2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Duplicon Cache: Mitigating Off-Chip Memory Bank and Bank Group Conflicts via Data Duplication

Ben (Ching-Pei) Lin*, Michael B. Healy†, Rustam Miftakhutdinov‡, Philip G. Emma † and Yale Patt*
* *University of Texas at Austin*
bencplin@utexas.edu, patt@ece.utexas.edu
† *IBM T.J. Watson Research Center*
mbhealy@us.ibm.com, pemma1113@gmail.com
‡ *Intel Corporation*
rustam.miftakhutdinov@intel.com

***Abstract*—Bank and bank group conflicts are major performance bottlenecks for memory intensive workloads. Idealized experiments show removing bank and bank group conflicts collectively can improve performance by up to 37.5% and by 22.5% on average for our mix of multi-programmed memory intensive workloads. We propose the Duplicon Cache to mitigate bank and bank group conflict penalties by duplicating select lines of data to an alternate bank group, giving the memory controller the freedom to source the data from the bank group which avoids conflicts. The Duplicon Cache is entirely implemented in the memory controller and does not require changes to commodity memory. We identify and address the main challenges associated with duplication: 1) tracking duplicated data efficiently, 2) identifying which data to duplicate, and 3) replacing stale duplicated data while protecting useful ones. Our evaluations show the Duplicon Cache configured with 128MB of storage (out of 16GB of main memory) improves performance by 8.3% while reducing energy by 5.6%.**

***Index Terms*—bank conflicts, bank group conflicts, duplication, set-associative cache, sectored cache, demand activates filtering, usefulness tracking, probabilistic replacement,**

I. Introduction

Main memory, such as Dynamic Random-Access Memory (DRAM), is organized into banks that allow independent memory requests to be serviced concurrently, increasing parallelism and performance. Conflicting requests that map to the the same bank, however, are serviced serially.

Such bank conflicts can be mitigated if conflicting requests can alternatively be serviced by another idle bank. This is possible if the data for the conflicting requests were previously duplicated to the other idle bank. Allowing data duplication across banks decreases the likelihood of bank conflicts at a cost of increased storage and coherence complexity. This tradeoff can be exploited to improve performance. To this end we propose Duplicon, which mitigates the effects of bank conflicts by duplicating select data in memory to an alternate bank. Duplicated data have lower access latencies on average, as they can be serviced by the alternate bank when the home bank is busy. By lowering the average access latency, Duplicon effectively acts as a cache of main memory, whereby a line is present in the Duplicon Cache if it has been duplicated.

We thank Intel Corporation and the Cockrell Foundation for their continued generous financial support of the HPS Research Group.

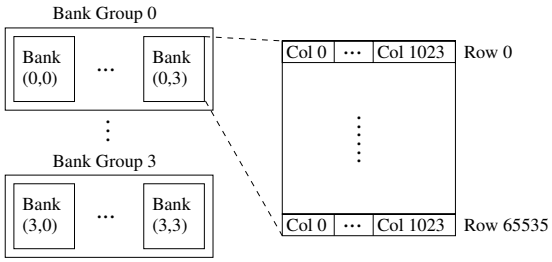

Fig. 1. Bank groups, banks, rows, and columns.

The Duplicon Cache is entirely implemented in the memory controller and does not require changes to commodity memory. Its main features are 1) a set-associative and sectored organization to allow efficient tracking of duplicated data, 2) Demand Activates Filtering to identify the right rows to duplicate, and 3) Usefulness Tracking and Probabilistic Replacement to allow stale duplicated data to be replaced while protecting useful duplicated data from being overwritten. Our evaluation shows Duplicon improves performance by 8.3% while reducing energy by 5.6% on average.

The rest of the paper is organized as follows: Section II provides background information on DRAM. Section III motivates how data duplication can mitigate bank and bank group conflicts. Section IV describes the key components of the Duplicon Cache. Section V describes the evaluation methodology and presents the results. Section VI discusses the related work, and Section VII summarizes the paper.

II. Background

A DRAM bank is divided into rows, with each row divided into columns. Only one row can be activated at any given time in each bank. Accesses to data in the already activated row are row buffer hits and have much lower latency. If one wishes to access data in a row different than the one currently activated, one must first Precharge the bank, then Activate the desired row. Once the desired row is activated, then Read and Write commands can be issued to desired columns within the activated row.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00031

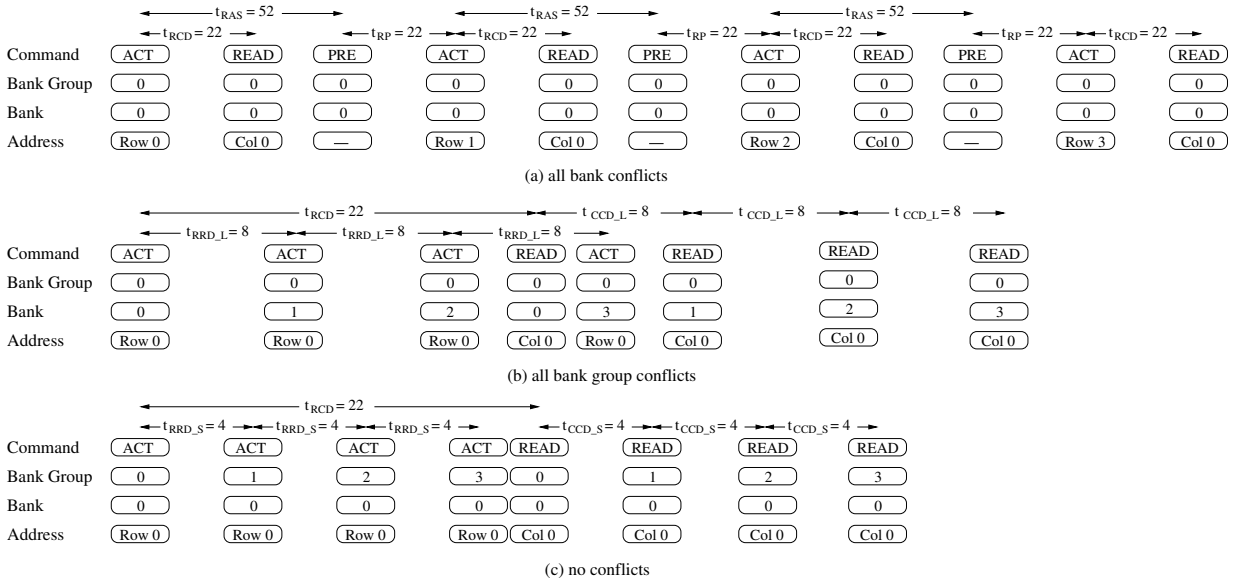

Fig. 2. Four concurrent memory requests to different rows in (a) the same bank (all bank conflicts), (b) different banks of the same bank group (all bank group conflicts), and (c) different bank groups (no conflicts); ACT = Activate, PRE = Precharge; figure is not to scale.

Circuit level limitations impose timing constraints on when Precharge, Activate, Read, and Write operations can be issued. When multiple memory requests map to different rows of data within the same bank, several long latency Precharge/Activate/Read(Write) operations must be performed serially to the same bank, degrading both memory latency and bandwidth. Such instances of requests mapping to different rows in the same bank are called bank conflicts.

Adding more banks reduces the likelihood of bank conflicts, but is costly. New memory architectures such as the fourth generation Double Data Rate (DDR4) DRAM reduce the cost of adding additional banks by partitioning banks hierarchically into bank groups, and imposing additional timing constraints on back-to-back operations to the same bank group. For example, additional delays are required for both back-to-back Read as well as back-to-back Activate operations to the same bank group. We use the term *bank group conflicts* to denote cases where memory requests to the same bank group incur additional delays that were otherwise not necessary had the requests been mapped to different bank groups.

Fig. 1 shows the organization of a DDR4 DRAM device into bank groups, banks, rows, and columns. We use the notation bank (m,n) to denote bank n in bank group m. There are 16 banks, partitioned into 4 bank groups. Each bank has 64K rows, and each row has 1K columns [1].

III. Motivation

A large system with many cores/threads sharing the memory system can have multiple outstanding memory requests to different rows at the same time. Fig. 2 shows how four concurrent memory requests to different rows will be serviced if the requests are to (a) the same bank (all bank conflicts), (b) different banks of the same bank group (all bank group conflicts), or (c) different bank groups (no conflicts). All banks are initially precharged.

In (a), the four requests are completely serialized. The service latencies are dominated by t_{RAS}, the minimum delay between an Activate(ACT) and a subsequent Precharge(PRE), and t_{RP}, the minimum delay between a Precharge and a subsequent Activate. In total, 244 DRAM cycles, or 488 processor cycles (assuming DDR4-3200 DRAM and 3.2GHz processor) were required to service all four requests.

In (b), since the requests are to different banks, the Activate operations are partially overlapped, but separated by t_{RRD_L} (i.e., long version of t_{RRD}), the minimum delay between Activates to the same bank group. Each Read operation can proceed t_{RCD} cycles after the appropriate row has been activated. The Reads are separated by t_{CCD_L} (i.e., long version of t_{CCD}), which is the minimum delay between back-to-back Reads (or back-to-back Writes) to the same bank group. In total 46 DRAM cycles (92 processor cycles) were required to service all four requests.

In (c) the requests are overlapped to the fullest extent possible. The Activates are now only separated by t_{RRD_S} (i.e., short version of t_{RRD}), the minimum delay between Activates to the different bank groups. Note this delay is halved compare to (b). Similarly the Reads are separated by t_{CCD_S}. In total 34 cycles (68 processor cycles) were required to service all four requests.

We devised a series of idealized experiments to measure the impact of bank and bank group conflicts on real workloads. The first experiment (i) approximates converting all bank conflicts into bank group conflicts by relaxing the bank

[1] these are the row/column dimensions for the 8Gb DDR4-3200 x8 DRAM device used in our evaluation; see Table I and [1] for details

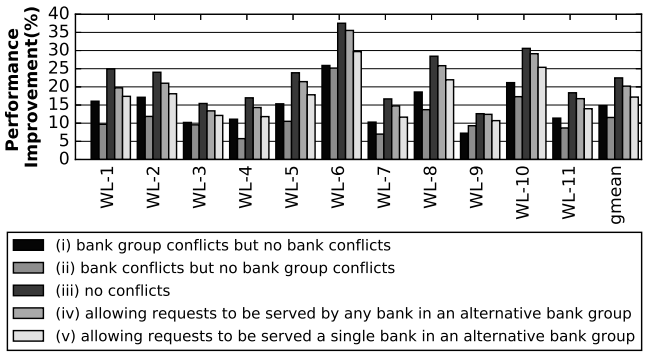

Fig. 3. Performance improvement when bank and/or bank group conflicts are removed or mitigated.

mapping within a bank group, allowing requests to be serviced by any bank in the same bank group: any request mapped to bank (m,n) can now alternatively be serviced by banks (m,0), (m,1), (m,2), or (m,3)[2], in essence converting all cases in (a) to cases in (b).

The second experiment (ii) approximates removing all bank group conflicts (but keeping bank conflicts) by setting the long variant (same bank group) of each timing constraint to the same value as the short variant (different bank group); that is, we set $t_{RRD\_L} = t_{RRD\_S}$, $t_{CCD\_L} = t_{CCD\_S}$, and $t_{WTR\_L} = t_{WTR\_S}$[3], in essence converting all cases in (b) to cases in (c). Cases in (a), however, remain bank conflicts and still suffer from serialized Precharges/Activates to the same bank.

The third experiment (iii) approximates removing both bank and bank group conflicts by relaxing all bank mapping constraints, allowing any request to be alternatively serviced by any other bank/bank group[2], converting both (a) and (b) accesses into (c) accesses.

Fig. 3 shows the results of the three experiments across a set of eleven 4-core multi-programmed workloads formed from the memory intensive SPEC 2006 benchmarks and Graph 500. The workloads are listed in Table II. Removing bank conflicts in (i) improved performance by 7.2% – 25.9% across the workloads, and by 14.8% on average; removing bank group conflicts in (ii) improved performance by 5.7% to 25.2% across the workloads, and by 11.6% on average; removing both bank and bank group conflicts in (iii) improved performance by 12.6% to 37.5% across the workloads, and by 22.5% on average. The results show removing bank and bank groups significantly improve performance, and removing both improved performance far more than removing either one in isolation.

Experiment (iii) approximated removing all bank and bank conflicts by allowing any request to be serviced by any other bank/bank group. Such relaxation is possible if all data are fully duplicated to all bank/bank groups; unfortunately, full duplication has unacceptable storage and coherence overheads.

While full duplication is infeasible, we find limited duplication is sufficient to remove most bank/bank group conflict penalties. In experiment (iv), only the bank mapping to the next bank group is relaxed, so requests to banks in bank group m can only be alternatively serviced by banks in bank group m+1 (mod 4). Fig. 3 shows that (iv) retains most of the benefit of (iii), improving performance by 12.4% – 35.5% across workloads, and by 20.2% on average. Experiment (v) further restricts duplication such that requests to bank (m,n) can only alternatively be serviced by bank (m+1 (mod 4), n). This improves performance by 10.7% – 29.7% across workloads, and by 17.2% on average, which is worse than (iii) and (iv), but still substantial.

(iv) and (v) only require duplication of data between pairs of bank groups, as opposed to all-to-all duplication. We can further reduce the level of duplication required by applying the caching principle and only duplicate a select subset of data from each bank group to the next bank group. To this end we propose the Duplicon Cache, a technique that mitigates the penalties of bank and bank group conflicts by duplicating select lines of data to an alternate bank group. We identify and address the following key components of Duplicon:

1) reserving storage space for duplicates in each bank (Section IV-A)
2) identifying the right granularity of duplication (Section IV-B)
3) determining where to duplicate data to (IV-C)
4) tracking duplicated data at the memory controller (IV-D)
5) limiting the duplication overhead (IV-E)
6) ensuring coherence between duplicates (IV-F)
7) determining which data to duplicate (IV-G)
8) determining which duplicated data to replace (IV-H)

IV. Duplicon Cache

A. Data Store: reserving storage for duplicates

A cache is made up of a Data Store and a Tag Store. The Duplicon Cache Data Store is created by reserving space in different bank groups for storing duplicated data. This is done by reserving a small region at the end of the physical memory address space at boot time, as shown in Fig. 4(a). We called this reserved physical address space the Reserved Storage. With 2^m bytes of total physical memory capacity, we reserve 2^k bytes of space at the end of the address space to store duplicate data. The Operating System (OS) is then led to believe there are only $2^m - 2^k$ bytes of physical memory available, and will not allocate memory in the Reserved Storage. In our evaluated configuration $m = 34$ and $k = 27$; that is, we reserve $2^{27}B = 128MB$ of storage out of $2^{34}B = 16GB$ of total physical memory, for a storage overhead of 1/128.

B. Line Size: granularity of duplication

The width of the DDR4 DRAM data bus is 64 bits. Individual DRAM devices have a narrower interface than the width of the data bus – for example, x4 devices have a 4

[2]the request still accesses the same row and columns at the new bank

[3]$t_{WTR\_L/S}$ is the delay between the end of a write burst and a subsequent Read to the same/different bank group

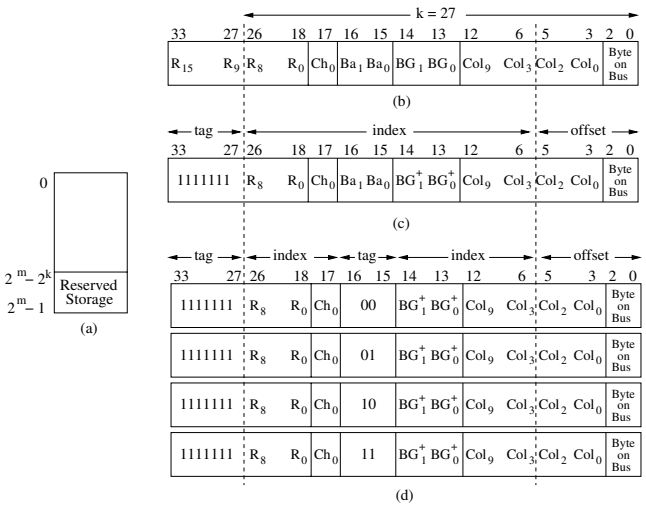

Fig. 4. (a) reserving a region in physical memory for duplicates, (b) the home physical address, (c) single duplication destination way in a direct–mapped Duplicon Cache (d) set of possible duplication destination ways in a 4–way set–associative Duplicon Cache.

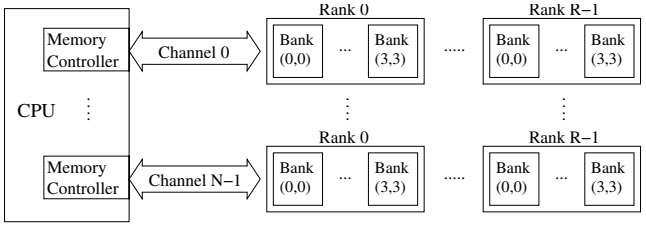

Fig. 5. Memory Controllers, Channels, Ranks, and Banks.

bit interface; x8 devices 8 bits; x16 devices 16 bits. DRAM devices are combined to match the width of the data bus. The combined DRAM devices form a DRAM rank. For example, 16 x4 devices form a rank (or 8 x8 devices; 4 x16 devices). Devices in the same rank operate in lockstep on the same commands and addresses, and each (bank group, bank, row, column) tuple specifies a 64 bit (8 bytes) piece of data in a rank.

DDR4 DRAM transfers data in bursts. The typical burst length is 8, so each access results in a burst of 8 columns in succession. The columns transferred are aligned, so columns 0–7 are always transferred together in the same burst, while columns 8–15 belong to another burst, etc. Since each column specifies an 8 byte piece of data in the rank, and since 8 columns are transferred in a burst on every access, the granularity of access to DDR4 DRAM is $8 \times 8 = 64$ bytes. We thus choose 64 bytes as the Duplicon Cache line size, the same as the granularity of access of DRAM, and duplicate data at 64 byte granularity.

C. Set Associativity: where to duplicate data to

Several ranks may share the same set of data/command/address buses that connect the DRAM to the processor. Each set of data/command/address buses is called a channel, and there are typically multiple channels. Data in DRAM is thus uniquely identified by its (Ch=channel, Ra=rank, BG=bank group, Ba=bank, R=row, Col=column) tuple. Different channels may be controlled by different memory controllers. The hierarchy of memory controllers, channels, ranks, and banks is shown in Fig. 5.

A mapping function maps a physical address to its corresponding (Ch, Ra, BG, Ba, R, Col) tuple. Each component of the tuple is computed as a hash of a subset of physical address bits. Fig. 4(b) shows an example mapping function: bit 17 is the channel bit (implying there are 2 channels); there are no rank bits (implying there is 1 rank per channel); bits 14 and 13 are the bank group bits; bits 16 and 15 the bank bits; bits 33 down to 18 the row bits (implying 64K rows per bank); bits 12 down to 3 the column bits (1k columns per row); bits 2 down to 0 specify the byte on the 8 byte wide data bus (i.e., offset within a column). We use this mapping in our evaluation, as it maximized the baseline performance among the different mappings we tried (including mappings that placed the channel bits lower).

All data in memory can be found in its home location, pointed to by its home address (Fig. 4(b)). Data may then be duplicated from its home location to the Reserved Storage. To access the Reserved Storage, the high bits $m-1$ down to k of the address are set to 1, while the low k bits dictate where in the Reserved Storage we are accessing. Where we duplicate data to in the Reserved Storage is a principal design decision for the Duplicon Cache and is subject to various constraints and tradeoffs

First, we must duplicate to a different bank group from the home location. This means the Reserved Storage must span at least two bank groups, requiring an overlap between the bits used to hash to the bank group and the low k bits of the address. In practice this overlap almost always exists, since the bank group bits are almost always placed among the lower order, higher entropy address bits to maximize bank group interleaving and minimize bank group conflicts. Some mapping schemes may include bits from outside the low k bits –by, for example, xor–ing lower order bits with higher orders bits above bit k to produce the bank group. This is fine as long as multiple bank groups can be reached while the address bits k and above are set to 1.

Next, Duplicating data across channels is undesirable as it necessitates moving data between different memory controllers, incurring additional data movement costs. Thus Duplicon only duplicates data within the same channel. This means only data residing in channels spanned by the Reserved Storage can be duplicated. In practice the Reserved Storage will almost always span all channels, since the channel bits, like the bank group bits, are usually placed among the lower order, higher entropy address bits to maximize channel interleaving.

While hard constraints exist for the duplication destination bank group (must be different from the home location) and the destination channel (must be the same), some flexibility exists for the duplication rank/bank/row/column. On one extreme we can have a direct–mapped scheme, shown by Fig. 4(c),

where the low k bits of the duplication destination address are identical to the low k bits of the home address, except that the bank groups bits (bits 14 and 13) BG_1BG_0 are replaced by $BG_1^+BG_0^+$, where $BG^+ = BG + 1$ (mod 4). In this scheme each line of data can only be duplicated to a single location in bank group BG^+. Alternatively, Duplicon can be organized as a 4–way set–associative cache, shown in Fig. 4(d). Here, in addition to the bank group bits BG_1BG_0 being replaced by $BG_1^+BG_0^+$, the bank bits (16 and 15) are now completely free in the duplication destination address, meaning data can be duplicated to any of 4 banks in bank group BG^+. Note the analogy to traditional caches: with traditional caches, a direct–mapped scheme is one in which the data can only be cached in a single location, while an j–way set–associative scheme is one in which the data may be cached in any one of j ways in a given set, and all of them need to be searched for a cache hit. In our case, the 4 banks of bank group BG^+ form the 4 ways of our 4–way set–associative Duplicon Cache. As with traditional caches, the original home address bits can be divided into (i) offset bits that dictate the byte offset within a line, (ii) index bits that dictate the set of locations to which the data may be cached, and (iii) tag bits that need to be tracked in order to differentiate between different data that map to the same set (i.e., have the same index bits). The breakdown of offset, index, and tag bits for both the direct mapped and 4–way set–associative schemes are shown in Fig. 4.

The direct–mapped cache corresponds to experiment (v) in Fig. 3, where data in bank (m,n) can only be alternatively serviced by bank (m+1 (mod 4), n); the 4–way set–associative cache corresponds to experiment (iv), where requests to bank group m can be alternatively serviced by any bank in bank group m+1 (mod 4). Fig. 3 shows the 4–way set–associative cache performs better, so the 4–way set–associative configura–tion is used in the rest of the paper.

D. Tag Store: duplicate tracking

Duplicon maintains a Tag Store in a dedicated SRAM table at the memory controller to track which data have been duplicated. The Tag Store is searched upon each memory request to check if a duplicated copy exists.

The Duplicon Cache is much larger than an ordinary pro–cessor cache; we sized the Duplicon Cache to be 128MB in our evaluation. Storing tags for so much data is expensive. Duplicon reduces the storage cost of tags via a sectored cache design [2], [3] where the cache sectors are DRAM rows. All columns in a sector share a single Address Tag, reducing the size of the Tag Store. The Tag Store additionally maintains valid bits per sector to mark which columns have been duplicated. One valid bit is required per Duplicon Cache line. As the Duplicon line size is 8 columns (64B), one valid bit is needed for every 8 columns; 128 valid bits are needed for the 1K columns of each sector. Collectively the valid bits form the Valid Columns Mask.

Since different channels may have different memory con–trollers, a separate Tag Store is maintained for each channel to track duplicated data within that channel. At each channel,

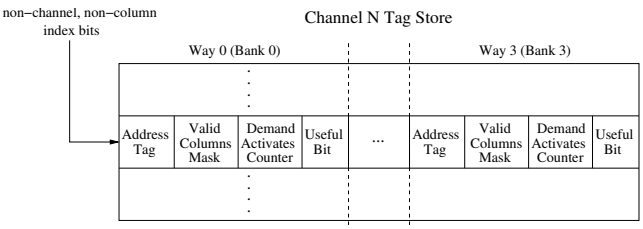

Fig. 6. Duplicon Cache Tag Store.

we use the non–column (because all columns of the same row belong to the same sector and share the same Address Tag) index bits from the home address to index into a set in the Tag Store. Each way in the set requires:

1) an Address Tag to identify the sector (i.e., which DRAM row the sector contains)
2) a Valid Columns Mask to identify which columns have valid data
3) a Demand Activates Counter (DAC), saturating counter used for the Duplicon Cache insertion policy (Section IV–G)
4) a Useful Bit, used for the Duplicon Cache replacement policy (Section IV–H)

Fig. 6 shows the Tag Store for a particular channel. For an access to hit in the Duplicon cache, the Address Tag for the sector needs to match, and the corresponding bit in the Valid Columns Mask needs to be set.

Section V–C1 addresses the Tag Store area cost. The Tag Store requires 142KB/channel in our evaluated configuration, for a total of 284KB with two channels.

E. Cache Fill: duplication

Data can be filled (i.e., duplicated) into the Duplicon Cache in two cases. The first is when the data is read from memory on a normal read request; the second is when the data is written to memory on a normal write request. In both cases the data passes through the memory controller, at which time a new duplication write request to the appropriate location in the Re–served Storage is created. Once created, the duplication write request gets queued and serviced by the memory controller like normal write requests.

Reducing write-caused interference: Normal write requests already interfere with the servicing of read requests. Typically memory controllers have write buffers [4]–[6] that batch up write requests, allowing read requests to be serviced without interference until the write buffer is drained. Duplicon simi–larly relies on the write buffer to batch up duplication write requests to minimize write–caused interference. If the write buffer is full, then duplication write requests can simply be dropped.

F. Coherence

Existing duplicates must be invalidated on every write request to the line by clearing the appropriate bit of the Valid Columns Mask in the Tag Store entry. In addition, the write

buffer must be searched to remove any pending duplication write requests to the line. This does not increase the write buffer hardware complexity, as existing write buffers already need to support searches for a particular line to allow data from buffered write requests to be forwarded to subsequent matching read requests.

G. Duplication policy

1) Demand Activates Filtering: Duplication incurs non–trivial costs in terms of storage and extra memory write traffic; thus it is important to only duplicate data that are likely to impact program performance. Duplicon uses two criteria to determine whether data should be duplicated. First, Duplicon tracks how often the data is accessed via demand (as opposed to prefetch) read requests (including requests that began as prefetch requests but then matched a demand request), as such requests are likely to be on the program critical path. Second, Duplicon tracks how often the data suffers from DRAM row misses/conflicts, as such accesses incur longer latencies.

Both criteria can be measured by counting the number of Demand Activates to the data. A Demand Activate is an Activate to a row for a demand read request. The number of Demand Activates identifies the number of demand non–row buffer hit accesses to the data, as row buffer hits do not require an Activate.

Duplicon tracks the number of Demand Activates in the Tag Store, which maintains a saturating Demand Activates Counter(DAC) for each cache sector (Section IV-D). We allocate a sector in one of the ways of the Tag Store on the first Demand Activate to the row, and increment the DAC for each subsequent Demand Activate. Duplication of lines in the row only proceeds after the DAC surpasses a threshold (Thrsh), but once the threshold is reached we duplicate on all accesses, not just Demand Activates. We swept over a large range of Thrsh values and found 15 to be a sweet spot for our evaluated configuration.

2) Number of duplicates allowed: The other parameter in the insertion/duplication policy is the number of duplicates allowed. Allowing multiple duplicates of the same data further decreases the likelihood of bank conflicts for that data, but at the cost of storage. Results in Section V show that having more than one duplicate provides little value, so we only allow a single duplicate.

H. Replacement policy

The Duplicon Cache has limited storage and associativity. Once a set is fully occupied, any new row that maps to the same set must either overwrite an existing sector in the set (replace), or not be allocated in the set (bypass).

1) Usefulness Tracking: Duplicon adopts the same replace–ment policy as the TAGE branch predictor [7] and tracks the usefulness of each duplicated cache sector via the Useful Bit in the Tag Store (Section IV-D). Sectors are initially marked not useful, but become useful when a duplicated column in the sector gets used (i.e., when the duplicated column gets sourced by a read request because of a conflict at the home bank/bank group). Sectors that are marked as useful definitely cannot be replaced; periodically all the Useful Bits are cleared. We swept over a range of Useful Bit reset periods and found resetting the Useful Bits every million memory requests worked well, although the sensitivity to the reset period is quite low provided the period is large enough.

Based on the values of the Useful Bit and the Demand Activates Counter(DAC), each cache sector in main memory is in one of four states:

1) **Invalid** (DAC = 0): no row has been allocated to the cache sector yet; the cache sector is empty
2) **Monitoring** ($1 \leq$ DAC $<$ Thrsh): a row has been allocated to the cache sector, and we are tracking Demand Activates to increment the DAC, but not yet duplicating
3) **Duplicating-not useful** (DAC $\geq$ Thrsh, Useful=0): a row has been allocated to the cache sector and we are duplicating on all accesses (even writes and prefetches), but no duplicated data has been used; the sector may be replaced
4) **Duplicating-useful** (DAC $\geq$ Thrsh, Useful=1): a row has been allocated to the cache sector and we are duplicating on all accesses (even writes and prefetches), and dupli–cated data has been used; the sector may not be replaced

2) Probabilistic Replacement: The Useful Bit protects sec–tors in the *Duplicating-useful* state from being overwritten, but does not protect sectors in other states. To give sectors in the *Monitoring* and *Duplicating-not useful* time to reach the *Duplicating, useful* state, we introduce a parameter ϵ which controls the probability that a sector in states *Moni-toring* or *Duplicating-not useful* can be replaced. A properly chosen ϵ parameter should give time for beneficial sectors in *Monitoring* and *Duplicating-not useful* to become useful, while still eventually replacing the sectors that never do. We performed a sweep of the ϵ parameter and used ϵ = 1/256 in our experiments.

The state machine for each cache sector is shown in Fig. 7. All cache sectors start in the *Invalid* state, and transition to the *Monitoring* state on the first Demand Activate, at which point the sector is allocated to the row. Sectors in the *Monitoring* state have their DAC incremented on each sub–sequent Demand Activate to the row. When the DAC reaches Thrsh, the sector transitions to the *Duplicating-not useful* state. From this state data is duplicated on each subsequent access (including prefetches and writes). If all the sectors in the set are allocated, then sectors in the *Monitoring* and *Duplicating-not useful* states may be replaced with probability ϵ each time another row wishes to allocate into the set. If a replacement occurs, then the Address Tag is updated to the new row, the Valid Columns Mask and Useful Bit are cleared, and the Demand Activates Counter is set to 1, putting the new row/sector in the *Monitoring* state. Sectors in *Duplicating-not useful* are promoted to *Duplicating, useful* when a duplicated column is used. Sectors in *Duplicating-useful* are protected from replacement. On a Useful Bit reset all sectors in the *Duplicating-useful* state are demoted to the *Duplicating-not useful* state.

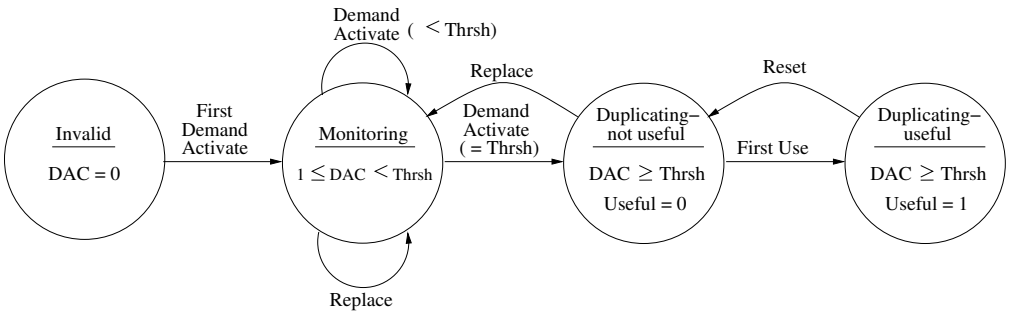

Fig. 7. Cache sector state diagram.

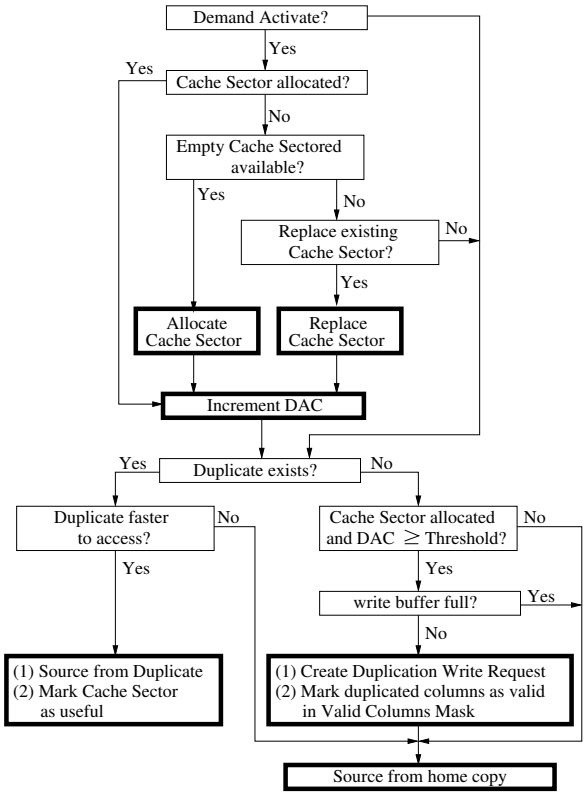

Fig. 8. Flow chart for read.

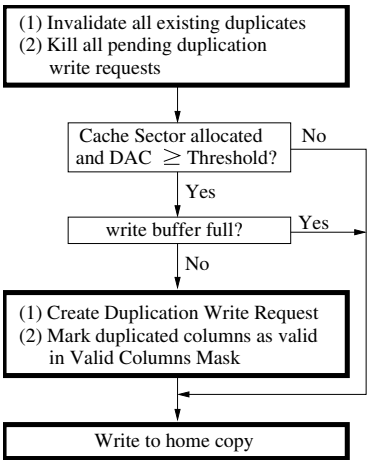

Fig. 9. Flow chart for write requests.

Fig. 8 is a flow chart that summarizes the sequence of events on read requests; Fig. 9 summarizes the sequence of events on write requests. Actions in both flow charts are enclosed in boldface boxes.

V. Evaluation

A. Methodology

We evaluated our mechanism on an execution-driven, cycle-accurate simulator for a 4-core out-of-order x86 processor. The frontend of the simulator is based on Multi2Sim [8]. The simulator models port contention, queuing effects, and bank conflicts throughout the cache hierarchy and includes a detailed DDR4 SDRAM model which models t_{CL}, t_{CWL}, t_{RP}, t_{RCD}, t_{RAS}, t_{RTP}, $t_{CCD(L/S)}$, $t_{RRD(L/S)}$, t_{FAW}, $t_{WTR(L/S)}$, and t_{WR}. Table I describes our baseline configuration. Chip power and energy are modeled using McPAT [9], and DRAM power and energy are modeled using CACTI [10].

The bank conflict probability is dependent on the banks/thread ratio in the system; the higher the ratio, the less likely bank conflicts. The latest Intel Xeon Processor E7-8894 v4 server CPU [11] supports 56 threads on 6 DDR4 channels. Servers are typically configured with 2 or 4 ranks per channel for optimal memory performance; loading the channel beyond 4 ranks results in having to run the channel at a lower frequency, hurting performance [12]. With 16 banks per rank, 4 ranks per channel, and 6 channels, there are $16 \times 4 \times 6 = 384$ banks in the system for 56 threads, for a banks/thread ratio of 384/56 = 6.86. Our evaluated configuration has 2 channels, 1 rank per channel, and 16 banks, for a total of 32 banks, and 4 threads, resulting in a banks/thread ratio of 8. This is higher than the banks/thread ratio in a current state-of-the-art system; hence our evaluation conservatively underestimates the impact of bank conflicts.

To mimic the effects of virtual-to-physical address translation in our simulator, we pass the Virtual Page Number (VPN) concatenated with the processor ID through a hash function (Paul Hsieh's SuperFastHash [13]) to generate the

TABLE I
BASELINE CONFIGURATION.

Core	4–Wide Issue, 128 Entry ROB, RS size 48, Hybrid Branch Predictor, 3.2 GHz Clock
L1 Caches	32KB I-Cache, 32KB D-Cache, 64 Byte Cache Lines, 2 Reads Ports, 1 Write Port, 3 Cycle Latency, 4–way Set–Associative, Write–back
Last Level Cache	Shared 4MB, 64 Byte Cache Lines, 12 Cycle Latency, 8–way Set–Associative, Write–back, Inclusive
Memory Controller	128 Entry Memory Queue, FR–FCFS [15] Open–Page Policy
Prefetcher	Stream Prefetcher [16]: Streams 64, Distance 64, Queue 128, Degree 4 with Feedback Directed Prefetching(FDP) [17] to throttle prefetcher
DRAM	2 Channels, 1 Rank/Channel, 16 Banks/Rank, 8Gb DDR4–3200 x8 chips Bus Frequency 1.6GHz (DDR 3.2GHz) t_{RCD}–t_{RP}–t_{CL}: 22–22–22 8KB Row Buffer (1KB x8)

"Physical Frame Number", which is then combined with the page offset to form the physical address. The DRAM chan–nel/bank group/bank/row/column addresses are then computed using the mapping function in Fig. 4(b) from this generated physical address. The processor ID needs to be passed to the virtual-to–physical SuperFastHash function to ensure that the same virtual page from different benchmarks running in the same multi–programmed workload do not get mapped to the same physical frame. By introducing this additional virtual–to–physical hashing in our simulation, we already maximize the entropy in the channel/bank group/bank; consequently, we believe bank conflict rates we see in our evaluations are probably very close, if not better, than bank conflict rates achievable by any actual hashing function (including ones that XOR bank group/bank bits with the low order row bits).

The ten most memory–intensive SPEC 2006 benchmarks with the highest single–threaded Last–Level–Cache (LLC) misses per kilo instructions (MPKI) without prefetching, along with the Graph 500 benchmark, were used to form 11 ran–domized 4–core multi–programmed workloads such that each benchmark appears in four workloads. Table II shows the workloads. Each workload is simulated until every application in the workload has completed at least 800 million instructions from a representative SimPoint [14]. Static power of shared structures is dissipated until the completion of the entire work–load. Dynamic counters stop updating upon each benchmark's completion.

We report the *Harmonic Mean of Weighted-IPCs* [18] for Chip–Multiprocessor (CMP) performance. The *Harmonic Mean of Weighted-IPCs* is the reciprocal of the *Average Normalized Turnaround Time(ANTT)* [19], and is a measure of both fairness and system throughput [19], [20]. All perfor–mance graphs report *Harmonic Mean of Weighted-IPCs* unless otherwise stated. We additionally report the *Weighted Speedup* [21] and *Unfairness* [20], [22], [23] for our best performing configuration. *Weighted Speedup* differs from *Harmonic Mean*

TABLE II
EVALUATED MULTI–PROGRAMMED WORKLOADS.

Name	Workloads
WL–1	bwaves + Graph500 + lbm + mcf
WL–2	bwaves + lbm + mcf + sphinx3
WL–3	bwaves + lbm + omnetpp + milc
WL–4	GemsFDTD + bwaves + Graph500 + leslie3d
WL–5	GemsFDTD + Graph500 + milc + soplex
WL–6	GemsFDTD + lbm + mcf + libquantum
WL–7	GemsFDTD + leslie3d + omnetpp + soplex
WL–8	Graph500 + leslie3d + libquantum + omnetpp
WL–9	leslie3d + libquantum + soplex + sphinx3
WL–10	libquantum + mcf + milc + sphinx3
WL–11	milc + omnetpp + soplex + sphinx3

of Weighted-IPCs in that *Weighted Speedup* is only a measure of system throughput. The equations for *Harmonic Mean of Weighted-IPCs(HMWI)*, *Weighted Speedup* (*WS*), and *Unfair-ness* are given below:

$$HMWI = \frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{shared}}}, WS = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$Unfairness = \frac{MAX(\frac{T_0^{shared}}{T_0^{alone}}, \frac{T_1^{shared}}{T_1^{alone}}, \ldots, \frac{T_{N-1}^{shared}}{T_{N-1}^{alone}})}{MIN(\frac{T_0^{shared}}{T_0^{alone}}, \frac{T_1^{shared}}{T_1^{alone}}, \ldots, \frac{T_{N-1}^{shared}}{T_{N-1}^{alone}})}$$

Where N is the number of cores, IPC_i^{alone} is the IPC of application i running alone on one core in the CMP system while other cores are idle, IPC_i^{shared} is the IPC of application i running on one core while other applications are concurrently running on other cores, T_i^{alone} is the number of cycles it takes application i to run alone, and T_i^{shared} is the number of cycles it takes application i to run with other applications.

B. Performance results and analysis

Ideal vs. realized performance: Fig. 10 compares the ide–alized performance potential of the Duplicon Cache against actual realized performance. We start with the idealized ex–periment (iv) in Fig. 3. Recall experiment (iv) allowed any requests to bank group m to be alternatively serviced by banks in bank group m+1 (mod 4). Effectively this represents an idealized Duplicon Cache where

1) there are no cold misses (i.e., everything is already duplicated)
2) data are duplicated to all banks in the alternate bank group (i.e., four duplicate copies are available)
3) all rows can be duplicated (i.e., no Demand Activate Filtering)
4) the Tag Store and Reserved Storage are infinitely sized
5) duplication write requests are free and do not cause interference

These idealized assumption are removed one by one until we end up with a realistic Duplicon Cache implementation.

Cold misses are added in experiment (1); now only data that have been encountered before can be serviced by the alternate bank group. Introducing cold misses removes some

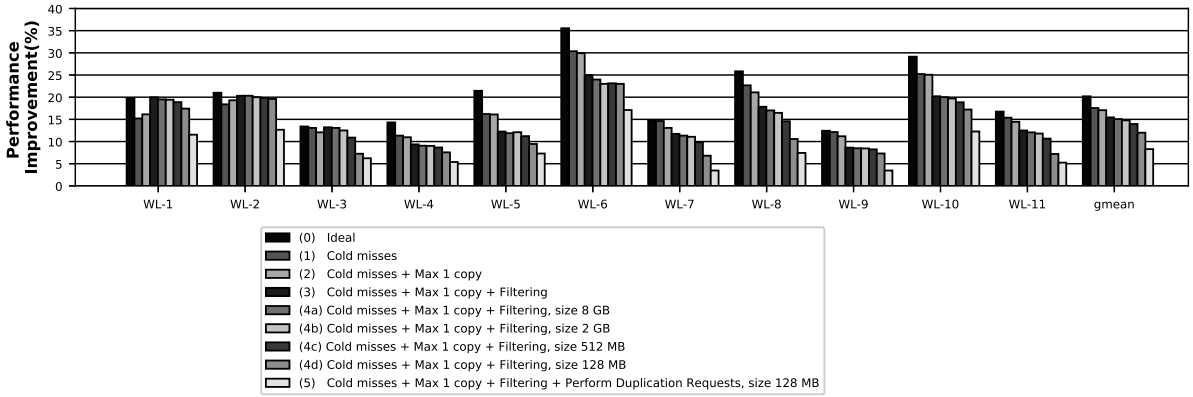

Fig. 10. Ideal vs. realized performance improvement.

of the potential, reducing the average potential performance gain from 20.2% to 17.6%.

Experiment (2) limits the maximum number of duplicates to 1; now each line of data is assigned to a single bank in the alternate bank group the first time it is encountered, and subsequent accesses can only alternatively source the data from that bank (previously the request can be serviced by any bank in the alternate bank group). Adding this constraint had little effect, changing the potential performance gain from 17.6% to 17.1%, which justifies the design choice to limit the maximum of duplicates to 1 in Section IV-G2.

Note in some cases (WL-1 and WL-2) limiting the maximum number of duplicates to 1 actually improved performance. This is because sourcing from an alternate bank can actually reduce row buffer locality in the alternate bank. In general row buffer locality improves as we limit duplication, since each requests stays in its home bank and does not interfere with rows in other banks. However when bank conflicts do occur the penalty is lower if the data has been duplicated to another bank.

Experiment (3) considers the effects of Demand Activates Filtering (Section IV-G). Demand Activates Filtering reduces the number of useless duplications, but also limits which rows can be duplicated, decreasing the performance gain potential. We modeled an infinite sized Tag Store and tracked Demand Activates for each DRAM row encountered. Recall duplication is only allowed after the row reaches the the Demand Activates threshold (Thrsh), so data has to be seen one more time after the row reaches the threshold before we allow it to be sourced from the alternate bank group. On average Demand Activates Filtering reduces the potential from 17.1% to 15.4%. Again on select workloads (WL-1, WL-2, WL-3) adding Demand Activates Filtering, which limits duplication, results in better row buffer locality and can improve the performance.

Experiments (4a), (4b), (4c), and (4d) consider the effect of sizing the Duplicon Cache from infinite sized to 8GB, 2GB ,512MB, and 128MB. The average performance gain drops from 15.4% to 15.1%, 14.8%, 14.0%, and 12.0%, respectively, showing bigger Duplicon Cache sizes can provide additional

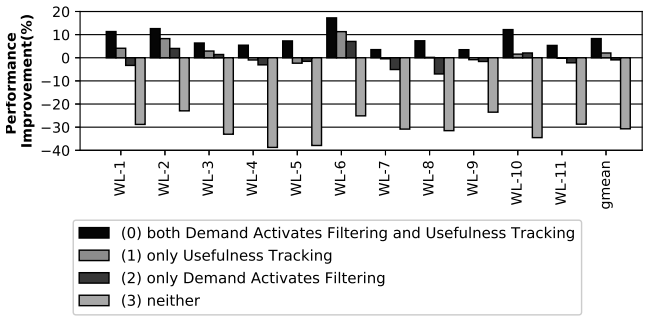

Fig. 11. Performance without Demand Activates Filtering/Usefulness Tracking.

gains, but at the cost of both additional main memory and Tag Store storage.

Up until now we have assumed duplication to be free. Experiment (5) considers the cost of actually performing duplication write requests. This drops the performance gain from 12.0% to 8.3%. 8.3% is the final performance gain realized after considering all costs and constraints.

Effectiveness of insertion and replacement policy: Duplicon employs Demand Activates Filtering (Section IV-G) to reduce the number of useless duplications, and Usefulness Tracking (Section IV-H1) to protect useful duplicated lines. Fig. 11 shows the importance of both mechanisms. (0) is the performance gain with both mechanisms; (1) is when Demand Activates Filtering is removed (i.e., the *Monitoring* state is removed from the state diagram in Fig. 7); (2) is when Usefulness Tracking is removed (i.e., the *Duplicating-useful* is removed); (3) is when both Demand Activates Filtering and Usefulness Tracking are removed. The results clearly show both mechanisms are required: removing Demand Activates Filtering drops the average performance gain from 8.3% to 2.1%; removing Usefulness Filtering drops the average performance gain to -0.9%; removing both drops the average performance gain to -30.7%. There is synergy between the two mechanisms – Usefulness Tracking is based on actual duplication outcome (i.e., something duplicated in this row

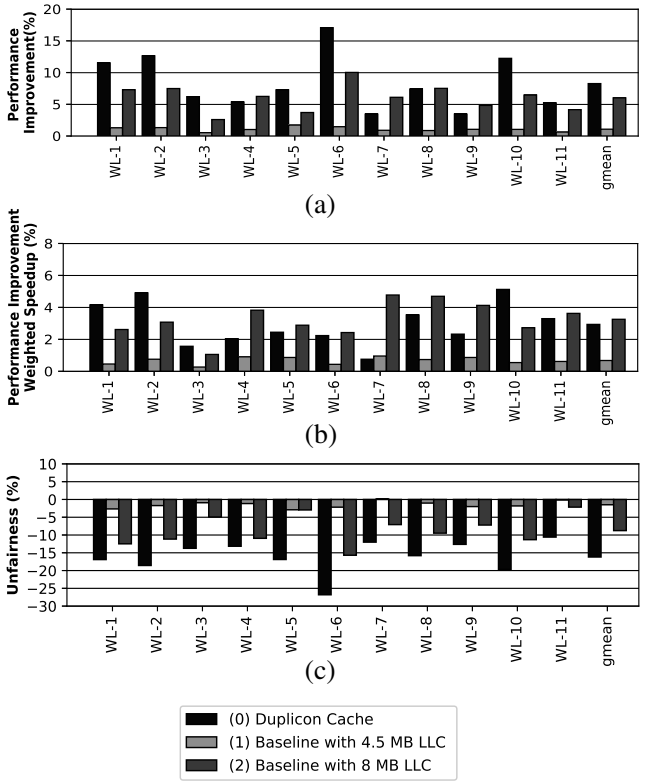

Fig. 12. Performance comparison to baseline with added LLC using different metrics: (a) *Harmonic Mean of Weighted-IPCs*, (b) *Weighted Speedup*, (c) *Unfairness*

was actually later used), whereas Demand Activates Filtering is heuristics based. We use the heuristics based Demand Activates Filtering to first determine what might be suitable for duplication, then use Usefulness Tracking to more rigorously evaluate if the row should have been duplicated.

Performance comparison to area equivalent baseline: The Duplicon Cache Tag Store takes up a non-trivial amount of storage – 284KB in total for our evaluated configuration (see Section V-C1 for details). This additional storage alternatively could have been used elsewhere on chip to improve performance – by increasing the on-chip Last-Level-Cache (LLC), for example. Since we evaluated with a 4MB 8-way set-associative LLC, the smallest increment at which the LLC can be increased is by an extra way, or 512KB, which is nearly double the amount of storage we added. Nonetheless we conservatively compare the Duplicon Cache with a 4MB LLC (config 0) against the baseline with a 4.5MB LLC (config 1). We in addition compare against the baseline with a 8MB LLC (i.e., doubling the LLC) (config 2). In addition to reporting performance in terms of the *Harmonic Mean of Weighted-IPCs* as in the rest of the paper, we also report the *Weighted Speedup* and *Unfairness*. For the *Unfairness* metric, lower is better. Fig. 12 shows the comparison.

Duplicon outperforms the baseline with 4.5MB LLC in all of our workloads and by all metrics. Duplicon also outperforms or matches the baseline with 8MB LLC in most cases, so while Duplicon requires additional on-chip area for the Tag Store, the extra area cost is well justified.

C. Area/storage/energy results and analysis

1) Tag Store area: The Tag Store is in dedicated SRAM tables at the memory controller. Each Tag Store entry is made up four fields: Address Tag, Valid Columns Mask, Demand Activates Counter, and the Useful bit. The width of each field is computed below:

1) **Address Tag (9 bits)** : Fig. 4 shows that there are 9 tag bits in the 4-way set-associative scheme (d). The 9 bits are: bits 33 to 27 of the physical address (row bits 15 to 9), and bits 16 and 15 of the physical address (bank bits 1 and 0)
2) **Valid Columns Mask (128 bits)**: as the Duplicon Cache line size is 8 columns (64B), one valid bit is needed for every 8 columns; there are 1K columns in each sector, so 128 total valid bits are required
3) **Demand Activates Counter (4 bits)**: we empirically found 15 to be a good value for the Demand Activates Counter threshold; thus we use a 4-bit saturating counter
4) **Useful Bit (1 bit)**

Each Tag Store entry has 9 + 128 + 4 + 1 = 142 bits. Each Tag Store set has 4 ways, so there are 4 $\times$ 142 = 568 bits per set. Recall we index into a Tag Store set using the non-channel, non-column index bits. Fig. 4 shows that are 11 such bits: bits 26 to 18 of the physical address (row bits 8 to 0), and bits 14 and 13 of the physical address (bank group bits 14 and 13). Thus there are $2^{11} = 2048$ sets in each Tag Store, and 2048 $\times$ 568 = 1163264 bits = 142KB per Tag Store table – i.e., 142 KB/channel (recall we maintain a Tag Store table per channel). Our configuration has two channels, so the total storage cost is 2 $\times$ 142KB = 284 KB. We believe this is a non-negligible but acceptable amount of storage to add to the uncore floorplan, as processor performance is less sensitive to uncore latencies as opposed to core latencies.

2) DRAM storage: Duplicon incurs 32MB/core of memory capacity overhead. Normal fluctuation between peak and average memory utilization of a datacenter already far exceeds 32MB/core, and a reasonably provisioned system will be able to absorb this additional overhead. In fact, [24] shows current datacenters only typically use around 40%-50% of memory.

3) Energy: Duplicon introduces extra power in two ways:

(a) Extra leakage from the Tag Store, and extra dynamic power due to Tag Store accesses
(b) Extra DRAM power from duplication write requests

For (a), we model the Tag Store as a cache in McPAT, model read and write accesses to the Tag Store as read and write accesses to the cache, and add the power/energy contribution from the cache to the total power/energy. For (b), we account for the duplication write requests in our CACTI DRAM power and energy model.

Fig. 13 shows the energy results. Duplicon reduces the total energy on 9 out of 11 workloads, reducing the total energy by

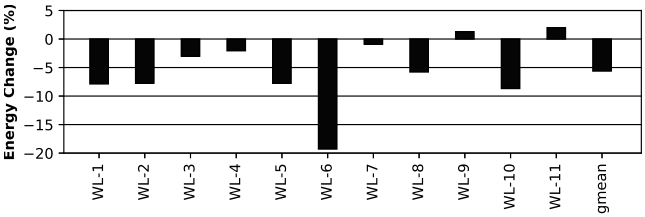

Fig. 13. Duplicon Cache Energy evaluation.

5.6% on average. The energy savings come from reducing the workload execution time.

VI. Related Work

While the general principle of duplicating data for reliability or performance (e.g., RAID) is well established, we are not aware of any prior work that specifically duplicates data in DRAM to reduce bank conflicts.

Data duplication in DRAM was proposed in RowClone [25]. Duplicon differs significantly from RowClone in that data duplication in RowClone is performed as part of the program –to, for example, clear an array by copying the zero page to it. Duplication in RowClone is architecturally visible and under explicit software control via special `memcopy` instructions. In contrast, Duplicon Cache duplication is performed for performance reasons, and is completely transparent to software.

Several work reduced memory latency by modifying DRAM to create fast and slow regions, then mapping hot data to the fast regions. Tiered-Latency DRAM [26] partitioned subarrays using isolation transistors into fast rows and slow rows, then cached hot data in the fast rows. CHARM [27] created fast subarrays with high aspect-ratios, then mapped hot pages to them. Dynamic Asymmetric-Subarray DRAM [28] and LISA [29] built upon Tiered-Latency DRAM and CHARM by proposing better mechanisms to move data between adjacent subarrays, either for bulk data transfers or to migrate data between fast and slow subarrays. LISA can also be used to reduce precharge latency. Multiple clone row DRAM [30] reduced access latency by using multiple physical rows to store a single logical row, increasing the number of sensed cells and reducing the sensing latency. However, in all these schemes conflicting requests to the same bank still need to be processed serially, although the queuing delay can be reduced if the conflicting requests hit in the fast region. In contrast, Duplicon allows conflicting requests to be serviced concurrently in different banks. In addition, all these schemes require changing commodity DRAM, while Duplicon does not.

Micron's Reduced Latency DRAM (RLDRAM) [31] and Fujitsu's FCRAM [32] reduced the number of cells per bitline. MoSys' 1T-SRAM is a high density SRAM with much faster access latency than DRAM [33]. Numerous work also examined adding an SRAM cache to DRAM [34]–[40] However, these approaches have much higher cost-per-bit compared to commodity DRAM and cannot be used as the main memory storage in a large system, whereas Duplicon Cache is built on commodity DRAM.

Liu et al. proposed reducing bank conflicts via OS-level bank partitioning [41]. This approach is problematic because there are not enough banks per thread to allow total isolation; Liu et al. stated 8–16 banks are needed per thread to achieve good performance, yet we show in Section V-A a state-of-the-art processor which supports 56 threads, 6 memory channels, and 4 ranks per channel (e.g., Intel Xeon Processor E7-8894 v4 [11]) has a banks/thread ratio of 6.86, which is short of the 8–16 banks required. Furthermore, the bank group/thread ratio is only 1.71, less than 2 bank groups per thread, guaranteeing some threads will experience bank group conflicts.

Other work [42]–[44] improve performance by improving the physical address to bank mapping, but if there are not enough banks/thread in the system (which we show in the paragraph above), then bank conflicts will occur even with an ideal mapping. DReAM [44] altered the mapping dynamically based on run-time feedback. This approach is problematic because data elements become out of place after the mapping is altered, and relocating all data elements to their new locations is expensive both in terms of performance and energy.

BLP-Aware Prefetch Issue (BAPI) increased bank-level parallelism by prioritizing prefetches that go to different banks [45]. Our baseline implementation effectively already implements BAPI, as we allow prefetches requests to be serviced out-of-order as they become ready. SALP [46] reduced the bank conflict penalty by allowing some operations to the same bank to overlap if they are to different subarrays. However, SALP requires changes to DRAM, while Duplicon does not.

Guo et al. duplicated data across different ranks of the same channel and sourced data from an alternate rank if the original rank is being refreshed [47], but did not propose using duplication to reduce bank conflicts.

VII. Conclusion

DRAM bank and bank group conflicts significantly degrade program performance. The Duplicon Cache is an effective technique to mitigate bank and bank group conflict penalties by identifying and duplicating select data across multiple bank groups. Duplicon is built on top of existing commodity DRAM and does not require changes to DRAM. The key parts of Duplicon are: 1) a set-associative and sectored architecture that allows for efficient tracking of duplicated data, 2) Demand Activates Filtering to identify which DRAM rows to duplicate, and 3) Usefulness Tracking and Probabilistic Replacement to protect useful data from being overwritten, while allowing stale data to be replaced. Our evaluation shows Duplicon improves performance by 8.3% while reducing energy by 5.6% on average.

Acknowledgment

We thank the members of the HPS research group, the anonymous reviewers, Esha Choukse, and Yongkee Kwon for insightful discussions, comments, and help with figures.

REFERENCES

[1] *MT40A1G8 DDR4 SDRAM Datasheet Rev. D*, Micron Technology, Inc., Nov. 2015.

[2] *TMS390Z55 Cache Controller, Data Sheet.*, Texas Instruments, 1992.

[3] I. Motorola, I. Microelectronics, I. B. M. Corporation, and M. Inc, *PowerPC 601: RISC Microprocessor User's Manual*. Motorola, 1993. [Online]. Available: https://books.google.com/books?id=hqesjwEACAAJ

[4] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens, "Eager writeback – a technique for improving bandwidth utilization," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/360128.360132

[5] C. Natarajan, B. Christenson, and F. Briggs, "A study of performance impact of memory controller features in multi–processor server environment," in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, ser. WMPI '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1054943.1054954

[6] J. Shao and B. T. Davis, "A burst scheduling access reordering mechanism," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346206

[7] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, 2007.

[8] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370865

[9] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172

[10] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to understand large caches."

[11] *Intel Xeon Platinum 8180 Processor*, Intel Corp., 2017, https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180.

[12] *Optimizing Memory Performance of Lenovo ThinkSystem Servers*, Lenovo, 2017, https://lenovopress.com/lp0697.pdf.

[13] P. Hsieh. (2008) Hash functions. [Online]. Available: http://www.azillionmonkeys.com/qed/hash.html

[14] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/781027.781076

[15] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/339647.339668

[16] J. Tendler, J. S. Dodson, J. S. Fields Jr., L. Hung, and B. Sinharoy, "POWER4 system microarchitecture," vol. 46, Oct. 2001.

[17] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2007.346185

[18] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," in *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, 2001.

[19] S. Eyerman and L. Eeckhout, "Restating the case for weighted-ipc met–rics to evaluate multiprogram workload performance," *IEEE Computer Architecture Letters*, vol. 13, July 2014.

[20] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736058

[21] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreaded processor," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/378993.379244

[22] R. Gabor, S. Weiss, and A. Mendelson, "Fairness and throughput in switch on event multithreading," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, Dec 2006.

[23] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, December 2007. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=79625

[24] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos–aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541941

[25] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540725

[26] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency dram: A low latency and low cost dram architecture," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, Feb 2013.

[27] Y. H. Son, O. Seongil, Y. Ro, J. W. Lee, and J. H. Ahn, "Reducing memory access latency with asymmetric dram bank organizations," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485955

[28] S.-L. Lu, Y.-C. Lin, and C.-L. Yang, "Improving dram latency with dynamic asymmetric subarray," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO–48. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830827

[29] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost inter-linked subarrays (lisa): Enabling fast inter-subarray data movement in dram," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.

[30] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L.-S. Kim, "Multiple clone row dram: A low latency and area optimized dram," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750402

[31] Micron, "Rldram memory," https://www.micron.com/products/dram/rldram-memory, accessed: 2016-04-09.

[32] Y. Sato, T. Suzuki, T. Aikawa, S. Fujioka, W. Fujieda, H. Kobayashi, H. Ikeda, T. Nagasawa, A. Funyu, Y. Fuji, K. Kawasaki, M. Yamazaki, and M. Taguchi, "Fast cycle ram (fcram); a 20-ns random row access, pipe-lined operating dram," in *VLSI Circuits, 1998. Digest of Technical Papers. 1998 Symposium on*, June 1998.

[33] P. N. Glaskowsky, "Mosys explains 1t-sram technology," *Microprocessor Report*, vol. 13, 1999.

[34] E. M. Systems, "Enhanced sdram sm2604," 2002.

[35] C. A. Hart, "Cdram in a unified memory architecture," in *Compcon Spring'94, Digest of Papers.* IEEE, 1994.

[36] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima, "The cache dram architecture: a dram with an on-chip cache memory," *IEEE Micro*, vol. 10, April 1990.

[37] W.-C. Hsu and J. E. Smith, "Performance of cached dram organizations in vector supercomputers," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org/10.1145/165123.165170

[38] G. Kedem and R. P. Koganti, "Wcdram: A fully associative integrated cached-dram with wide cache lines," in *Proceedings of the 11th Annual International Symposium on High Performance Computing Systems*. Citeseer, 1997.
[39] NEC, "Virtual channel sdram," 1999.
[40] R. H. Sartore, K. J. Mobley, D. G. Carrigan, and O. F. Jones, "Enhanced dram with embedded registers," Mar. 23 1999, uS Patent 5,887,272.
[41] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370869
[42] Z. Zhang, Z. Zhu, and X. Zhang, "A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/360128.360134
[43] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155624
[44] M. Ghasempour, J. D. Garside, A. Jaleel, and M. Luján, "Dream: Dynamic re-arrangement of address mapping to improve the performance of drams," *CoRR*, vol. abs/1509.03721, 2015. [Online]. Available: http://arxiv.org/abs/1509.03721
[45] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving memory bank-level parallelism in the presence of prefetching," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669155
[46] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337202
[47] Y. Guo, P. Huang, B. Young, T. Lu, X. He, and Q. G. Liu, "Alleviating dram refresh overhead via inter-rank piggyback caching," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, Oct 2015.