

# The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies

Jeffrey Stuecheli<sup>1,2</sup>, Dimitris Kaseridis<sup>1</sup>, David Daly<sup>3</sup>, Hillery C. Hunter<sup>3</sup> & Lizy K. John<sup>1</sup>

<sup>1</sup>ECE Department, The University of Texas at Austin, Austin, TX, USA

<sup>2</sup>IBM Corp., Austin, TX, USA

<sup>3</sup>IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA

{stuechel, kaseridi}@ece.utexas.edu, {dmdaly, hhunter}@us.ibm.com, ljohn@ece.utexas.edu

## ABSTRACT

In computer architecture, caches have primarily been viewed as a means to hide memory latency from the CPU. Cache policies have focused on anticipating the CPU's data needs, and are mostly oblivious to the main memory. In this paper, we demonstrate that the era of many-core architectures has created new main memory bottlenecks, and mandates a new approach: coordination of cache policy with main memory characteristics. Using the cache for memory optimization purposes, we propose a **Virtual Write Queue** which dramatically expands the memory controller's visibility of processor behavior, at low implementation overhead. Through memory-centric modification of existing policies, such as scheduled writebacks, this paper demonstrates that performance-limiting effects of highly-threaded architectures can be overcome. We show that through awareness of the physical main memory layout and by focusing on writes, both read and write average latency can be shortened, memory power reduced, and overall system performance improved. Through full-system cycle-accurate simulations of SPEC cpu2006, we demonstrate that the proposed **Virtual Write Queue** achieves an average 10.9% system-level throughput improvement on memory-intensive workloads, along with an overall reduction of 8.7% in memory power across the whole suite.

## Categories and Subject Descriptors

B.3.1 [Memory Structures]: Semiconductor Memories—*Dynamic memory (DRAM)*; B.3.2 [Memory Structures]: Design styles—*cache memories, Primary Memories, Shared Memory, Interleaved Memories*

## General Terms

Design, Performance

## 1. INTRODUCTION

It is now well-understood that in the nanometer era, technology scaling will continue to provide transistor density improvements,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISCA'10, June 19–23, 2010, Saint-Malo, France.

Copyright 2010 ACM 978-1-4503-0053-7/10/06 ...\$10.00.

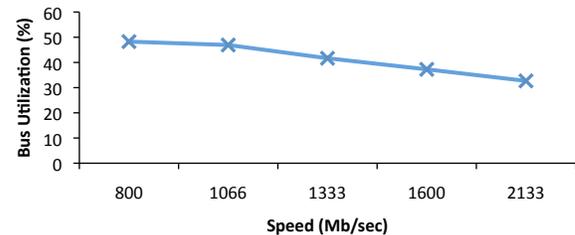


Figure 1: DDR3 single rank bus utilization efficiency, limited by DRAM parameters (tRC, tRRD, tFAW), and bus turnaround time (tWRT)

but that power density and performance improvements will slow. In response, processor designers now target chip-level throughput (instead of raw single-core performance), packing increasing numbers of cores and threads onto a chip. In 2000, virtually all server processors were single-core, single-thread; today we see 16 threads (Intel Nehalem EX), 32 threads (IBM POWER7™), and 128 threads (Sun Rainbow Falls) [3].

The processor-memory interface has been particularly challenged by this many-core trend. Technology scaling provides roughly 2x the number of transistors per lithography generation, so when core or thread counts more than double per generation, the result is generally a decrease in the available cache size per core and/or thread. From first principles, a drop in on-chip cache size will result in higher miss rates and higher memory bandwidth demands. Single socket memory demands have thus been rapidly increasing, not only due to core and thread counts, but also from the transition to throughput-type designs, which provide fewer cache bits per thread.

These many-core architectures struggle not only to provide sufficient main memory bandwidth per core/thread, but also to schedule high memory bus utilization efficiency. Server processors generally have one or two main memory controllers per chip, meaning that many cores share a single controller and a memory controller simultaneously sees requests from different work streams. In this context, locality is easily lost, and it becomes difficult to find and schedule spatially sequential accesses. Inefficient scheduling results in performance reductions and consumes unnecessary energy.

Most servers currently use JEDEC-standardized Double-Data-Rate (DDR) memory [2], so a fairly accurate understanding of memory bandwidth scaling can be obtained by looking at DDR trends. In terms of raw IO (Input/Output) speeds, DDR has continued to improve, with peak speeds doubling each generation (400Mbps DDR, 800Mbps DDR2, 1600Mbps DDR3). IO frequencies continue to scale, but other key parameters, such as reading a memory cell or turning a bus around from a write to a read

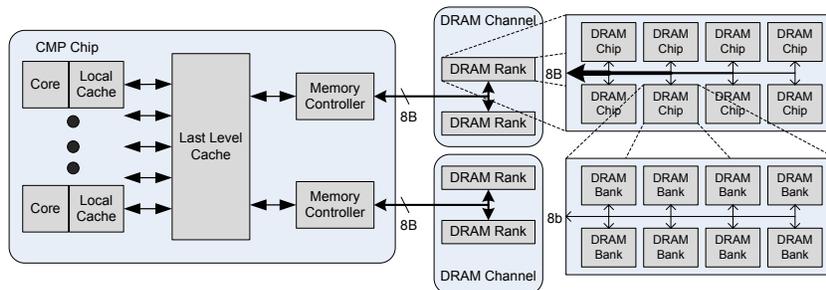


Figure 2: Baseline CMP and memory system

operation, are not scaling at comparable rates. At higher signaling rates, the electrical integrity of busses becomes much more difficult to maintain – both within the DRAM chips and across the main memory path to/from the processor. This results in a complex set of timing parameters which dictate that gaps be inserted when the access stream transitions from a write to a read or vice-versa, significantly degrading effective memory bandwidth. This problem has worsened with each memory generation. For example, tWRT, the Write-to-Read Turnaround delay, has stayed within a range of 5-10ns for DDR through DDR3. Therefore, as DRAM IO frequencies increase, the number of cycles wasted between each access grows. Figure 1, calculated from JEDEC parameters, shows the dramatic impact of these mandatory timing delays on effective bandwidth: even with perfect scheduling, utilization of a single memory rank can be as low as 25-40% for high-frequency DDR. Clearly, this trend cannot continue – in the many-core era, we must find ways to improve not only raw memory bandwidth, but also memory bandwidth efficiency.

Memory efficiency improvements in highly threaded systems have primarily been addressed in the past with various mechanisms to reorder requests received by the memory controller. Work on *Fair Queuing Memory Systems* by Nesbit et al. [18] utilizes *fair queuing* principles at the memory controller, to provide fairness and throughput improvements. *Parallelism-Aware Batch Scheduling* [17] alleviates the same problem by maintaining groups of requests from various threads and issuing them in batches to the controller. While these techniques are useful in improving read scheduling, they do not directly address the memory traffic initiated by write operations. The *Eager writeback* technique [9] takes a step in dealing with write bandwidth to memory, but its approach has minimal communication between the *last-level cache* and the *memory controller*.

Given more write queuing resources, the memory controller can more efficiently determine priorities, pack commands into page mode accesses, and efficiently schedule write requests to the memory bus. The challenge is how to efficiently expand the memory controller’s write queuing resources. We propose a *Virtual Write Queue* which leverages the existing last-level cache to expand the memory controller’s visibility, allowing improved performance and reduced power consumption. Our proposed scheme uses *Scheduled Writebacks* (writeback operations initiated by the DRAM scheduler) to effectively utilize the available memory bandwidth. This enables significantly longer write bursts that amortize bus turnaround times, and decrease high latency conflicts with read operations.

Based on full-system, cycle-accurate, simulations performed on SIMICS [22] and GEMS [13], *Virtual Write Queue* demonstrates an average 10.9% system throughput improvement on the memory intensive workloads of the SPEC CPU2006 Rate suite [23], when configured using 1 memory rank. In addition, the proposed

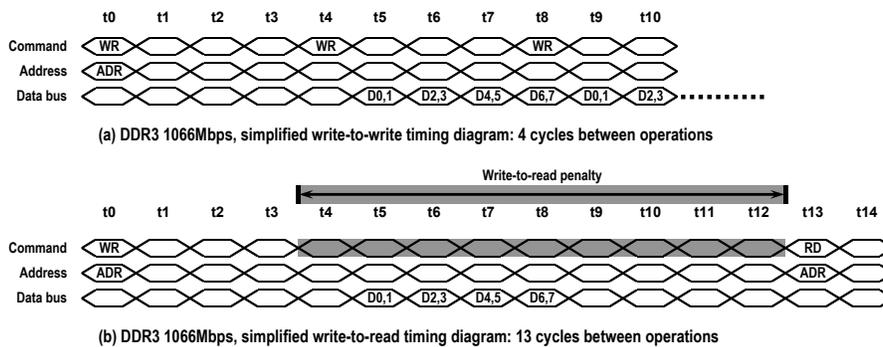
scheme is able to achieve a memory power reduction of 8.7% across all the SPEC CPU2006 suite. Both throughput and power reductions are demonstrated over a baseline implementation that incorporates a *First-Ready, First-Come-First-Served* (FR\_FCFS) [21] memory controller augmented with the *Eager writeback* technique [9] (FR\_FCFS+EW). Overall, the *Virtual Write Queue* requires a modest 0.3% hardware overhead by heavily reusing the existing structures and functionality of a typical high performance server design.

To the best of our knowledge this is the first proposed scheme that directly coordinates *memory controller scheduling* with *cache writeback traffic*. While the motivation and evaluations presented in this paper are based on DDR-DRAM, it is important to note that this paper tackles a problem which will only become worse as new memory technologies are incorporated into proposed hybrid and tiered main memory systems. Phase-change memory [19], NOR and NAND Flash, and other non-DRAM storage technologies have noticeably longer write times and larger page sizes (so as to amortize access costs). These factors aggravate existing memory inefficiencies such as *write-to-read* gaps, and make it even more crucial that efficient access opportunities, such as page mode accesses, be made apparent to the memory controller. The techniques in this paper can be extended easily for new technologies, and would be highly effective for improving performance of non-DRAM main memory subsystems.

The rest of the paper is organized as follows. Section 2 describes the typical organization of a DDR DRAM, and in Section 3 we characterize key aspects of memory system behavior. Section 4 elaborates on the proposed *Virtual Write Queue*. Finally, Section 5 includes results and evaluation, followed by a summary of related work in the literature in Section 6.

## 2. MAIN MEMORY BACKGROUND AND TERMINOLOGY

Figure 2 illustrates our baseline CMP and memory system. To maximize memory bandwidth and memory capacity, server processors have multiple *memory channels* per chip. As Figure 2 shows, each channel is connected to one or more *DIMMs* (memory cards), each containing numerous *DRAM chips*. These DRAM chips are arranged logically into one or more *ranks*. Within a rank, each DRAM chip provides just 4-8 bits of data per data cycle, and a rank of 8-16 DRAM chips works in unison to produce eight bytes per data cycle. The *DRAM burst-length* (BL) specifies an automated number of data beats that are sent out in response to a single command, commonly 8 data beats, to provide 64Bytes of data. From the time of applying an address to the DRAM chips, it takes about 24ns (96 processor clocks at 4GHz) for the first cycle of data, but subsequent data appear at high frequency, closer to 2-3 processor clocks.



**Figure 3: Write-to-read turnaround noticeably worsens command latency and databus utilization.  $t_{WPRST}$  [2] further worsens turnaround, but has been removed for simplicity**

Internally, DRAM chips are partitioned into *banks* that can be accessed independently, and banks are partitioned into *pages*. DRAM *page mode* provides the opportunity to read or write multiple locations within the same DRAM page more efficiently than accessing a new page. Page mode accesses require that the memory controller find requests with adjacent memory addresses, but are executed with fewer timing constraints, and incur lower per-access power than non-page mode accesses.

DRAM chips are optimized for cost, meaning that technology, cell, array, and periphery decisions are made with a high priority on bit-density. This results in devices and circuits which are slower than standard logic, and chips that are more sensitive to noise and voltage drops. A complex set of timing constraints has been developed to mitigate each of these factors for standardized DRAMs, such as outlined in the JEDEC DDR3 standard [2]. These timing constraints result in "dead times" before and after each random access; the processor memory controller's job is to hide these performance-limiting gaps through exploitation of parallelism.

### 3. CHARACTERIZATION OF MEMORY SYSTEM BEHAVIOR

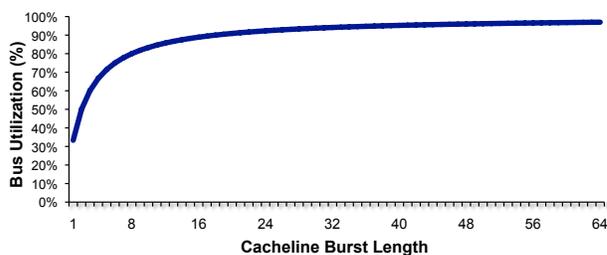
In order to understand memory system opportunities, we investigated the particular challenges associated with DRAMs in server systems. In our analysis we identified three characteristics as the most important factors in addressing memory interface utilization from the perspective of write memory traffic: a) *bus turnaround penalty*, b) *page mode opportunities*, and c) programs' *bursty behavior*.

#### 3.1 Bus turnaround penalty

As described in the introduction, memory IO frequencies have been improving (resulting in raw bandwidth increases), but timing constraints related to signaling and electrical integrity have, for the most part, remained constant. This is especially well illustrated by  $t_{WRT}$ , which defines the minimum delay from the completion of a write to the initiation of a read at the same DRAM rank.  $t_{WRT}$  is 7.5ns on DDR3 devices – at 4GHz, this is 30 CPU cycles – a very significant penalty for any reads issued after a write.

Figure 3 conceptually illustrates DDR3 write command timings. Successive writes can be issued back-to-back, realizing 100% data bus utilization (Figure 3.a). On the other hand, when a read follows a write to the same device, the read must not only wait for the write's completion, but also for the bus turnaround time to elapse. This adds noticeable latency to the read operation (Figure 3.b), and results in dismal data bus usage: 31%. For server-class 1066Mbps

DDR3, the extra nine DRAM cycles between a write and read amount to a  $\approx 66$  cycle read penalty at 4GHz.



**Figure 4: Bus utilization based on cache burst lengths**

Memory scheduling efficiency is thus heavily influenced by the mixing of read and write operations. Timing gaps are required when read and write operations are intermingled in a continuous memory access sequence, but with many application streams and limited queuing resources, these turnarounds are generally difficult for the memory controller to avoid. As shown in Figure 4, memory data bus utilization can be greatly improved by significantly increasing the number of consecutive read/write operations the memory controller issues before switching the operation type on the bus. For example, if the scheduler can manage 32 reads/writes per scheduling block, utilization grows to 94%. Throughout the paper, we refer to a stream of consecutive reads or writes as a *cacheline burst* to memory.

#### 3.2 Page mode opportunities

While DRAM devices output only 16-64 bits per request (depending on the DRAM type and burst settings), internally, the devices operate on much larger,  $\approx 2$ Kbit, pages (sometimes referred to as *rows*). Each random access causes all 2Kbits of a page to be read into an internal buffer, followed by a "column" access to the requested sub-block of data. Since the time and power overhead of the page access have already been paid, accessing multiple columns of that page decreases both the latency and power of subsequent accesses. These successive accesses are said to be performed in *page mode*. With page mode, latency is approximately one-half that of accesses to different, random pages. Active power reduction is realized through elimination of extra page reads and extra precharge page writebacks (because DRAM page read is destructive, the data must be restored). Additional details on DRAM architectures can be found in Jacob et al. [5].

Using a DDR3 memory power calculator from Micron Technology [16], we obtained the DRAM chip power characteristics shown

in Table 1 for various page mode access rates. We have chosen a server-type configuration, to match the environment used for the class of benchmarks studied in this paper (2Gb DDR3 DRAMs, 1.5V, 1066Mbps, 2:1 read:write ratio, 45% DRAM utilization, multi-rank DIMMs); we also include termination power. Memory power varies widely, depending on the class and capacity of the memory subsystem (frequency and number of ranks, for example). As shown, activation power is greatly reduced with page mode. Total DRAM power can be reduced by 1/3 if operations are executed as four references for each activate.

**Table 1: DRAM power estimation based on [16]**

% Page Mode	Activate Power (mW)	Total Power (mW)	% DRAM Power Saved
0%	196.9	450.8	Baseline
25%	147.2	401.1	11%
50%	98.1	352.0	22%
75%	49.0	303.9	33%

Page mode DRAM access greatly improves both memory utilization and power characteristics, but the optimization possibilities for read and write operations are significantly different. Reads are visible as the program (or a prefetch engine) generates them; this should enable spatial locality in reference sequences to be executed in page mode. Aligning load references into a page mode sequence often increases latency for critical operations, since they are delayed by younger page hits. Page mode read opportunities are thus commonly limited by conflicts with latency minimization mechanisms [17]. In contrast, write operations (in the common writeback cache policy) are generated as older cache lines are evicted to make room for newly allocated lines. As such, spatial locality at eviction time can be obscured through variation in set usage between allocation and eviction, as shown in Figure 5.

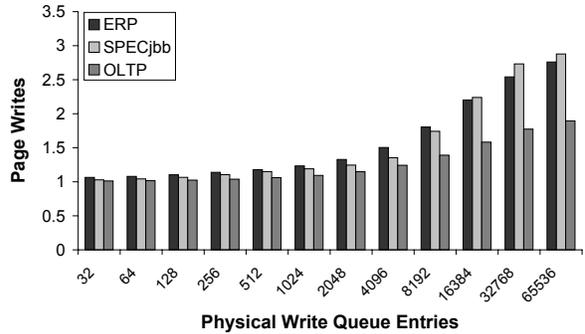
Figure 5 shows the total number of page mode writes possible for various workloads, for a range of memory controller write queue sizes. The workloads and simulation environment for this characterization data are described in Section 5. For practical write queue sizes, such as 32 entries, there is essentially no page mode opportunity (approximately one write possible per page activate). That stated, a large amount of spatial locality is contained within the various cache levels of the system, but today’s CPU-centric caches do not give the memory controller visibility into this locality. Significantly larger memory controller write queues, though impractical, would provide the needed visibility and enable significant page mode opportunities.

### 3.3 Bursty behavior

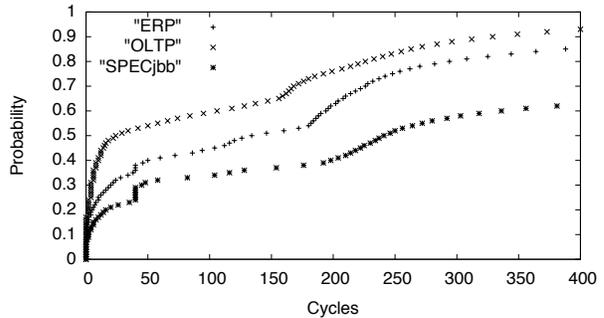
Most programs exhibit bursty behavior. At the last-level cache, this results in phases when a large number of load misses must be serviced. Common cache allocation/eviction policies compound this effect, as bursts of cache fills create bursts of forced writebacks, thus clustering bursts of reads with bursts of writes.

Figure 6 shows the distribution of time between main memory operations for three workloads. The workloads and analysis are discussed in more detail in Section 5. Note that 20-40% of all memory requests occur with less than ten cycles delay after the previous memory operation, while the median can be in the hundreds of cycles and many requests take significantly longer.

Over-committed multi-threaded systems have always experienced some degree of cache thrashing, as workloads evict one another’s data, and threads re-warm their local cache. However,



**Figure 5: Page writes per activate vs. number of Physical Write Queue entries for commercial workloads**



**Figure 6: Distribution of time between memory requests for commercial workloads**

this is no longer a problem seen only in large-scale SMP systems. Instead, this phenomenon is inherent to today’s virtualized computing environments as disparate partitions share a physical CPU. If write operations can be executed early, bursts of read operations execute with lower combined bandwidth demands. Ideally, the memory controller will always have write operations ready to be sent to idle DRAM resources.

## 4. A COORDINATED CACHE/MEMORY SYSTEM DESIGN

The characteristics identified in the previous section could be addressed if the memory controller were to have increased elasticity and visibility for scheduling write operations to memory. Typical memory controllers contain 10’s of queued write operations [7]. Queue structures are costly in terms of area and power, and the size of the write queue is a critical parameter for overall memory performance. In our solution, we propose that the set members close to the LRU position of the last-level cache function as a very large write queue (e.g., 64k effective entries for the lower one-quarter of a 16 MB last-level cache). We refer to the LRU section of the last-level cache as the *Virtual Write Queue*, in that we overload and repurpose the usage of this region. We show that a coordinated cache/memory policy based on direct management from the *Virtual Write Queue* mitigates each of the challenges described in Section 3, increasing the performance of the memory subsystem. Specifically:

- 1. Bus turnaround penalty avoidance:** Through a *Scheduled writeback* policy, we efficiently drain more pending write operations, minimizing the probability of interleaved read and write operations, which incur costly scheduling penalties ( $\approx 66$  processor cycles @ 4GHz).

2. **Harvesting page mode opportunities:** Directed cache lookups, to a broad region of the LRU space, enable harvesting of additional writes to be executed in page mode at higher performance and lower power.
3. **Burst leveling:** With the *Virtual Write Queue*, the ability to buffer  $\approx 1000x$  more write operations than in a standard memory controller enables significantly greater leveling of memory traffic bursts.

Figure 7(a) shows the *Virtual Write Queue*, which logically consists of some LRU fraction of the last-level cache (1/4 in this example); the *Physical Write Queue* in the memory controller; and the added coordinating control logic and structures. We refer to the structure as virtual because no additional queueing structures are added. The overall area cost is small:  $\approx 0.3\%$  overhead over a typical cache directory implementation (see Section 4.7).

**Scheduled Writeback:** Traditional writeback cache policies initiate memory writes only when a cache fill replaces an LRU cacheline. We refer to this as *forced writeback*. There are two problems with this policy. First, writes are only sent to the memory controller at the time of cache fills, so idle DRAM cycles cannot be filled with writes. This was addressed with *Eager Writeback* [9], where cachelines are sent to the memory controller when it appears to be idle (an empty write queue is detected). The second problem deals with the mapping of write locations to DRAM resources. Since the memory controller is aware of each DRAM's state, it would ideally decide which writeback operation can be executed most efficiently. With *Eager Writeback*, this selection is made by the cache, without knowledge of what would be best for DRAM scheduling. We introduce *Scheduled Writeback* to solve this problem. With *Scheduled Writeback*, the memory controller can direct the cache to transfer lines that map to specific DRAM resources.

As shown in Figure 7(b), the primary microarchitectural addition over traditional designs is the Cache Cleaner. The Cache Cleaner orchestrates *Scheduled Writebacks* from the last-level cache to the *Physical Write Queue*. While the cache and the *Physical Write Queue* are structurally equivalent to traditional designs (and thus hardware overhead is minimal), the logical behavior is significantly altered. This is reflected in the distribution of *dirty* lines in the system. Specifically, dirty lines have been cleaned from the lower section of the cache, and the *Physical Write Queue* is maintained at a higher level of fullness, with an ideal mix of commands with respect to scheduling DRAM accesses. In addition, typical memory controllers decide write operation priority based on only the *Physical Write Queue*, whereas this system uses the much larger *Virtual Write Queue*. The *Physical Write Queue* becomes a directly managed staging buffer, of the now much larger window of write visibility.

## 4.1 High Level Description of Coordinated Policies

At steady state, the *Physical Write Queue* is filled to a defined  $\approx$ full level with a mix of operations to all DRAM resources. This level is chosen to keep the queue as full as possible, while retaining the capacity to receive cache writebacks. *Scheduled Writebacks* can vary in length, depending on the number of eligible lines found in the same DRAM page; the write queue must maintain capacity to absorb these operations.

The DRAM scheduler executes write operations based on the conditions of the DRAM devices, read queue operations, and the current write priority. Write priority is determined dynamically

depending on the fullness of the *Virtual Write Queue*. As write operations are executed to DRAM, the fullness of the *Physical Write Queue* is decreased, so the Cache Cleaner refills the *Physical Write Queue* to the target level. The Cache Cleaner will search the last-level cache *Virtual Write Queue* region for write operations to the desired DRAM resource. This DRAM resource is chosen in two ways: 1) If the memory controller attempts a burst of operations to a specific rank, an operation mapping to that rank will be sent; 2) alternately, if no burst is in progress, the *Physical Write Queue* will be rebalanced by choosing the rank with the fewest operations pending. This maintenance of an even mix of operations to various DRAM resources enables opportunistic write execution, in that a write is always available to any DRAM resource that becomes idle.

As part of the Cache Cleaner function, we harvest additional writes which map to the same DRAM page as the write selected by the Cache Cleaner for writeback. This is accomplished via queries to cache sets which map to the same DRAM page (Section 4.3). In our system, we define groups of such cache sets based on the cache and DRAM address translation algorithms. In our evaluation we found groups of four cache sets to be an ideal size.

Upon completion of the *Scheduled Writebacks*, the *Physical Write Queue* once again contains an ideal mix of operations to be scheduled. While we have described a sequence of operations, in practice, the structure can concurrently operate on all steps, accommodating periods of high utilization.

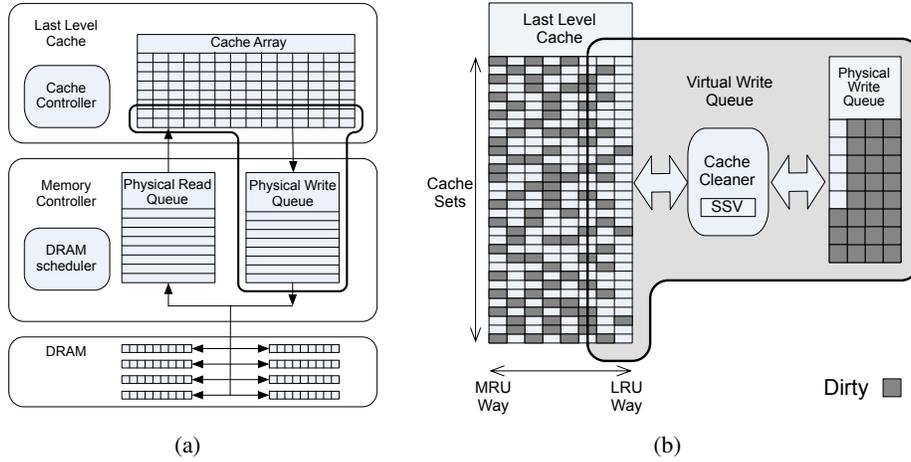
## 4.2 Physical Write Queue Allocation

As previously described, the highest barrier to efficient utilization of a main store (whether DRAM or future technologies) is the transition between read and write operations. In addition to write-to-read turnaround, alternating between different ranks on the same bus can introduce wasted bus cycles. To have good efficiency, DRAM banks must additionally be managed such that operations to the same bank, but to different pages, are avoided. These characteristics motivate creation of long bursts of reads or writes to ranks, while avoiding bank conflicts. The *Physical Write Queue* allocation scheme addresses the formation of write bursts.

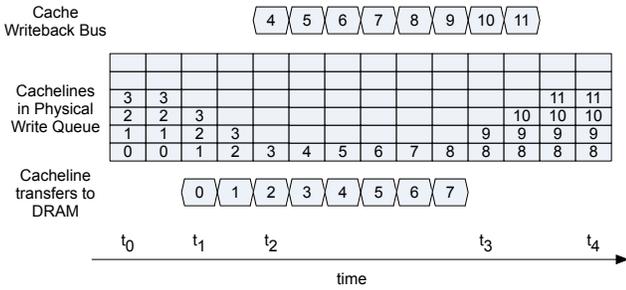
A key aspect of the *Virtual Write Queue* solution is its two-level design. Since writes can only be *executed* from the *Physical Write Queue*, our *Scheduled Writebacks* must expose parallelism of the *Virtual Write Queue* into the *Physical Write Queue* to achieve the highest value from last-level cache buffering. This is accomplished by maintaining the best possible mix of operations in the *Physical Write Queue*, given what is visible in the entire *Virtual Write Queue* structure.

We accomplish this in two ways. First, we create the capability to opportunistically execute *write operations* to any rank. In this way, we can react to a temporarily idle rank with a burst of writes at any moment. Extending this idea, we must maintain several writes to each rank which can be executed without idle cycles. Ideally, we would maintain many writes to the same DRAM pages. When it is not possible to maintain accesses targeting the same rank and page, we find operations to the same rank, but a different bank. For a write burst to a rank that is longer than what can be stored in the *Physical Write Queue*, we directly generate *Scheduled Writebacks* in concert with execution of writes, such that we can overlap the cache writeback latency with the *Physical Write Queue* transfers to memory. Once the initial latency of the first cache writeback has passed, our cache has the required bandwidth to maintain a busy DRAM bus.

An example timing diagram for this *Virtual Write Queue* function is shown in Figure 8. In this example, the *Physical Write Queue* initially contains four cachelines which map to a target rank



**Figure 7: Memory controller and DRAM for a) a typical system with separate read and write queues, and b) the proposed *Virtual Write Queue***



**Figure 8: Virtual Write Queue timing diagram of operation**

(cachelines 0 to 3 in the first column). At  $t_0$ , the scheduler initiates an eight cacheline write burst. While the initial four cachelines of data are available in the write buffer, the remaining lines must be transferred from the last-level cache using the *Scheduled Writeback* mechanism. In this case, a request is made at  $t_1$  to the Cleaner logic coincident with the initiation of the writes to memory. To maintain back-to-back transfers on the DRAM bus, the Cache Cleaner must be able to provide data within the delay of the transfer of data from the *Physical Write Queue* to main memory. For example, in a system using DDR3 1066 memory with a burst length (BL) of 8, each cacheline requires  $\approx 8$ ns to be transferred (*i.e.*,  $\approx 32$  CPU cycles). Thus the Cleaner must be able to provide a cache line within 32ns, assuming four cachelines stored in the *Physical Write Queue*. Our design analysis shows that this is easily achieved for typical last-level cache latencies (10 ns measured on Intel I7 965 [4]). In the example of Figure 8, we show that the first writeback data, cacheline 4, arrives at time  $t_2$ . At this point, the *Physical Write Queue* has been depleted of lines 0-3, and data is streamed from the last-level cache. As the eight-line write burst completes at time  $t_3$ , the remaining lines from the last-level cache transfer are used to refill the *Physical Write Queue*. At time  $t_4$ , the physical queue is once again full, and ready to execute another write burst.

### 4.3 The Cache Cleaner

To qualify as an efficient implementation, our scheme must (1) not interfere with the mainline cache controller; (2) be power efficient; and (3) be timely. Specifically, the *Cache Cleaner* must not affect hit rates or cause excessive access to cache directory and

data arrays; it must avoid excess reads of cache directory arrays for power efficiency reasons; and cache lines to be cleaned must be located in a timely manner. The Cleaner uses a Set State Vector to accomplish these goals.

#### 4.3.1 The Set State Vector (SSV)

While the cache lines in the *Virtual Write Queue* are contained within the state information of the cache directory itself, direct access is problematic. Specifically, cache directory structures are optimized for efficient CPU-side interaction. This CPU-centric directory organization conflicts with the access requirements of the Cache Cleaner. The Cache Cleaner would like to efficiently search across many sets, in search of *dirty* cache lines to be scheduled for writeback to the DRAM. We enable this efficient search with the addition of the *Set State Vector (SSV)*.

At the most basic level, the SSV is used as a structure to decouple the cache cleaner from the actual cache directory for the following reasons. First, the SSV provides a much denser access method. If the actual directory were used, many more bits would need to be read and interpreted, wasting power. This is shown in Figure 9(a), where each 8-way cache set maps to one SSV bit. In addition to the density gains, the organization of the SSV is tailored such that array reads return 64 bits for the specific DRAM resource being targeted (taking advantage of reading entire rows of the storage arrays). This is shown in Figure 9(b), where the set address is remapped to the DRAM rank and bank address. Lastly, the SSV avoids interference between the cache cleaner and the main cache directory pipeline by providing a dedicated read port for the cache cleaner.

The process of calculating the SSV bit is implemented as part of the cache access pipeline where all of the state and LRU information is available as part of the existing cache lookup. While there is additional power in accessing the SSV array, the structures are only a fraction of the size of the actual directory, thus the additional power is not significant.

Each SSV entry contains the *dirty data criticality* of each set. For our system, we define sets with *dirty* data in the oldest 1/4 of the cache as critical. We derive this criticality from the LRU bits present in the cache pipeline. Our system uses an 8-way true LRU algorithm implemented with 28 bits for each cache set. These 28 bits are used to define pairwise age between every combination of the eight members ( $7+6+5+4+3+2+1=28$  bits). The LRU distance is then calculated by adding the 7 direction bits for each way. If the

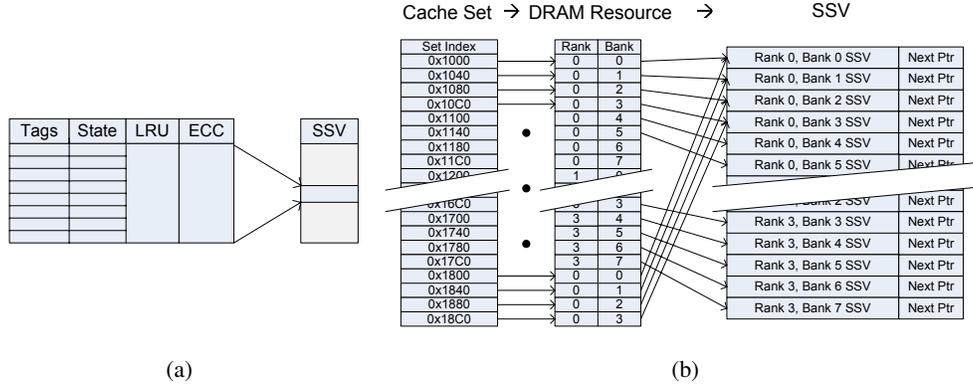


Figure 9: Set State Vectors (SSVs): a) directory set map to SSV entries, and b) mapping of cache sets to SSVs

distance is greater than the criticality distance (5 in our example), the SSV bit is set. While our system is based around true LRU, this concept can be extended to other replacement algorithms such as pseudo-LRU. For example the relative age in a tree based pseudo-LRU can be estimated by counting the number of pointers towards each way.

#### 4.3.2 Cleaner SSV Traversal

Adjacent entries in the SSV are not necessarily adjacent sets in the cache. The dense packing is based upon the mapping of the system address into the physical mapping on the DRAM’s channel/rank/bank resources. Adjacent entries in the SSV all map to the same channel/rank/bank resource.

A mapping example is shown in Figure 9(b). In this example we have a *closed-page* mapping for four ranks, each with 8 banks. In this case, every 32nd cache line maps to the same SSV. In general the mapping logic must be configured to match the DRAM mapping function; this is not a significant constraint, since mappings are known at system boot time. Our scheme requires all bits that feed the DRAM mapping function to be contained within the set selection bits of the cache. This enables not only the SSV mapping function, but also *page mode harvesting*. This restriction does not produce significant negative effects, since all bits above the system page size are effectively random, and large last-level cache sizes have several higher order bits available for more sophisticated mapping functions (which avoid power of two conflicts that are common in simple lower-order bit mappings) [24].

The SSV is then subdivided into regions for each channel / rank / bank configured in the system. The Cache Cleaner maintains a working pointer for each of these configured regions. As the Cache Cleaner receives writeback requests from the memory controller, the associated working pointer reads a section of the SSV (with the matching *Next Ptr* in Figure 9(b)). A set is selected, which will determine the specific set with which a writeback request will be generated. This request is sent to the cache controller to initiate the actual cache cleaning operation.

### 4.4 Read/write Priority Mechanism

To be most effective, our write queuing system must be able to dynamically adjust the relative priority between *read* and *write* operations. The system must be able to respond to the workload’s overall *read-to-write* ratio, in addition to being able to handle *bursts* of operations. A well-behaved system will always execute *reads* instead of *writes*, as *writes* only indirectly slow a program as queuing in the system becomes full. Given finite queuing, *writes* must at some point step in front of *reads*. This is an important aspect of our system, in that the *Virtual Write Queue* provides

much larger effective buffering capability. To efficiently manage this capacity we must react to the overall fullness of the *Virtual Write Queue*. The behavior of our system as the capacity nears full differs significantly from the previously-proposed *Eager Writeback* technique [9] (which essentially results in no benefits in sustained high bandwidth situations). This is illustrated in the following example.

Some workloads, such as *Stream* [14], require a high level of sustained, regular, bandwidth. *Stream* consumes sequential vectors of data that are not contained within the cache. These vectors are processed within loops with various *read-to-write* ratios (e.g., 1:1 R:W for *copy*, 3:1 for *triad*). Despite changes in the *read-to-write* ratio, the workloads’ memory bandwidth requirement is constant: every instruction executed must read from memory. There is therefore no inherent period of execution in which writes to memory can be hidden. Despite this constraint, memory utilization is significantly reduced if the memory bus is switched between read and write operations at the native requirements of the workload. For workloads with this type of behavior, improving performance requires that we force long write burst lengths, even when no idle bus slots are available. For the *copy* kernel, the native 1:1 ratio yields a memory bus utilization of 31%, while a burst of 32 cachelines can achieve 94% (Figure 4); a 300% improvement in memory efficiency.

**Priority Mechanism:** We base our *read-to-write* priority on a count of the *data criticality* bits set in the SSVs associated with each rank (8, one for each bank). The count is updated as bits are *set/reset* in the corresponding SSVs. We then utilize *high* and *low water marks* to trigger high priority *writes*. The *high water mark* is chosen such that we do not overflow the LRU of the cache with forced writebacks. In our evaluations we found a value of 4096 to be effective. This represents a half full *Virtual Write Queue*. Larger values resulted in overflow of some cache sets, decreasing the ability to control writebacks. The *low water mark* defines the number of consecutive cachelines to be written. We found a burst of 32 lines to be effective (*low water mark* of  $4064 = 4096 - 32$ ).

### 4.5 Write Page Mode Harvester Logic

The cache eviction mechanism is augmented to query adjacent lines in the directory, such that groups of requests within the same memory page can be detected and sent as a group for batch execution to the DRAM. When a line is pulled out of the cache array, we search the associated sets of the cache that contain possible page mode addresses. If the corresponding page mode addresses are found, these will be sent as a group to the memory controller, to be processed as a burst page-write command sequence. In our evaluation we found that three look-ups associated

**Table 2: Core and memory-subsystem parameters used in evaluation section**

Core Characteristics	Clock Frequency	Pipeline	Reorder Buffer /Scheduler	Branch Predictor
		4 GHz	30 stages / 4-wide fetch / decode	128/64 Entries
Memory Subsystem	L1 Data & Inst. Cache	L2 Cache	Outstanding Requests	Memory Bandwidth
	64 KB, 2-way associative, 3 cycles access time, 64 Bytes block size, LRU	16 MB, 8 ways associative, 12 cycles bank access, 64 Bytes block size, LRU	16 Requests per Core	16.6 GB/s
	Controller Organization	DRAM	Controller Resources	Virtual Write Queue
	2 Memory Controllers 1, 2, and 4 Ranks per Controller 8 DRAM chips per Rank	8GB DDR3-1066 7-7-7	32 Read Queue & 32 Write Queue Entries	2 LRU ways 4096 High & 4064 Low Watermark

with a block of four cachelines provided significant gains without excessive directory queries.

#### 4.6 Prevention of Extra Memory Writebacks

Since our system speculatively writes dirty data back to memory, there is some chance that extra memory write traffic is introduced. Specifically, if a store occurs to the data after it is cleaned, the cleaning operation is wasted. As a solution to this problem, we add additional cache states to indicate a line was once dirty, but has been cleaned. Lines in *Cleaned* states are excluded from being cleaned a second time. A complete extension to the MOESI protocol would require *Cleaned* version of all four valid states. This presents additional overhead in that the total number of states would reach nine. Since MOESI systems require three state bits of encoding, three unused state encodes are available. To avoid the overhead of adding a fourth state bit, we choose to exclude the *Shared Cleaned* state, maintaining the same state overhead as standard MOESI. Our justification for the exclusion of *Shared Cleaned* is best explained through the state transitions shown in Table 3. In the table, the *Cleaned* states are represented with a lowercase *c*, e.g. *Modified Cleaned* as  $M_c$ .

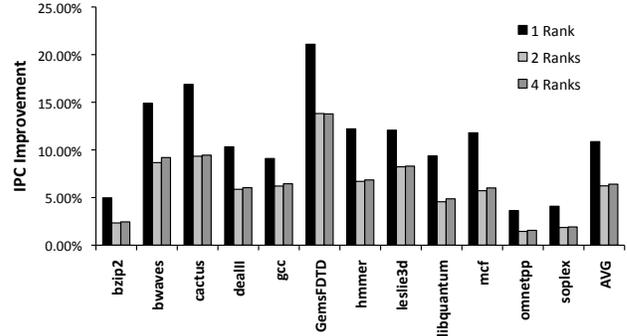
**Table 3: Extra coherence protocol transitions introduced to prevent extra memory writebacks**

#	Initial State	Event	Next State	Comment
1	M	Clean	$E_c$	Scheduled Writeback
2	S	Store	M	M If no $O_c$ in system
3	S	Store	$M_c$	$M_c$ If $O_c$ in system
4	$M_c$	Snooped Read	$O_c$	
5	$O_c$	Store	$M_c$	
6	$O_c$	Snooped Read	$O_c$	
7	$E_c$	Store	$M_c$	
8	O	Clean	S	Disallowed due to lack of $S_c$
9	$E_c$	Snooped Read	S	Loss of cleaned Information

There are two cases of potential transitions into the *Shared Cleaned* state ( $S_c$ ). In row 8 of Table 3, we show the case of an *Owned* ( $O$ ) line that if cleaned would transition to *Shared Cleaned*. In our simulations we do not clean *Owned* lines thus this case is avoided. In row 9 we show the case of a line in *Exclusive Cleaned* state ( $E_c$ ), where a read operation is snooped. Here the *Exclusive Cleaned* line must transition to an *Shared* state. Since the *Cleaned* modifier is not required for coherence, we simply use the traditional *Shared* state in this case. In our analysis we observed no degradation due to this policy.

#### 4.7 Overall Overhead Analysis of Virtual Write Queue

The actual storage overhead of the proposed *Virtual Write Queue* is limited to the overhead for the SSVs. The rest of the scheme


**Figure 10: IPC improvements of Virtual Write Queue over prior work (FR\_FCFS + Eager) [21, 9]**

primarily reuses existing structures in the last-level cache and memory controller of a typical CMP system. All of the remaining structures added to the last-level cache controller and memory controller are negligible in size compared to the storage required for the SSVs. We evaluate the overhead of the SSV by comparing it to the cache directory. For a 16MB 8-way associative cache used in our analysis, each cache set requires 346bits (see Figure 9(a):  $8 * 32$ bits Tag bits,  $8 * 3$ bits State Bits, 28 LRU bits and 38 ECC bits). Since we add only one SSV bit per cache set, our overhead is approximately  $1/346 \approx 0.3\%$  (4Kbytes of storage for the SSV compared to 1384Kbytes for the cache directory).

## 5. EVALUATION

To evaluate the effectiveness of the proposed *Virtual Write Queue*, we used Simics [22], extended with the Gems toolset [13], to simulate cycle-accurate out-of-order processors and a detailed memory subsystem. We configured our toolset to simulate an 8-core SPARCv9 CMP with 8GB of main memory. The memory subsystem model includes an inter-core last-level cache network that uses a directory-based MOESI cache coherence protocol along with a detailed memory controller. Both the last-level cache and the memory controller were augmented to support a baseline memory controller and the proposed *Virtual Write Queue*. Our baseline implementation simulates a *First-Ready, First-Come-First-Served* (FR\_FCFS) [21] memory controller with the addition of *Eager writebacks* [9], referred to as *FR\_FCFS+EW* in our evaluation. Enhancements beyond FR\_FCFS, such as PARBS [17] and Fair-Queueing [18] focus on memory *read* fairness and throughput improvement in heterogeneous environments. In this study we evaluated improvements to the *write* scheduling using homogeneous workloads. System improvements targeting *read* scheduling are non-conflicting and can be used in combination with *Virtual Write Queue*. Therefore a baseline system of FR\_FCFS with *eager writebacks* (FR\_FCFS+EW) allows us to evaluate our

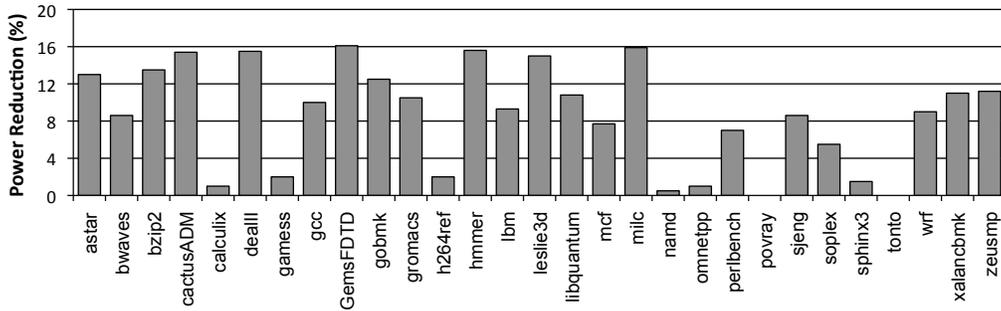


Figure 11: Power reductions achieved by *Virtual Write Queue* for SPEC CPU2006 Rate

proposed scheme. We use the SPEC CPU2006 Rate scientific benchmark suite [23], compiled to the SPARC ISA with full optimizations (peak flags). Each benchmark is fast-forwarded for 4 billion instructions to reach its execution steady-state phase. We then simulate for the next 100M instructions to warm up the last-level cache and our memory controller structures; followed by a final 100M instructions used in our evaluation. Table 2 includes the basic system parameters.

### 5.1 System Throughput Speedup Analysis

Cycle-accurate simulations of SPEC CPU2006 Rate showed that the *Virtual Write Queue* enables significant throughput gains for workloads with high memory bandwidth requirements. These gains over the baseline FR\_FCFS+EW [21, 9] system is shown in Figure 10. We show speedups for three memory bus configurations (1, 2, and 4 DRAM ranks per channel). SPEC workloads not listed in the figure did not show any measurable change in performance. As expected, workloads with high memory utilization showed benefits due to reduction in bus penalties by forming long back-to-back write bursts. The largest speedup is observed on the single rank system. In this case, the controller does not have other ranks to which to send requests, and the “write-to-read-same-rank” penalty is incurred at every bus turnaround. For the 2-rank system, we show smaller gains, since the baseline system is able to schedule around “write-to-read-same-rank” penalties in many cases. In that case, delays due to rank-to-rank transitions become more important. The performance of the 4-rank system is very close to that of the 2-rank since the controller incurs fewer “write-to-read-same-rank” penalties, but generates more frequent rank-to-rank transitions. Overall, the *Virtual Write Queue* achieved average improvements of 10.9% in throughput when configured with 1 rank, while for the cases of 2 and 4 ranks the IPC improvements were found to be 6.4% and 6.7%, respectively.

### 5.2 Page Mode Analysis

Using the *Virtual Write Queue* we see significant increases in the amount of page mode write operations over the baseline FR\_FCFS+EW [21, 9] system. Our full-system simulation shows an average of 3.2 write accesses per page – this contributes to system throughput gains (as observed in Section 5.1) and memory power reduction. While the throughput gains are only observed in high bandwidth workloads, the power reductions are more universal. In Figure 11 we show the DRAM power reduction for each workload estimated using the Micron power estimator [16]. Overall, we observe an average DRAM power reduction of 8.7%. As shown in [20], main memory can be a significant portion of total system power. For *stream*, Rajamani *et al.* indicate that memory power can be 48% of high performance system power, even after accounting for supply losses. The 11-15% power saved for half of

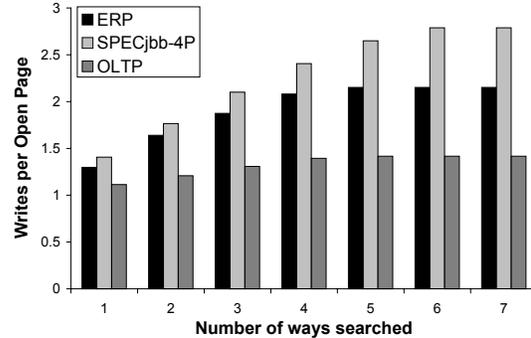


Figure 12: Writes per open-page for varying number of cache ways allocated to the *Virtual Write Queue*. The *Virtual Write Queue* exploits the LRU ways of the last-level cache to virtually expand the write queue.

the workloads through the page mode write scheduling of *Virtual Write Queue* would thus result in a 5-7% system-level savings.

To further evaluate the page mode behavior of the *Virtual Write Queue* we performed a set of simulations using three diverse commercial workloads. Due to the complexity and size of the commercial workloads, a detailed evaluation using a cycle-accurate, full system simulator is prohibitively expensive. As a solution, for commercial workloads we utilize trace-based cache simulations. The first workload is an *On-Line Transaction Processing* (OLTP) workload driven by hundreds of simulated individual clients generating remote transactions against a large database. The second workload represents a typical *Enterprise Resource Planning* (ERP) workload. As with the OLTP workload, the ERP workload was driven by simulated users who sent remote queries and updates to the database. Finally, the third workload is SPECjbb2005 benchmark [23] that targets the performance of a three-tier client/server system with emphasis on the middle tier. We performed the trace-based simulations using an in-house cache simulator augmented to model the *Virtual Write Queue*.

We augmented our cache simulator to monitor the total page mode writes that were possible when varying the number of cache ways that were allocated to the *Virtual Write Queue*. The simulation results are shown in Figure 12. Whenever a dirty line was evicted from the cache, the last  $N$  ways of the cache were checked for other dirty lines that mapped to the same DRAM page. As expected, there is a steady increase in page mode opportunity as we increase the number of ways considered, with significant increases in 2 ways, and diminishing returns after considering more than 4 ways, or half the cache. The increased performance from considering more ways of the set must be balanced against the overhead of additional writes introduced by the cleaning. Though

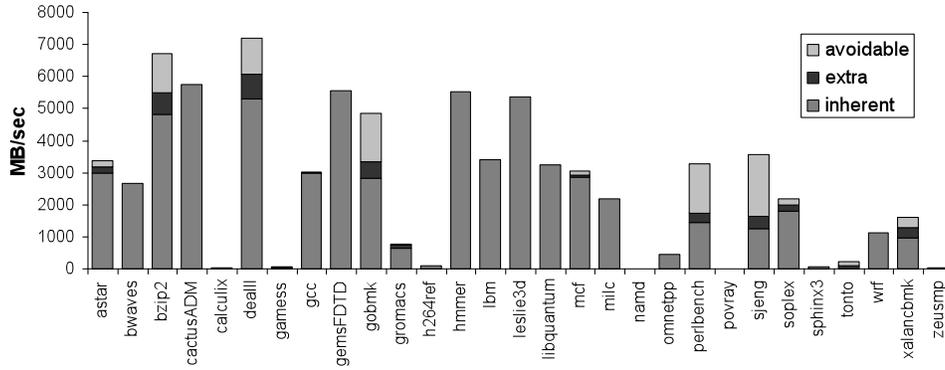


Figure 13: Extra Writeback avoidance for SPEC CPU2006

not shown, it is worth noting the implied difference in performance results across the three workloads. The OLTP workload sees limited benefit, while the ERP and SPECjbb see much larger opportunities.

### 5.3 Prevention of Extra Memory Writeback Analysis

As described in Section 4.6, the *Virtual Write Queue* mechanisms has the potential to generate additional writeback traffic for cases where a cleaned line is modified after its speculative writeback to the memory. In this section we evaluate the magnitude of the problem, and the solution presented in Section 4.6. Through our cycle-accurate simulations, we classified the writebacks to memory into the following categories: a) *Inherent*: normal bandwidth that is not created by speculative writebacks, b) *Extra*: bandwidth created through speculative writeback that is *not* removed though our cache state enhancements, and c) *Avoidable*: Speculative bandwidth that is avoided with the addition of the proposed cache state enhancements. In Figure 13 we present the total memory bandwidth, in MB/sec, as was collected from our simulations. As shown, certain workloads, such as *sjeng*, see significant extra bandwidth that is eliminated using the cache state enhancements of Section 4.6. Note, our power estimates in Section 5.2 assume these cache state enhancements. In all cases, the power savings achieved at the DRAM using page mode offset the increase from extra traffic (since most of the extra traffic contains no bank activates). In addition, we performed a limited evaluation of commercial workloads to gauge how problematic this behavior is. Again, there are significant differences between the workloads. Using the lower 2-ways of LRU in the *Virtual Write Queue*, SPECjbb showed 1% increase in writebacks compared to an increase of 6% for ERP and 9% for OLTP.

## 6. RELATED WORK

**Multi-threaded aware DRAM schedulers:** Multi-threaded aware proposals have primarily addressed mechanisms to reorder requests received by the memory controller. Work on *Fair Queuing Memory Systems* by Nesbit et al. [18] utilizes *fair queuing* network principles on the memory controller to provide fairness and throughput improvements. *Parallelism-Aware Batch Scheduling* [17] alleviates the problem by maintaining groups of requests from the various threads in the system and issuing them in batches to the controller. A suite of work has considered minimizing prefetch impacts on overall memory subsystem performance, some of which consider prioritizing prefetch according to page mode opportunity [10] [11]. As previously described, the *Virtual Write*

*Queue*, which realizes performance and power savings by focusing on writes, is non-conflicting and can be combined with the previous read-oriented memory optimization approaches.

**System and DRAM interaction:** Proposals to improve interactions between the DRAM controller and other system components have been proposed in the following areas. The *Eager Writeback* technique [9] addresses breaking the connection between cache fills and evictions, but the approach has minimal communication between the last-level cache and the memory controller and thus misses performance and power opportunities which arise through knowledge of logical-to-physical address mapping and troublesome memory timing constraints. In the work evaluating ZettaRAM [25], *Eager Writeback* is shown to work synergistically with the more advanced write characteristics of the described memory technology. This is an example of how future memory technologies can benefit from more sophisticated write-back schemes. We expect the *Virtual Write Queue* to further improve the system characteristics. *SDRAM-aware scheduling* from Jang et al. [6] addresses management of the on-chip network such that requests are ordered with regard to memory efficiency.

**DRAM write-read turnaround:** For specialized applications, DRAMs have been offered with separated read and write IOs, allowing for high bus utilization, even for mixed read/write access streams [15]. Borkenhagen et al. [1], describe the problem of DRAM *write-to-read* turnaround delay, and recognize the need for cache / memory controller interaction to most effectively alleviate its performance effect. Borkenhagen et al. propose a *read predict* signal, which provides the memory controller early notice that a read may soon arrive at the memory controller. If *read predict* is asserted, the memory controller will not issue pending writes, to avoid incurring a *write-to-read* turnaround delay which would delay the read.

**DRAM power management:** Several approaches have been proposed for DRAM power management, however most leverage memory sleep states, rather than exploiting page mode power savings. [8] considers the power cost of opening and closing pages, and [12] proposes a *Page Hit Aware Write Buffer* (PHA-WB), a 64-entry structure residing between the memory controller and DRAM, which holds onto writes until their target page is opened by a read. The PHA-WB, however, was evaluated for a write-through cache, for which memory-level access locality will be much more apparent than in a writeback structure. Writeback caches provide significant improvements in available memory bandwidth and power consumption, so are thus a more realistic baseline for many-core server-class systems.

## 7. CONCLUSIONS

In this work we address the barriers to efficient DRAM operation, which is a key attribute of many-core architectures. We propose a novel approach that coordinates last-level cache and DRAM policies, which significantly improves both system performance and energy characteristics. We modify existing structures and mechanisms in order to allow greater sharing of system state across units, therefore, enabling better scheduling decisions.

Specifically, we expand the memory controller's visibility into the last-level cache, greatly increasing write scheduling opportunities. This enables several improvements in system behavior. We are able to increase page mode writes to DRAM, which both decreases power consumption and increases memory bus efficiency. The longer write bursts achieved through *Scheduled Writebacks* amortize bus turnaround penalties, increasing bus utilization. The larger effective write queue improves read/write priority determination in the DRAM scheduler, enabling burst read operations to proceed uninhibited by write conflicts for longer periods.

We demonstrate through cycle-accurate simulation that the proposed *Virtual Write Queue* scheme is able to achieve significant raw system throughput improvements (10.9%) and power consumption reductions (8.7%) with very low hardware overhead ( $\approx 0.3\%$ ). Overall, the *Virtual Write Queue* demonstrates that co-optimizations of multiple system components enable low-cost, high-yield improvements over traditional approaches.

## Acknowledgements

The authors would like to thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. The authors acknowledge the use of the Archer infrastructure for their simulations, and Kyu-Hyoun Kim for assistance in DRAM bus utilization calculations. This work is sponsored in part by the National Science Foundation under award 0702694 and CRI collaborative awards: 0751112, 0750847, 0750851, 0750852, 0750860, 0750868, 0750884, 0751091, and by IBM. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or IBM.

## References

- [1] J. Borkenhagen, B. Vanderpool & L. Whitley, "Read prediction algorithm to provide low latency reads with SDRAM cache," *US Patent 6801982*, 2004.
- [2] DDR3 SDRAM Standard, JEDEC JESD79-3, <http://www.jedec.org>, June 2007.
- [3] P. Glaskowsky, "High-end server chips breaking records," [http://news.cnet.com/8301-13512\\_3-10321740-23.html](http://news.cnet.com/8301-13512_3-10321740-23.html), Aug. 2009.
- [4] J. Hruska, "Nehalem by the numbers: The Ars review," <http://arstechnica.com/hardware/reviews/2008/11/nehalem-launch-review.ars/3>.
- [5] B. Jacob, S. Ng & D. Wang, "Memory systems: Cache, DRAM, disk," *Morgan Kaufmann Publishers Inc.*, USA, 2007.
- [6] W. Jang & D. Pan, "An SDRAM-aware router for networks-on-chip," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 800–805, 2009.
- [7] R. Kalla, B. Sinharoy & J. M. Tandler, "IBM Power5 chip: A dual-core multithreaded processor," *IEEE Micro*, vol. 24, no. 2, pp. 40–47, 2004.
- [8] N. Y. Ker & C. H. Chen, "An effective SDRAM power mode management scheme for performance and energy sensitive

- embedded systems," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 515–518, 2003.
- [9] H. Lee, G. Tyson & M. Farrens, "Eager writeback - a technique for improving bandwidth utilization," in *Proceedings of the International Symposium on Microarchitecture*, pp. 11–21, 2000.
- [10] W. Lin, S. Reinhardt & D. Burger, "Reducing DRAM latencies with an integrated memory hierarchy design," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 301–312, 2001.
- [11] J. Lin, H. Zheng, Z. Zhu, Z. Zhang & H. David, "DRAM-level prefetching for fully-buffered DIMM: Design, performance and power saving," in *International Symposium on Performance Analysis of Systems & Software*, pp. 94–104, 2008.
- [12] S. Liu, S. Memik, Y. Zhang & G. Memik, "A power and temperature aware DRAM architecture," in *Proceedings of the 45th Annual Design Automation Conference*, pp. 878–883, 2008.
- [13] M. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News (CAN)*, September 2005.
- [14] J. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.
- [15] Micron Technologies, Inc., "Exploring the RLDRAM II Feature Set," *Technical Report: TN-49-02*, 2004.
- [16] Micron Technologies, Inc., DDR3 SDRAM system-power calculator, revision 0.1, Mar. 2007.
- [17] O. Mutlu & T. Moscibroda, "Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers," *IEEE Micro* vol. 29, pp. 22–32, 2009.
- [18] K. Nesbit, N. Aggarwal, J. Laudon & J. Smith, "Fair queuing memory systems," in *Proceedings of the International Symposium on Microarchitecture*, pp. 208–222, 2006.
- [19] M. Qureshi, V. Srinivasan & J. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the International Symposium on Computer Architecture*, pp. 24–33, 2009.
- [20] K. Rajamani *et al.*, "Power Management for Computer Systems and Datacenters", tutorial at the *International Symposium on Low Power Electronics and Design (ISLPED)*, 2008.
- [21] S. Rixner, W. Dally, U. Kapasi, P. Mattson & J. Owens, "Memory access scheduling," in *Proceedings of International Symposium on Computer Architecture*, pp. 128–138, 2000.
- [22] Simics Microarchitect's Toolset, <http://www.virtutech.com>.
- [23] Standard Performance Evaluation Corporation, <http://www.spec.org>.
- [24] M. Valero, T. Lang & E. Ayguadé, "Conflict-free access of vectors with power-of-two strides," in *Proceedings of the International Conference on Supercomputing*, pp. 149–156, 1992.
- [25] R. Venkatesan, A. AL-Zawawi & E. Rotenberg, "Tapping ZettaRAM for Low-Power Memory Systems," in *11th International Symposium on High-Performance Computer Architecture*, pp. 83–94, 2005