expected because the LLC capacity lost to write buffering and the scrub width are both bounded analytically. Interestingly, astar.rivers and sphinx3 enjoy large savings in reads and writes. These two workloads benefit from significant volumes of read and write reuses of dirty blocks captured by our in-LLC write buffer. Such behavior has been exploited in LLC optimization studies [28]. On top of each group of bars, we show the percentage of DRAM writes that arise from dirty LLC victims or dirty inclusion victims in our proposal. Ideally, this percentage should be zero because our proposal uses the clean LRU policy. However, occasionally it may victimize a dirty block due to absence of clean blocks in the target LLC set or presence of a dirty inclusion victim. At most 11% and on average only 3% of DRAM writes arise due to such circumstances establishing our proposal's success in controlling DRAM write bandwidth.

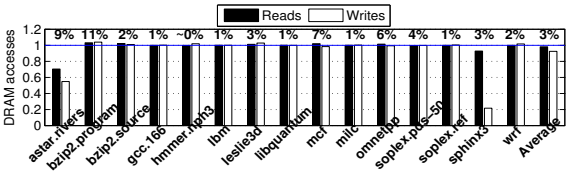

Fig. 13. Volume of DRAM reads and writes.

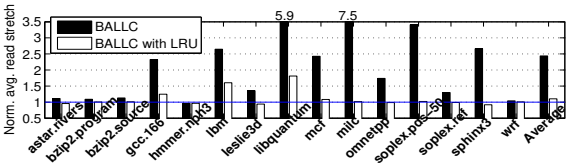

Fig. 14. Normalized read stretch length.

The primary goal of our proposal is to offer long uninterrupted DRAM read stretches. Figure 14 shows the average length of the DRAM read stretches normalized to the baseline for our proposal (BALLC) and when our proposal is executed with the baseline LRU policy (BALLC with LRU) as opposed to the clean LRU policy. The length of a DRAM read stretch is measured as the number of reads performed between two consecutive write stretches. Our proposal improves the average read stretch length by about 2.4× (144%). Interestingly, for libquantum, the average read stretch length improves by 5.9×, but performance improvement is only 2% (Figure 12). For libquantum, the read bandwidth demand is very high throughout the execution. Our proposal offers long read stretches and delays reads when doing major write draining. This is only slightly better than regular relatively shorter read and write stretches of the baseline for libquantum. BALLC executed with LRU policy improves the average read stretch length by only 10% establishing the importance of clean LRU.

In Figure 2 of Section III, we showed an 11% improvement by eliminating the penalty of channel turn-arounds and precharge/activate operations for writes in the baseline. Figure 15 shows the potential speedup when our proposal is executed with zero channel turn-around (NoTA) and zero precharge/activate penalty for writes (NoWPRE_ACT). Elimination of turn-arounds improves the average speedup of our proposal from 11% to 12%. Removal of precharge and activate penalty during write draining further improves the average speedup to 14%. Sphinx3 loses performance compared to BALLC as the overheads are eliminated. This application benefits from hits to the retained LLC dirty blocks in BALLC. However, when the turn-around and precharge/activate penalty for writes is nullified, writes complete faster and LLC residency of dirty blocks decreases eliminating some of the LLC hits enjoyed by BALLC. Across the board, overall gains are marginal indicating that our proposal is near-ideal in terms of channel turn-around and precharge/activate overheads of writes.

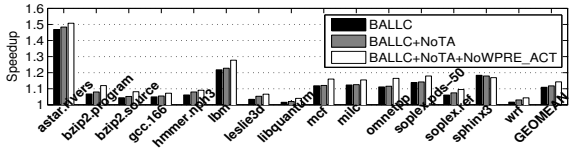

Fig. 15. Influence of channel turn-arounds and write locality in our proposal.

Sensitivity to LLC Capacity. All the results presented so far assume an 8 MB LLC. Figure 16 shows speedup on a 16 MB LLC normalized to the baseline with 8 MB LLC. The left bar shows that the baseline, on average, improves by 15%. Due to increased LLC access latency, lbm suffers from a small performance loss. The right bar shows that our proposal improves performance by 23%, on average (i.e., 8% average speedup over the 16 MB baseline).

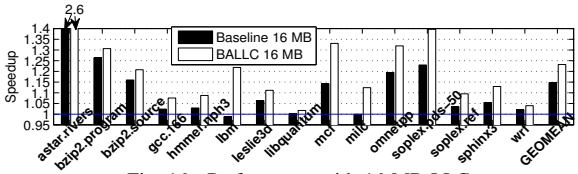

Fig. 16. Performance with 16 MB LLC.

Sensitivity to DRAM Bandwidth. Figure 17 evaluates our proposal on a DDR3-2133 DRAM system. The speedup figures are relative to the baseline with DDR3-1600 DRAM. The DRAM latencies of the two configurations are nearly same. The leftmost bar shows that the baseline improves by 17% on average when moving from DDR3-1600 to DDR3-2133. The middle bar shows that our proposal improves by 30% on average. The rightmost bar reproduces the performance of our proposal with DDR3-1600.

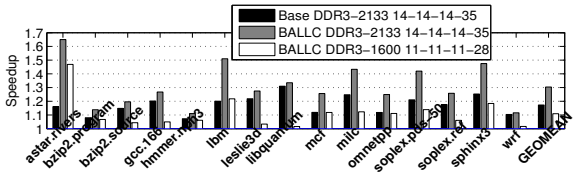

Fig. 17. Impact of DDR3-2133 DRAM.

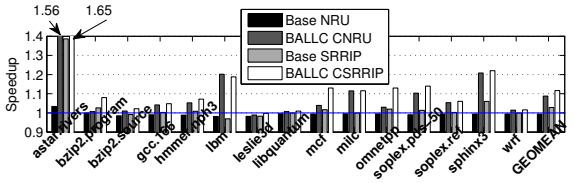

Fig. 18. Performance with an 8 MB LLC exercising NRU and SRRIP policies.

Application to Non-LRU Policies. Figure 18 quantifies the performance of our proposal for an 8 MB LLC exercising NRU or SRRIP policy normalized to the baseline 8 MB LLC with LRU policy. On average, the baseline loses performance by 1% when switching to NRU (Baseline NRU bar) and improves by 3% when switching to SRRIP (Baseline SRRIP bar). Our proposal exercising the clean NRU policy enjoys an average speedup of 9% (BALLC CNRU bar). With clean SRRIP, our proposal experiences a 12% speedup on average (BALLC CSRRIP bar).

Our evaluation with the TADRRIP and SRRIP+SHiP policies shows that the baseline enjoys an average 4.3% and 6% improvement, respectively. Our proposal achieves an average 12.4% and 13% improvement in performance when using clean TADRRIP and clean SRRIP+SHiP, respectively.

B. Performance of All Mixes

Figure 19 shows the speedup of our proposal on an 8 MB LLC relative to the baseline. The homogeneous and heterogeneous mixes are sorted by speedup. Averaged over fifty workload mixes, our proposal enjoys a speedup of 12%. Our proposal makes use of about five LLC ways (equivalent to 40K blocks or 2.5 MB) as the write buffer capacity averaged across all epochs of all the LLC banks and all the fifty mixes. In our proposal, the average DRAM read latency (from the arrival of a read request at the memory controller to the departure of the fill) decreases by 17% and DRAM write draining throughput improves by 50% on average compared to the baseline. The row hit rate for DRAM writes improves from 35% in the baseline to 40% in our proposal, averaged over the fifty mixes. The improvements in write throughput and write row hit rate stem from our proposal's intelligent scrub walk algorithm. Uninterrupted read stretches reduce waiting time and latency of the reads.

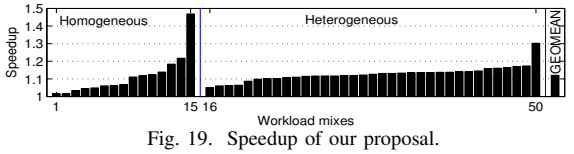

Fig. 19. Speedup of our proposal.

Comparison to Related Proposals. Figure 20 compares our proposal against VWQ [64], DAWB [36], LWPG writeback [73], and DBI+AWB [60]. These proposals were discussed in Section V. Each group of bars shows the speedup comparison (averaged over fifty mixes) for a different LLC policy on an 8 MB LLC. Our proposal uses the clean versions of these policies. All speedup figures are relative to the baseline 8 MB LLC using the LRU policy. For each LLC policy, VWQ, DAWB, LWPG writeback, and DBI+AWB

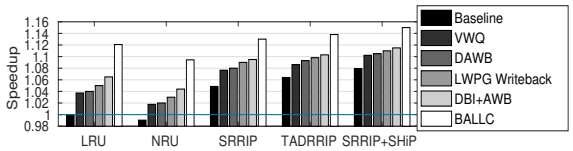

Fig. 20. Comparison to related proposals.

enjoy gradual improvement in performance. However, our proposal clearly stands out by a sizable margin of additional performance. With the SRRIP+SHiP policy, which is the best-performing LLC policy among the ones evaluated, the baseline speeds up by 8%. VWQ, DAWB, LWPG writeback, and DBI+AWB achieve 10%, 10.5%, 11%, and 11.5% speedup respectively, while our proposal enjoys a 15% speedup. The primary benefit of our proposal stems from long read stretches enabled by clean LRU policy and run-time adaptive write buffering in LLC. VWQ is the only proposal among the evaluated ones that uses an in-LLC write buffer (statically sized though). We quantify its difference with our proposal as follows: our proposal operating with the LRU policy (as opposed to clean LRU) performs close to VWQ with LRU policy indicating that the gains of our proposal come from the clean LRU policy and the associated enabling mechanisms of in-LLC adaptive write buffering.

Approaching Unbounded Write Buffer Performance. Figure 21 evaluates the baseline and our proposal on 8 MB LLC configurations with different sizes of DRAM channel's write buffer. The speedup figures are averaged over fifty mixes and are relative to the baseline with 32-entry write buffer per channel. The leftmost four bars show the baseline performance for 1024-entry, 2048-entry, 4096-entry, and infinite write buffer per channel. The configuration with infinite write buffer is same as Unbounded_WnoR+DrainAtEnd discussed in Section III. The next three bars show the speedup of our proposal with 32-entry, 8-entry, and 4-entry write buffer per channel. We make three important observations. First, our proposal with a 32-entry write buffer delivers the same performance as the baseline with a 1024-entry write buffer. Second, the unbounded write buffer speedup is 16%, while our proposal comes close achieving a speedup of 12%. Third, our proposal with an 8-entry write buffer outperforms the baseline with a 32-entry write buffer opening up opportunities to reduce the complexity of the write buffer design.

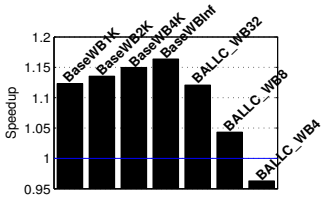

Fig. 21. Impact of DRAM write buffer size.

VIII. SUMMARY

We have presented a bandwidth-aware LLC policy that intelligently controls the DRAM read and write bandwidth demands. The proposed policy offers long stretches of exclusive DRAM bandwidth to reads by forcing the LLC replacement decisions not to evict dirty blocks. This is enabled by a dynamically computed population bound on LLC dirty blocks within the lower portion of the LRU stack. When this population is reached, the LLC scrubs out a certain fraction of dirty blocks following a DRAM-aware scrub walk of the LLC sets to improve locality, bank-parallelism, and rank-parallelism of DRAM writes. Averaged over fifty eight-way multiprogrammed workloads, the bandwidth-aware LLC policy proposal improves performance by 12%. This performance comes close to the performance of a configuration that has unbounded DRAM write buffers.

REFERENCES

[1] A. M. Amin and Z. Chishti. Rank-aware Cache Replacement and Write Buffering to Improve DRAM Energy Efficiency. In *ISLPED* 2010.

[2] A. Arunkumar and C.-J. Wu. ReMAP: Reuse and Memory Access Cost-aware Eviction Policy for Last-level Cache Management. In *ICCD* 2014.

[3] A. Basu, et al. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *MICRO* 2007.

[4] N. Beckmann and D. Sanchez. Maximizing Cache Performance Under Uncertainty. In *HPCA* 2017.

[5] N. Beckmann and D. Sanchez. Talus: A Simple Way to Remove Cliffs in Cache Performance. In *HPCA* 2015.

[6] N. Chatterjee, et al. Staged Reads: Mitigating the impact of DRAM writes on DRAM reads. In *HPCA* 2012.

[7] M. Chaudhuri, et al. Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches. In *PACT* 2012.

[8] M. Chaudhuri. Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches. In *MICRO* 2009.

[9] N. Doung, et al. Improving Cache Management Policies Using Dynamic Reuse Distances. In *MICRO* 2012.

[10] E. Ebrahimi, et al. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *ASPLOS* 2010.

[11] E. Ebrahimi, et al. Parallel Application Memory Scheduling. In *MICRO* 2011.

[12] P. Faldu and B. Grot. Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches. In *PACT* 2017.

[13] V. V. Fedorov, et al. ARI: Adaptive LLC-memory Traffic Management. In *ACM TACO*, **10**(4), Article 46, 2013.

[14] A. P. Ferreira, et al. Increasing PCM Main Memory Lifetime. In *DATE* 2010.

[15] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In *ISCA* 2011.

[16] S. Ghose, H. Lee, and J. F. Martinez. Improving Memory Scheduling via Processor-side Load Criticality Information. In *ISCA* 2013.

[17] HP Labs. CACTI. Available at http://www.hpl.hp.com/research/cacti/.

[18] I. Hur and C. Lin. Memory Scheduling for Modern Microprocessors. In *ACM TOCS*, **25**(4): 10, 2007.

[19] E. Ipek, et al. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA* 2008.

[20] A. Jain and C. Lin. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *ISCA* 2016.

[21] A. Jaleel, et al. High Performance Cache Replacement using Re-reference Interval Prediction (RRIP). In *ISCA* 2010.

[22] A. Jaleel, et al. Adaptive Insertion Policies for Managing Shared Caches. In *PACT* 2008.

[23] D. A. Jimènez. Insertion and Promotion for Tree-based PseudoLRU Last-level Caches. In *MICRO* 2013.

[24] D. A. Jimènez and E. Teran. Multiperspective Reuse Prediction. In *MICRO* 2017.

[25] M. R. Jokar, M. Arjomand, and H. Sarbazi-Azad. Sequoia: A High-endurance NVM-based Cache Architecture. In *IEEE TVLSI*, **24**(3): 954–967, 2016.

[26] J. Kim, et al. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In *ASPLOS* 2017.

[27] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache Replacement Based on Reuse Distance Prediction. In *ICCD* 2007.

[28] S. M. Khan, et al. Improving Cache Performance by Exploiting Read-Write Disparity. In *HPCA* 2014.

[29] S. M. Khan, Z. Wang, and D. A. Jimènez. Decoupled Dynamic Cache Segmentation. In *HPCA* 2012.

[30] S. M. Khan, Y. Tian, and D. A. Jimènez. Dead Block Replacement and Bypass with a Sampling Predictor. In *MICRO* 2010.

[31] S. M. Khan and D. A. Jimènez. Insertion Policy Selection Using Decision Tree Analysis. In *ICCD* 2010.

[32] S. M. Khan, et al. Using Dead Blocks as a Virtual Victim Cache. In *PACT* 2010.

[33] M. Kharbutli and Y. Solihin. Counter-based Cache Replacement and Bypassing Algorithms. In *IEEE TC*, **57**(4): 433–447, 2008.

[34] Y. Kim, et al. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA* 2010.

[35] Y. Kim, et al. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO* 2010.

[36] C. J. Lee, et al. DRAM-aware Last-level Cache Writeback: Reducing Write-caused Interference in Memory System. *Technical Report TR-HPS-2010-002*, The University of Texas at Austin, 2010.

[37] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager Writeback – A Technique for Improving Bandwidth Utilization. In *MICRO* 2000.

[38] H. Liu, et al. Cache Bursts: A New Approach for Eliminating Dead Blocks and Increasing Cache Efficiency. In *MICRO* 2008.

[39] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic Shared Cache Management (PriSM). In *ISCA* 2012.

[40] R. Manikantan, K. Rajan, and R. Govindarajan. NUcache: An Efficient Multicore Cache Organization Based on Next-Use Distance. In *HPCA* 2011.

[41] Micron Technology Inc.. DDR3 SDRAM System-Power Calculator. Available at http://www.micron.com/∼/media/documents/products/power-calculator/ddr3_power_calc.xlsm.

[42] T. Moscibroda and O. Mutlu. Distributed Order Scheduling and its Application to Multi-core DRAM Controllers. In *PODC* 2008.

[43] A. Mukkara, N. Beckmann, and D. Sanchez. Whirlpool: Improving Dynamic Cache Management with Static Data Classification. In *ASPLOS* 2016.

[44] J. Mukundan and J. F. Martinez. MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler. In *HPCA* 2012.

[45] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO* 2007.

[46] O. Mutlu and T. Moscibroda. Parallelism-aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA* 2008.

[47] K. J. Nesbit, et al. Fair Queuing Memory Systems. In *MICRO* 2006.

[48] S-Y. Park, et al. CFLRU: A Replacement Algorithm for Flash Memory. In *CASES* 2006.

[49] M. K. Qureshi, et al. Adaptive Insertion Policies for High Performance Caching. In *ISCA* 2007.

[50] M. K. Qureshi, et al. A Case for MLP-Aware Cache Replacement. In *ISCA* 2006.

[51] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO* 2006.

[52] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *HPCA* 2007.

[53] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand-based Associativity via Global Replacement. In *ISCA* 2005.

[54] K. Rajan and R. Govindarajan. Emulating Optimal Replacement with a Shepherd Cache. In *MICRO* 2007.

[55] S. Rixner. Memory Controller Optimizations for Web Servers. In *MICRO* 2004.

[56] S. Rixner, et al. Memory Access Scheduling. In *ISCA* 2000.

[57] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *IEEE CAL*, **10**(1): 16–19, 2011.

[58] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *MICRO* 2010.

[59] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *ISCA* 2011.

[60] V. Seshadri, et al. The Dirty-Block Index. In *ISCA* 2014.

[61] V. Seshadri, et al. The Evicted-address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In *PACT* 2012.

[62] T. Sherwood, et al. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS* 2002.

[63] A. Silberschatz, P. B. Galvin, and G. Gagne. Operating System Concepts. 10th Edition, Wiley, 2018.

[64] J. Stuecheli, et al. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *ISCA* 2010.

[65] L. Subramanian, et al. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *ICCD* 2014.

[66] L. Subramanian, et al. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *MICRO* 2015.

[67] L. Subramanian, et al. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA* 2013.

[68] E. Teran, et al. Minimal Disturbance Placement and Promotion. In *HPCA* 2016.

[69] E. Teran, Z. Wang, and D. A. Jimènez. Perceptron Learning for Reuse Prediction. In *MICRO* 2016.

[70] Y. Tian, S. M. Khan, and D. A. Jimènez. Temporal-based Multilevel Correlating Inclusive Cache Replacement. In *ACM TACO*, **10**(4), article 33, 2013.
[71] R. Ubal, et al. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT* 2012.
[72] X. Wang, et al. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *HPCA* 2017.
[73] Z. Wang, S. M. Khan, and D. A. Jimènez. Improving Writeback Efficiency with Decoupled Last-write Prediction. In *ISCA* 2012.
[74] Z. Wang, et al. WADE: Writeback-aware Dynamic Cache Management for NVM-based Main Memory System. In *ACM TACO*, **10**(4), Article 51, 2013.
[75] Z. Wang, S. M. Khan, and D. A. Jimènez. Rank Idle Time Prediction Driven Last-level Cache Writeback. In *MSPC* 2012.
[76] C-J. Wu, et al. SHiP: Signature-Based Hit Predictor for High Performance Caching. In *MICRO* 2011.
[77] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *ISCA* 2009.
[78] X. Zhang, et al. A Read-Write Aware Replacement Policy for Phase Change Memory. In *APPT* 2011.
[79] D. Zhang, et al. Write-back-aware Shared Last-level Cache Management for Hybrid Main Memory. In *DAC*, Article No. 172, 2016.
[80] L. Zhang, et al. Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs. In *ISCA* 2016.
[81] M. Zhou, et al. Writeback-aware Partitioning and Replacement for Last-level Caches in Phase Change Main Memory Systems. In *ACM TACO*, **8**(4): 53:1-53:21, 2012.
[82] M. Zhou, et al. Writeback-aware Bandwidth Partitioning for Multi-core Systems with PCM. In *PACT* 2013.