Preventing PCM Banks from Seizing Too Much Power

Andrew Hay[⋆] Karin Strauss[†◇] Timothy Sherwood[‡] Gabriel H. Loh[†*] Doug Burger[†◇]

[⋆]Dept. of Comp. Science
University of Auckland
Auckland, NZ
andrewh@cs.auckland.ac.nz

[†]Microsoft Research
Microsoft, Inc.
Redmond, WA, USA
kstrauss@microsoft.com

[◇]Dept. of Comp. Science and Engineering
University of Washington
Seattle, WA, USA
sherwood@cs.ucsb.edu

[‡]Dept. of Computer Science
University of California, Santa Barbara
Santa Barbara, CA, USA
gabe.loh@amd.com dburger@microsoft.com

ABSTRACT

Widespread adoption of Phase Change Memory (PCM) requires solutions to several problems recently addressed in the literature, including limited endurance, increased write latencies, and system-level changes required to exploit non-volatility. One important difference between PCM and DRAM that has received less attention is the increased need for write power management. Writing to a PCM cell requires high current density over hundreds of nanoseconds, and hard limits on the number of simultaneous writes must be enforced to ensure correct operation, limiting write throughput and therefore overall performance. Because several wear reduction schemes only write those bits that need *to be written, the amount of power required to write a cache line back to memory under such a system is now variable, which creates opportunity to reduce write power. This paper proposes policies that monitor the bits that have actually been changed over time, as opposed to simply those lines that are dirty. These polices can more effectively allocate power across the system to improve write concurrency. This method for allocating power across the memory subsystem is built on the idea of "power tokens," a transferable, but time-specific, allocation of power. The results show that with a storage overhead of 4.3% in the last-level cache, a power-aware memory system can improve the performance of multiprogrammed workloads by up to 84%.*

Categories and Subject Descriptors

B.3.1 [**Hardware**]: Memory Structures-*Semiconductor Memories*

General Terms

Performance

Keywords

Memory, Power, Performance, Resistive Memories, Phase-Change Memory, Write Throughput, Tokens

[*]Gabriel Loh conducted this work while at Microsoft Research as a visiting researcher. He is now with AMD Research.

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil

1. INTRODUCTION

With the continued effective scaling of DRAM technology in doubt [4, 7], resistive memories such as Phase Change Memory (PCM) offer a scalable memory alternative. Rather than represent the value of a bit with an electrical charge, PCM cells store the bit values with the physical state of a chalcogenide material. Under different current patterns, the material can be melted and solidified into either an amorphous or crystalline state, and these different states can be measured by the differences in their resistivity. The last few years have seen a flurry of work attempting to evaluate the potential of Phase Change Memory (PCM) as a alternative to DRAM for main memory. The densities potentially achievable from PCM may make it an attractive target to replace DRAM entirely, and its non-volatility opens many interesting new doors for system optimization [2, 3].

While PCM has read power and delay in the same realm as DRAM [8], writes are very different. Compared to DRAM, PCM writes consume significantly more power, and take significantly more time to complete. The durability of PCM cells, while better than NAND flash cells, is still multiple orders of magnitude lower than in DRAM. While recent work has shown how PCM's unique failure modes can be effectively managed with error correcting schemes [6, 15, 17, 18, 19, 22, 23, 24], and how slow write times can be mitigated in many instances [1, 8, 12, 14, 23], the dynamic power management issues unique to PCM writes have been left largely unaddressed.

The problem is that a write to a PCM cell is an inherently power-intensive operation. As the chalcogenide glass is forced to undergo a state change through heating and then controlled cooling, a bank can draw large amounts of power [20]. Delivering this power is a serious challenge, so much so that PCM systems limit the total number of concurrent writes allowed. This concurrency limitation, when coupled with the long times required to complete a PCM write, creates a tremendous performance bottleneck. However, all is not lost. Unlike DRAM, where activating a row destroys all of the data in its cells, PCM can selectively target individual bits for writing. This ability is already used to minimize the total wear in the system by only overwriting those bits that actually change [24].

Rather than to build a system where the number of memory writes is limited by worst-case update patterns, the main idea behind this paper is to take advantage of the fact that typically only a small portion of the bits (13% on average in our experiments) are written to, and thus consume power. By managing power allocation at a finer granularity, we can intelligently limit consumption to allow a greater number of simultaneous writes, increasing overall system performance. The difficulty in such a scheme is two-fold. First, we need to guarantee that the system will *never* exceed the allocated power budget, as doing so may introduce errors in the

system as circuits fail to receive adequate voltage and as excessive currents cause premature aging via electro-migration. Second, the allocation scheme must happen with a minimum of coordination between the banks of memory (which know the old values of the bits and thus how many must change) and the memory controller (which must allocate power across all chips in the memory system). Round trips on the bus to coordinate power allocation add latency and reduce the effective memory bandwidth – both very bad things from a performance standpoint. We propose a method of allocating memory write power that attempts to balance these concerns with the need for increased write concurrency. This prevents memory controller queues from becoming full, which would otherwise result in read stalls and therefore degraded performance.

In particular, we explore a new class of power-aware memory controllers based on the idea of tokens. Here, tokens are used to represent the capability of drawing a particular amount of power over a predefined epoch of time. We discuss power tokens, the architecture of a PCM system using them, and finally how to calculate tokens so that the memory controller can only issue writes when there is sufficient power to support them. We evaluate our proposal and analyze the effectiveness of power tokens in increasing write bandwidth, and ultimately, performance. However, before we get into the details of our architecture, we first discuss PCM in general and describe why memory controllers will need to be increasingly power-aware.

2. BACKGROUND AND RELATED WORK

If PCM is to become a viable main memory technology, it needs to overcome at least the following three challenges: the inferior performance seen by both reads and writes, limited cell lifetime due to wear, and the higher power required per access (especially with respect to writes).

Access Time: Reads and writes are both slower on PCM than on DRAM, writes especially so [8]. This asymmetry is unlike DRAM, for which read and write times are equivalent. Most techniques proposed to address the write latency issue, and its negative impact on read latency due to bank conflicts, leverage write locality to coalesce as many changes to the data as possible in alternative buffers before they are finally written to PCM [8, 14, 23]. While these optimizations help shield the processor from the impact of slow writes, eventually writes must make their way out to memory.

Cell Lifetime: Limited cell lifetime can be dealt with by using combinations of three strategies: (1) reducing the total number of PCM cell writes [1, 8, 13, 14, 20, 22, 23, 24], (2) spreading cell wear [15, 18, 23, 24], and (3) tolerating cell failures [6, 17, 19, 21]. Most important to this discussion are strategies in the first category because, in addition to reducing wear, they reduce the power required for write operations. We do not propose any new schemes that would make write power allocation easier at the cost of increased wear. All of the above wear management and error correction schemes are fully applicable when our power allocation strategies are used. In fact, our work builds on the idea of differential writes [24], as it can potentially both reduce wear *and* reduce energy.

With differential writes, before a PCM block is written, the old value is read out of the array, compared with the new data to be stored, and only bits that need to change are then written. These read-before-write checks already happen in many proposed PCM devices. This presents a power management opportunity: rather than assume the worst case number of bits to be flipped for each write, as done with DRAM, writes can be coordinated such that the total number of *bits being written* at any time in the system does not exceed a bound. In doing so, we can improve on more conservative power-naive allocation strategies by allowing more concurrent write operations under the same power bound. In fact, our experiments have shown that, on average, only 13% of a cache line's bits are flipped on a write to the PCM memory.

Write Power Management: While there is little prior work directly addressing the power management needs faced by PCM, several of the schemes proposed to improve cell lifetimes have the added benefit of reducing write power. For example, the "Flip-n-Write" strategy [1] tracks when more than half the bits need to be flipped during a single write, at which time the logical sense of the bits is inverted (rather than the bits themselves) with an "invert" bit. For more careful tuning, a block may be partitioned into multiple sub-blocks and each sub-block is associated to a bit indicating whether the bits should be read out as they are or inverted before being returned to the requester. This has the advantage that at most half the bits will ever have to be written on any particular write and is similar to techniques often used to reduce power on high capacitance buses. Because the bound is hard, it is easy to schedule the power accordingly, but at the disadvantage of leaving some performance on the table. We use Flip-n-Write as our baseline and quantify these differences in the results section.

When it comes to this finer grain power management, the challenge is that in current systems the information needed to make good power allocation decisions (namely the number of bits to be flipped) is stranded in the PCM cells and not directly available to the memory controller. To maintain correct functional behavior of the overall PCM circuitry, a memory controller must still guarantee that the power consumption of *a PCM chip or a collection of PCM chips* never exceeds the limits of the delivery capabilities. Seemingly simple approaches, such as simply adding more pins, are stymied by the fact that PCM, as DRAM and other memory technologies before it, service a very low-margin, highly cost-sensitive market. However, violating power limits can lead to voltage drops in the power supply or excessive currents flowing through the system. Voltage drops may lead to logical errors, flops entering meta-stable states, incomplete PCM phase transitions, PCM read errors, etc., and excessive currents may accelerate chip aging due to electro-migration. In the next section we describe how power tokens allow us to engineer a system that will never violate these limits, yet will allow much more aggressive write concurrency.

3. POWER TOKENS

To manage the power delivery across and within the chips of the memory system, we introduce the concept of a PCM *power token* (or simply *token*). Consider a PCM chip that can supply a maximum of P_{limit} Watts of power through its power pins, and that writing a single PCM cell requires P_{bit} Watts. Therefore, the PCM's power delivery system can support $w_{max} = \lfloor P_{limit}/P_{bit} \rfloor$ concurrent bit writes. For each bit write, we associate a single power token that represents the power required to write that bit. The memory controller maintains a pool w_{pool} of power tokens for each PCM chip that starts with w_{max} tokens when no writes are in progress. As writes issue, the number of available tokens decreases, and when writes complete the associated tokens are returned to the pool. Logically, before each write operation, the memory system reads the existing PCM array contents, compares those bits to the new values to be written (as discussed, this is done anyway to support differential reads for write-endurance reasons), and determines the total number of bits w_Δ that need to be written. We show how to estimate this number without needing to wait for the read. If $w_\Delta \leq w_{pool}$, then the write may proceed and consume w_Δ tokens

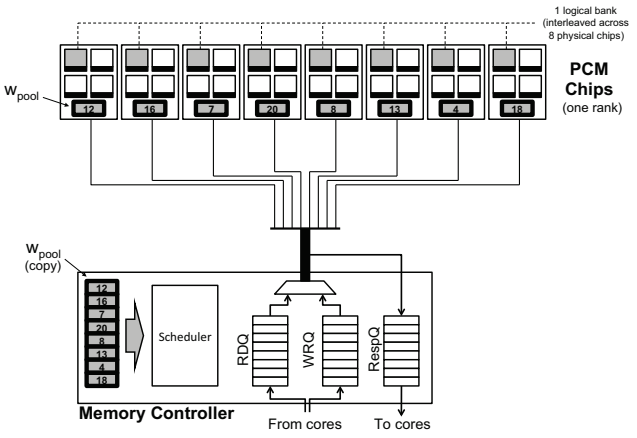


Figure 1: Block diagram of single rank of a general power-aware PCM memory architecture. The logical memory bank is composed of eight physical banks across eight separate PCM chips. As power is constrained on a per-chip basis, a pool of active counters per chip is maintained by the scheduler.

for the duration of the write. Since a bit write may not proceed without first allocating a corresponding token, and since the total number of tokens ever handed out at any instant is limited to w_{max}, then the PCM chip is *guaranteed* to never exceed the chip's power delivery capabilities of P_{limit}.[1]

3.1 Architecture

While the high-level concept of power-tokens is simple and intuitive, the implementation of a practical PCM memory architecture employing such a scheme requires addressing several issues. Figure 1 shows a block diagram of a power-aware PCM architecture. The read/write data-path between the memory controller and the PCM is interleaved across multiple chips, *i.e.*, a logical memory bank is physically composed of eight (in this example) physical banks across the eight separate PCM chips. There is a maximum power draw for each individual PCM chip due to the amount of current its pins can support, and therefore every chip has its own pool of power tokens, tracked by a counter w_{pool}. The memory controller maintains a copy of the counters that tracks the number of power tokens $w_{pool,i}$ available for each chip i, thereby avoiding constant back-and-forth communication with the individual chips to query about the number of available tokens. The memory controller scheduler/arbiter monitors the requests in the read queue (RDQ) and write queue (WRQ), and depending on bus availability, bank availability, circuit timing constraints, and per-chip power token availability, issues commands to the PCM chips. Completed requests wait in the response queue (RespQ) for availability of the interconnect from the memory controller back to the core(s).

The memory controller is responsible for issuing low-level commands (*e.g.*, row activate, column read) directly to the PCM chips. The memory controller scheduler typically takes advantage of bank-level parallelism by overlapping commands to independent banks, but must carefully schedule resources to avoid conflicts. For example, multiple read commands must be timed such that the data-transfer portions do not overlap in time because (in this example) there is a single shared data-bus. With power tokens, the memory controller must also ensure that all chips have enough tokens left in their respective token pools before issuing write commands. If there are not enough tokens for the write to proceed, then the memory controller may choose to work on another memory request that either is not a write, or requires sufficiently few tokens to proceed.

Figure 2 shows an example of the memory controller operation with PCM power tokens. Each chip has four 4-bit wide banks and a power supply (and pins) that can handle writing up to four bits at once, hence each chip starts with four tokens. Note that without the token scheme (and without "Flip-n-Write"), each chip would have to conservatively assume that any write modifies all four bits, and therefore only one write operation could be in progress at any time.
(a) Prior to issuing a write X, the memory controller first checks to see if the number of tokens required (*i.e.*, the number of bit modifications) is less than the number of tokens currently available in the token pool. In this example, 13 bits are different (spread across the different chips), but since each chip has enough tokens to support the write, the memory controller may issue this write X.
(b) The memory controller issues X and decrements the per-chip token-pool counters by the number of bits written per chip (the memory controller would actually modify its local mirrored copy of the counters). The memory controller has other candidate writes Y and Z to issue to other banks. Write Y only modifies five bits, three of these bit modifications are for chip 2, which only has two tokens left due to the ongoing write X in bank B. In this case, Y cannot issue and stalls until a future time when the necessary tokens are available. Write Z modifies nine bits, so it actually needs more total tokens than Y, but it turns out that the chips being modified happen to each have enough tokens to support this write.
(c) The memory controller issues write Z and decrements the corresponding token counts. Note that there are now two simultaneous writes in progress to different banks.
(d) The latency of the write operation in the PCM array is a fixed constant, so after this amount of time has elapsed, write X will have completed and the power tokens can be returned to their respective pools; the memory controller re-increments its copies of the token-pool counters to reflect this.

It is important to note that the "write" memory command is different from the actual modification of the PCM cells. The first causes the data bits to be sent from the memory controller to the PCM chips where they are latched in the row buffers. The latter is the operation that demands high currents and is addressed by the power token approach.

3.2 Computing Token Requirements

A central assumption for a token-based PCM power management scheme is that the memory controller knows how many bits are modified by a write (and thus how many tokens need to be allocated). It may be difficult for the memory controller to know if enough tokens exist, however, because it does not have a copy of the original unmodified data with which to compare the dirty version that needs to be written back, as the original data resides on the other side of the memory bus in the PCM arrays.

Naive Approaches

There are two straightforward approaches for determining the number of modified bits: either transferring the old value of each line to be written back to the memory controller for comparison, or doing the comparison at the PCM row buffers. The first option requires two full data transfers for every write, one first to read the old data back to the memory controller, and then the new write out to the PCM with the updated values. The extra data transfers increase bus contention and overall write latency, while consuming addi-

[1]This is a slightly simplified power budgeting example. In a real system, additional power would be provisioned for leakage current, peripheral circuits, read operations, etc., but the budget for writes would simply be reduced by a corresponding amount.

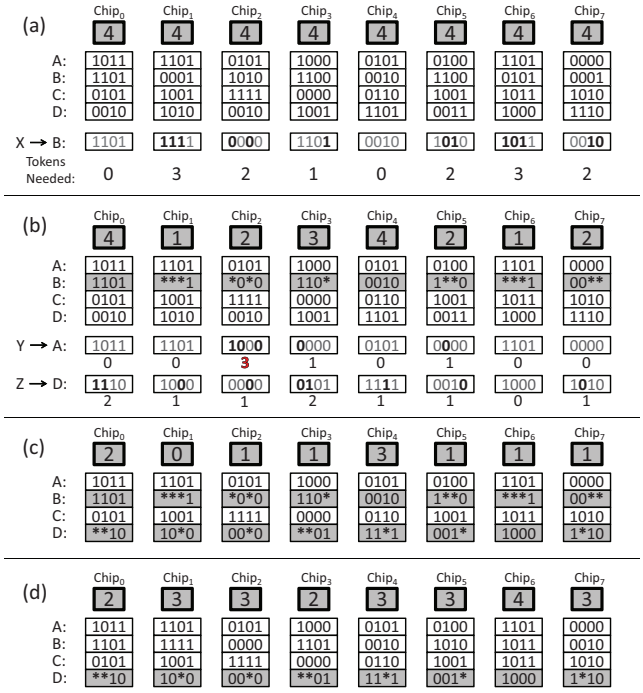


Figure 2: A token-based PCM power management example. There are logical banks (A-D) spread across the eight chips. Each chip can support at most four concurrent bit writes. Banks currently being written to are shaded, and bits actively being modified are indicated with asterisks. (a) A write operation that modifies 13 bits. (b) The write issues and decrements the token counters. There are two more writes waiting to issue, of which only one has enough tokens in all banks. (c) The second write issues and decrements the token counters. (d) The first write completes and returns the tokens it was using.

tional power. The latter approach requires the addition of some minor population-count logic in the PCM, but more importantly it requires modifications to the memory protocol/interface to support the token-count response; this in turn leads to a more complicated memory scheduler which now needs to reserve two data bus slots (one to send the data, one for the token-count response).

Both of these approaches are functionally viable, but neither are very attractive due to the additional required back-and-forth communications between the memory controller and the PCM chips. The fundamental problem is that the original and modified data reside in two physically separate locations and so data must be transferred one way or the other.

Conservative Token Cost Estimation

To bypass the problem of having original and modified data existing in separate locations, we take advantage of two observations. First, another copy of the unmodified data does actually exist, but instead of being physically distant, it is located elsewhere *in time*. Second, instead of allocating a number of tokens exactly matching the number of modified bits, *over-estimating* the number of tokens needed still guarantees that the power consumed by the PCM chips never exceeds P_{limit}.

Modern processors contain multi-level cache hierarchies, usually with write-back policies. In this organization, writes to the cache block accumulate in upper caches (closer to the processor) until the copy is finally evicted and written back to the Last Level Cache (LLC), as shown in Figure 3. Prior to the write back of a line from upper level caches, the LLC still contains a copy of the unmodified value consistent with main memory. Therefore, caches can perform the comparison and population-count during the write back from an upper cache to the LLC, and record the number of modified bits in the LLC along with the cache block, as shown in Figure 3(c). When the LLC evicts the block for write-back to main memory, it also sends along the recorded count of modified bits. The memory controller then uses this count when ensuring that the PCM has enough tokens for the write operation. This requires that the LLC maintain the inclusion property with upper caches, otherwise we are forced to assume the worst case (all bits have changed).

A complication arises when a cache block is written back to the LLC more than once. On the first write-back, the LLC's copy of the block matches that of the PCM main memory. If the block is subsequently re-fetched into upper level caches and further modified, when it is written back a second time, the LLC's copy no longer matches that of the PCM main memory. Rather than try to keep a copy of the original data so that an accurate comparison can always be made (with the large storage overhead it would entail), we instead take advantage of the observation that a conservative estimate of the number of modified bits still prevents any violations of the PCM chips' power limits. When a dirty block is written back to the LLC, we compare the two blocks and count the number of modified bits. We store this count with this cache block in its tag array entry. On the first write-back, this count is accurate. On any subsequent write-backs, we again compare the incoming block to this now dirty LLC block, count how many bits were modified, and add this new count to the existing count. This approach can over-count when a bit is first modified from, say, a zero to a one, and then on a later write-back modified from a one back to a zero, but it is guaranteed to always yield a *conservative* bit-modification count (*i.e.*, equal to or greater than the actual number of bits that are different).

Conservative Per-Chip Token Limits

Unfortunately, knowing the total number of modified bits (or a conservative estimate) is not sufficient. The memory controller must know the number of bits modified *per PCM chip*. For example, back in Figure 2(c), there were enough *total* tokens to support writing Y, but chip 2 did not have enough tokens. If a 64-byte cache block is interleaved across eight chips, then each chip covers 64 bits. To track the number of bit modifications, each of those 8 interleaving sub-blocks now needs its own counter. Considering that each cache block has a baseline cost of about 558 bits (512 for the data, 46 for tag, replacement, coherency state, sharers), and we now need 48 more bits of storage, this translates to an overhead of 8.6%. We will later call this policy simply **Conservative**.

There are a variety of ways to reduce this overhead. Based on the bit-modification analysis mentioned in Section 2, only 13% of the bits are modified in a 64-bit block on average. Instead of using a full $c = \lceil \log_2 n \rceil$-bit counter for n bits, we can use a k-bit counter where $k < c$. This k-bit counter accurately tracks the number of bit modifications up to $2^k - 2$. If the counter reaches its maximum value of $2^k - 1$, then it is conservatively interpreted as a value of n (*i.e.*, all bits modified). Occasionally this may cause the memory controller to allocate more tokens than needed, but in the common case the counter will provide enough precision. After exploring different sized counters, we found that 3 bits (per 64-bit interleaving) gave a good balance between storage overhead (3-bit counters have an overhead of 4.3%) and performance. In the evaluation section we will refer to this as **Conservative 3-bits**.

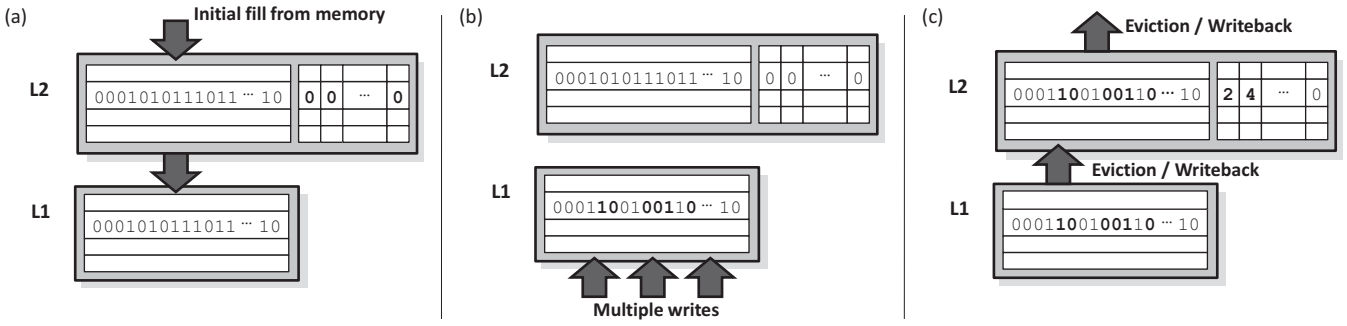


Figure 3: Token cost estimation and flow of a cache line through the memory subsystem. When the line is originally loaded from memory, (a) the counters are reset to zero. (b) The L1 cache modifies the line, but does not write it back immediately, so the values in the L1 and L2 caches differ. (c) The line is eventually written back to the L2 cache and the counters are updated based on differences in the data. When the last level cache (L2 in this case) writes back the line, it sends the memory controller the value of the accumulated counters for the line, as an estimate of the number of modified bits.

Policy	Overhead
Conservative across all chips	1.6%
Conservative 6-bits per chip	8.6%
Conservative 3-bits per chip	4.3%
Conservative with token release	4.3%

Table 1: Storage overheads for token policies.

While the counter policies above should adequately capture most of the potential benefits of dynamic PCM power management, it is theoretically possible for these policies to be *overly* conservative due to multiple write-backs to the LLC (discussed above). To examine the potential of reallocating this power after it is discovered it is not needed, we consider a final policy **Conservative with token release**. After the write data have been transferred from the memory controller to the PCM chips, the PCM chips count the actual number of bits that need writing, and then report this value back to the memory controller. The memory controller then returns any excess power tokens back to the pool. In practice, this policy is harder to implement than the other conservative policies because it requires an extra communication between the PCM chips and the memory controller, but it provides a useful point of comparison to elucidate just how conservative the other policies are. One may imagine even more dynamic schemes that attempt to optimistically release power tokens as writes converge to desired values early, *e.g.*, when using multi-level bit storage schemes. While our conservative schemes may leave some performance on the table here, taking advantage of these opportunities requires more significant two-way communication. Communication of this nature is difficult to support under centrally scheduled traditional memory bus architectures and is thus beyond the scope of this work.

Figure 3 shows each counter associated with a contiguous segment of bits (*e.g.*, the first counter tracks the number of writes to the most significant eight bits, the second counter tracks the next eight bits, etc.). This mapping assumes that the data are interleaved across memory chips in the same fashion (*i.e.*, all eight most significant bits map to the same PCM chip), and the counter provides the bound on the number of tokens required to write this data to that chip. If PCM chips interleave data in a different order, then each counter would track a different set of bits in the cache line corresponding to the interleaving order.

Table 1 summarizes the storage overheads of different policies.

4. EVALUATION

4.1 Evaluation Setup

4.1.1 Simulator

The simulator used in this study is built around performance characteristics and memory system of a standard multicore processor. Because we are considering the effects primarily at the memory system level, past all levels of cache, we have chosen to drive our simulation with long traces gathered with Pin [9], rather than shorter but more detailed pipeline simulations. Our Pin tool simulates each benchmark on a private 64KB 4-way associative L1 cache, and outputs evicted lines and cache misses.

We use an in-house trace-driven simulator to model a shared L2 cache, the memory controller, and the PCM memory chips; the L2 cache is our last level cache. It consumes several traces simultaneously to achieve an effect similar to a multiprogrammed workload. The timing model considers the effect of cache-to-cache and cache-to-memory bus contention, bank conflicts, and memory bus scheduling constraints. Additionally, for policies that keep track of counters, the L2 cache write latency is double the default to account for the extra latency of reading and updating those counters.

The memory controller for the simulator uses standard bus and chip scheduling strategies, but with the added constraint that a write must have sufficient power tokens to proceed. By default, the memory controller gives preference to scheduling reads (*i.e.*, at each cycle schedules a read request) or, if no read requests remain to be scheduled, it schedules a write request. Like some commercial memory controllers, if the write queue becomes completely full, the memory controller issues a *write burst*, issuing only writes and delaying reads until all the writes in the queue have been issued. Table 2 shows the default parameters used for our simulations.

4.1.2 Methodology

To understand how the use of power tokens affects application performance, we simulate a multicore system running multiprogrammed workloads based on SPEC2006 [5]. We use a number of mixes to cover a variety of write and read operation intensities, from medium to very high, and demonstrate the range of behaviors that one might expect under different write and read densities.

Table 3 lists all workloads, the benchmarks that comprise them, and the number of write and read requests to the memory controller

Caches	L1 size	64 KB
	L2 size	8MB
	Line size	64 bytes
	Latency	L1: 1, L2:10 (20)
	Associativity	L1: 4, L2: 16
	Replacement	L1: LRU, L2: LRU
Memory	Banks per chip	8
	Chips per Rank	8
	Ranks	1
	Rank width	64 bytes
CPU	Clock freq	2 GHz
Memory Controller	Queue sizes	24 R/W
PCM Chips	Write	250ns
	Read	60ns
	Full-bit writes	1
Memory Latencies	CPU to L2	25 CPU cycles
	L2 to MC	25 CPU cycles
	MC to Bank	30 CPU cycles

Table 2: Set of parameters used for simulations.

Workload Mix	Benchmark Names	Reads (per K-insts.)	Writes (per K-insts.)
High(4)	mcf, gemsFDTD, astar, sphinx3	6.45	3.11
Mid(4)	mcf, gromacs, gemsFDTD, h264ref	2.68	1.56
Low(4)	gromacs, h264ref, astar, sphinx3	2.31	1.08
Astar(8)	astar(8)	8.05	5.65
Gems(8)	gemsFDTD(8)	4.15	2.6
ACGS(8)	cactus(2), soplex(2), gemsFDTD(2), astar(2)	5.09	2.09
AGSZ(8)	zeusmp(2), soplex(2), gemsFDTD(2), astar(2)	3.99	1.74
ACLS(8)	cactus(2), leslie(2), soplex(2), astar(2)	5.03	1.64
ACSZ(8)	zeusmp(2), cactus(2), soplex(2), astar(2)	3.42	1.25
Leslie(8)	leslie(8)	3.62	0.96
CGSZ(8)	zeusmp(2), cactus(2), soplex(2), gemsFDTD(2)	2.83	0.75
CLSZ(8)	zeusmp(2), cactus(2), leslie(2), 450(2)	2.87	0.46
Cactus(8)	cactus(8)	1.62	0.46
Zeusmp(8)	zeusmp(8)	0.79	0.39

Table 3: Workload *mixes*. The number of copies of each benchmark are in parentheses.

per thousand instructions executed – the higher these numbers, the greater the intensity of requests hitting the memory controller.

All benchmarks were compiled with full optimization, and all workloads executed for 1.2 billion instructions with the *ref* inputs after being fast-forwarded to the most significant SimPoint [11], resulting in trace files of L1 cache fills and evictions. Using the mixes of benchmarks described above, each workload was run on the simulated multicore system for a combined one billion instructions, after a 200 million cache warm-up phase.

The actual power required for a write may vary due to parametric variations between PCM cells, and also due to whether a cell is being set or reset. We conservatively assume the worst-case power for writing a single bit, and so our results underestimate the full potential of power tokens. We did consider using different token costs for writing zeros and ones, but this required that the LLC maintain two sets of counters for the number of zeros written and the number of ones written. This optimization did not provide sufficient performance increase compared to our other schemes to justify the overhead of doubling the number of counters in the LLC.

4.1.3 Policies

Each workload is run with a number of token *policies*. Each policy represents a different memory controller power allocation strategy. Table 4 lists the names of policies we evaluate, along with how many tokens are used and more details on how the policy allocates write power. The next question is how many PCM bits can be written to concurrently (*i.e.*, the baseline power available for write operations).

Modern DDR3 DRAMs use about 100 mA of current for an activate-precharge command sequence; for a representative DDR3-1066 x16 memory [10], the command sequence requires 95 mA after subtracting out background/leakage currents. Out of this 95 mA current, about one fourth (21 mA) goes to the precharge command that performs the actual write operation from the row buffer back to the DRAM array. DDR3 supports eight banks that can each process commands independently; in an ideal case, all eight banks could write to their arrays at the same time, thereby drawing 168 mA of current. For a realistic DRAM where eight precharges cannot be simultaneously launched, the current draw would be lower. For PCM, a write operation requires significantly higher currents, (*e.g.*, 300 μA per bit for a reset operation [8]). Assuming the same write current limitation of 168 mA for the DRAM, a PCM would only be able to write 560 bits (168 mA/0.3 mA-per-bit), or just *slightly more than a single 64-byte cache line*. Enabling the writing of an entire row buffer (1KB-2KB) or multiple concurrent writes would require a much more power/current than that supplied to current DRAM chips, which would in turn make the PCM much more difficult to use as a "drop-in" DRAM replacement.

Limiting writes to a single operation at a time affects performance negatively. To optimize the baseline against which we compare, we implement "Flip-n-Write" [1], which flips at most half of the bits being written. While there is enough power for only one full write, "Flip-n-Write" enables a power-naive memory controller to perform two writes concurrently, a significant improvement to our baseline. Our power-aware policies use "Flip-n-Write" only when a write would flip more than half the bits being written.

The first policy, *limited-2*, is our baseline PCM configuration. Using "Flip-n-Write", it can write up to two banks concurrently. The other extreme is *unlimited*, a policy that ignores power limitations on writes — the best case scenario in terms of write concurrency. The *oracle* policy uses power tokens, but has perfect knowledge of the number of tokens required to write. Finally, we have three different conservative counting policies (see Section 3.2 for token counting details). First, the *conservative* policy counts tokens in the L2 cache without limiting the number of bits for the counters. The *conservative 3-bits* policy uses just three bits per counter (24 bits for the 8 counters). Finally, the *conservative token-release* policy has no bit limitation on counters, and returns to the memory controller unnecessary tokens assigned to the PCM chips; this policy releases unused tokens earlier to the memory controller in the hope of enabling a new write operation to start earlier, but it also consumes an extra cycle in the memory bus due to the token return. We simulate these six polices with each workload. This allows us to explore the maximum speedups possible with power management for PCM, and compare them to the behavior of other realistic power management schemes. We also include how many traces the experiments used, either 4 or 8, in the workload label.

4.2 Overall Effectiveness

Figure 4 shows the overall speedup normalized to *limited-2* for all the workloads. As expected, the most write-intensive mixes show the greatest benefits with power management. The *unlimited* policy shows a wide range of speedups versus the *limited-2* policy, with the average speedup being 20% (geometric mean) and a maximum speedup in Astar(8) of 85%. This is expected as *limited-2* is limited to 2 concurrent writes, whereas *unlimited* has no such power limitations.

The *oracle* policy behaves very similarly to *unlimited*, differing by 2% on average. This is due to the average number of bits changed being about 13% of a cache line on average (see Figure 8 for more details) and writes rarely causing the memory controller to

Policy	Tokens Requested	Policy Description
limited-2	maximum	power ignorant memory controller
unlimited	0	ignores power limitations
oracle	exact # of modified bits	best possible token counting in L2
conservative	bit flips in L2	limit for counting bit flips
conservative 3-bits	*conservative* with bounded counters	realistic counting scheme
conservative token-release	bit flips in L2	releases unnecessary tokens early

Table 4: Experiment policies.

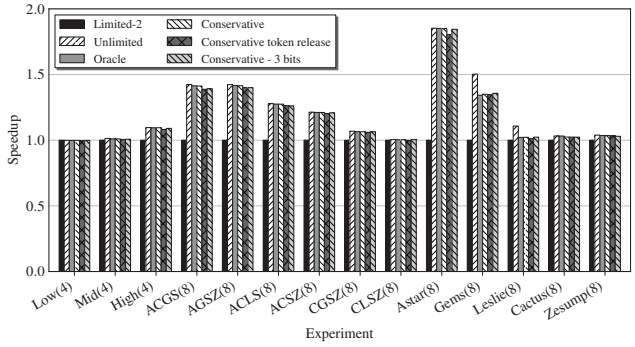
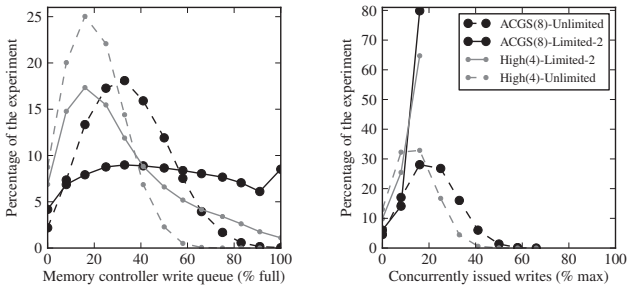


Figure 4: Speedup over *limited-2* for all experiments.

Experiment	% of time in write burst for *limited-2*	Speedup of *unlimited*
Astar(8)	77.1%	85.2%
Gems(8)	61.1%	50.3%
ACGS(8)	47.4%	42.4%
AGSZ(8)	46.3%	42.4%
ACLS(8)	34.9%	27.9%
ACSZ(8)	29.3%	21.4%
Leslie(8)	11.5%	10.8%
High(4)	10.8%	9.6%
CGSZ(8)	10.1%	6.8%
Cactus(8)	5.8%	3.2%
Zeusmp(8)	5.1%	3.9%
Mid(4)	3.3%	1.3%
CLSZ(8)	0.5%	0.5%
Low(4)	0.1%	-0.1%

Table 5: Percentage of time spent in write bursts for *limited-2* and average speedup of *unlimited* versus *limited-2*, sorted from highest write burst percentage to lowest.



(a) Write queue (% full) (b) Concurrent writes (% max)

Figure 5: Write queue occupancy and write concurrency distributions.

run out of tokens. We omit *oracle* from remaining plots that show *unlimited* due to their similarity.

Much like the *oracle* policy, the *conservative* policy behaves similarly to *unlimited*, differing again by 2%. This shows that the overestimation approach works quite well, most of the time performing as well as *oracle* (*i.e.*, as if it could perfectly predict how many tokens are needed for each write).

The *conservative token-release* policy, an extension of *conservative* that returns unused tokens to the memory controller, performs slightly worse than *unlimited*, on average 3% slower. The reason is that *conservative* only slightly overestimates its token count, so there is not much benefit in returning tokens earlier. Unfortunately, *conservative token-release* occupies an extra memory data bus cycle per write, which increases conflicts on the bus, lowering write and read bandwidths and degrading performance. For this reason, we do not further evaluate *conservative token-release*.

The *conservative 3-bits* policy, a more realistic implementation, is slightly worse than *unlimited*, differing by 2%. Compared to the *limited-2* policy, *conservative 3-bits* has an average speedup of 17% and a maximum speedup of 84% for Astar(8). Finally, the small performance difference between *unlimited* and *conservative 3-bits* indicates that differentiating the number of tokens the set and reset operations use would not improve performance much further.

4.3 Characterization

Write bursts: These performance results can be characterized by the percentage of time the workload spends in write bursts, *i.e.*, postponing read requests while draining the write queue when it is full. Table 5 shows the average time a workload spends in write bursts and the speedup of *unlimited* over *limited-2*. The greater the time *limited-2* spends in write bursts, the greater the speedup because reads are delayed more often. This holds true both over the entire experiment and during parts of the experiment where write bursts occur more frequently than average (not shown).

Write bursts occur when the write queue in the memory controller fills up completely. This can only occur at sufficient write densities, *i.e.*, when the rate of write completion at PCM chips is lower than the rate of write request arrivals to the memory controller. The policies that have no power limitations (*oracle*) or that use power management (all others, except *limited-2*) can support much greater densities of writes, resulting in fewer write bursts and better performance.

When workloads exhibit high write densities, power-aware policies can smoothly schedule extra writes, while *limited-2* has a limit of just 2 concurrent writes. Figure 5 provides more insight by comparing two workloads (ACGS(8) and High(4)) and two policies (*unlimited* and *limited-2*). Figure 5(a) shows the distribution of how full the memory controller write queue is over time. Figure 5(b) shows the distribution of how many writes are simultaneously in flight, as a fraction of the maximum (we assume 8 banks, so the maximum would be 8 concurrent writes, provided that there is sufficient write power). These two figures show that *unlimited* spends most of the time with its queues at low occupancy, which is a direct result from being able to issue a higher number of concur-

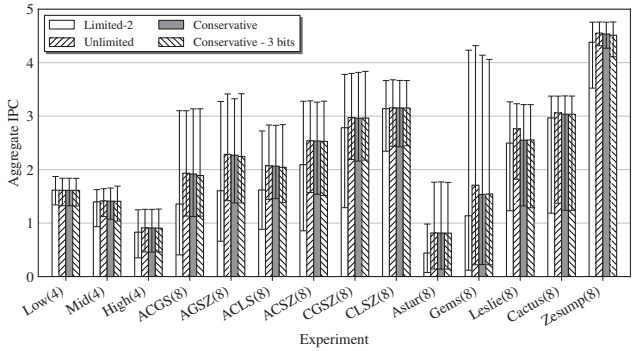


Figure 6: Aggregate IPC for all experiments.

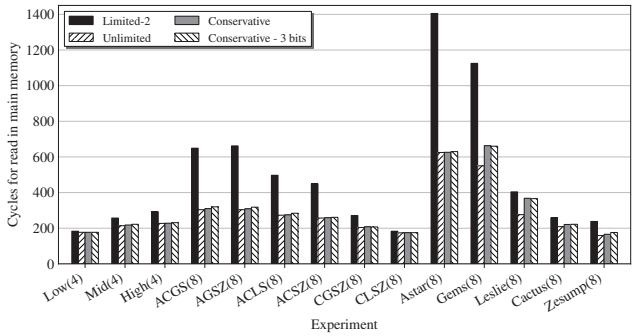


Figure 7: Read latency for read requests to main memory.

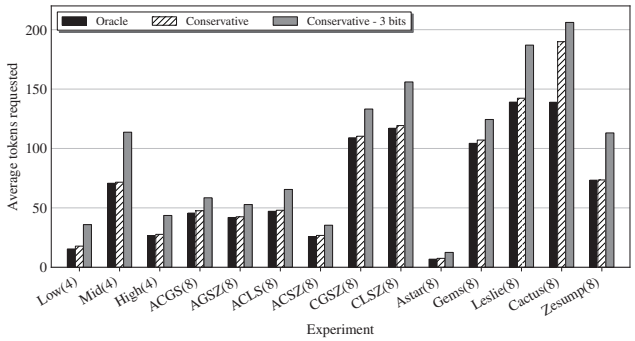


Figure 8: Average tokens requested for each experiment.

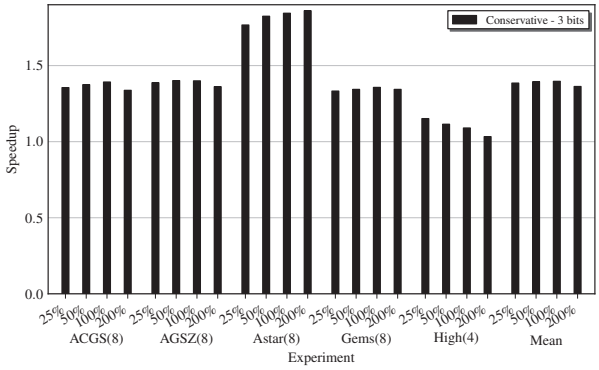


Figure 9: Sensitivity to L2 cache size.

rent writes. The *limited-2* policy is limited to 25% of the maximum write concurrency, so its queues are full for longer.

Aggregate IPC: Figure 6 shows the aggregate IPC for all workloads, *i.e.*, the number of instructions run (1 billion past the 200 million cache warm-up) divided by the total number of cycles to execute them. We include error bars in Figure 6 calculated by taking the maximum and minimum IPC for a set of 80 measurements collected regularly as experiments run (past the cache warm-up phase).

The *limited-2* policy experiences much lower ranges of IPC and rarely has a maximum IPC higher than other policies, showing that *limited-2* experiences periods of higher slowdown. This again is explained by the higher latency of read requests experienced due to write bursts. Even worse than the average slowdown are stalls that users may experience during a period of high write activity.

Read Latency: Figure 7 shows the average read latency for each workload. This latency is the time between a read request enters the memory controller to the time the requested line reaches the memory controller's output queue. Policies with larger speedups over the *limited-2* policy show larger differences between their read latencies, as only reads can result in direct delays to an experiment.

Taking into account the various delays and the read latency in the PCM chip, a read request, if issued immediately, takes 105 (processor) cycles, or 52.5ns. On average, *unlimited* reads require 282 cycles to complete, nearly half the time as a *limited-2* read, which takes 491 cycles on average. Reads in the realistic *conservative 3-bits* at 304 cycles are only 22 cycles slower than *unlimited*.

Tokens: Finally, Figure 8 characterizes the number of tokens requested by each of the power token policies, broken down by workload and policy. The *conservative token-release* policy requests the same number of tokens as the *conservative* policy, and is not displayed. These experiments do not consider write cancellation or write pausing [16].

In general, experiments with a high speedup request small numbers of tokens. On average, *oracle* requests 69 tokens — equivalent to the number of bits flipped on average. This means that on average 13% of the bits in a cache line are flipped on a write, with the highest and lowest percent of bit flips being 27% and 1%, respectively. The *conservative* policy does only slightly worse at 74 tokens, and *conservative 3-bits* requests 96 tokens on average.

4.4 Sensitivity Analysis

Some questions remain about how the effectiveness of power tokens depend on specific parameters. We show a representative subset of workloads, ACGS(8), AGSZ(8), Astar(8), Gems(8), and High(4), with the *conservative 3-bits* and *limited-2* policies, and vary one parameter at a time. We display the speedup of *conservative 3-bits* over *limited-2*. We omit results for *unlimited* and *conservative* because their behavior is similar to *conservative 3-bits*.

L2 cache size: We simulated L2 caches that are as large as 25%, 50%, and 200% as the default cache size (8MB). Figure 9 shows that the effect of larger caches depends on the number of write bursts experienced by *limited-2* compared to *conservative 3-bits*. For low numbers (*e.g.*, High(4)), *limited-2* significantly benefits from higher hit rates because they eliminate many memory requests that cause those few original write bursts. For high numbers (*e.g.*, Astar(8)), the larger caches reduce the L2 write-back

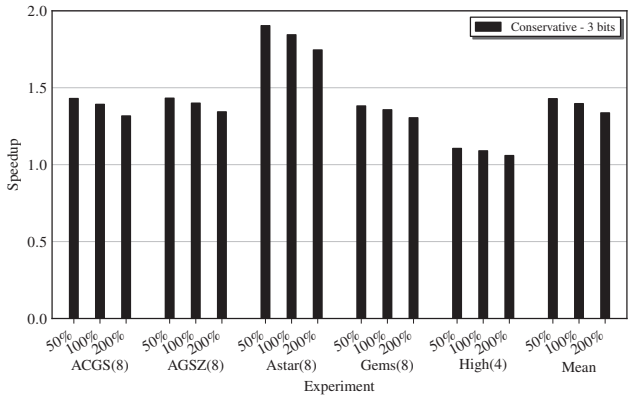


Figure 10: Sensitivity to read latency of PCM chips.

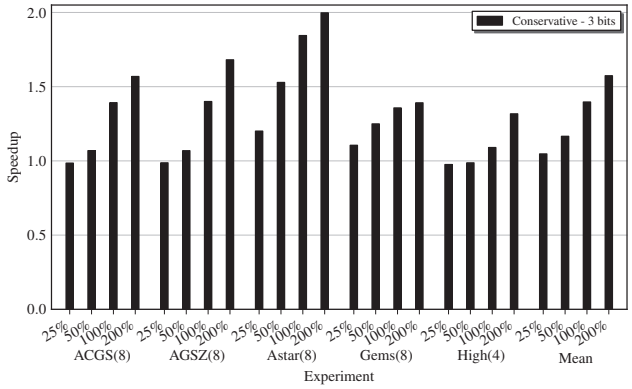


Figure 11: Sensitivity to write latency of PCM chips.

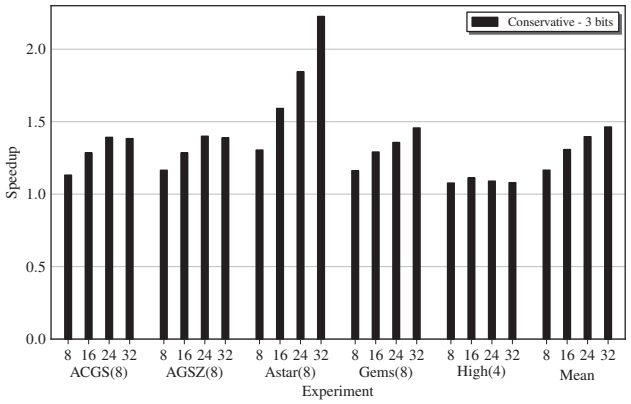


Figure 12: Sensitivity to memory controller input queue size.

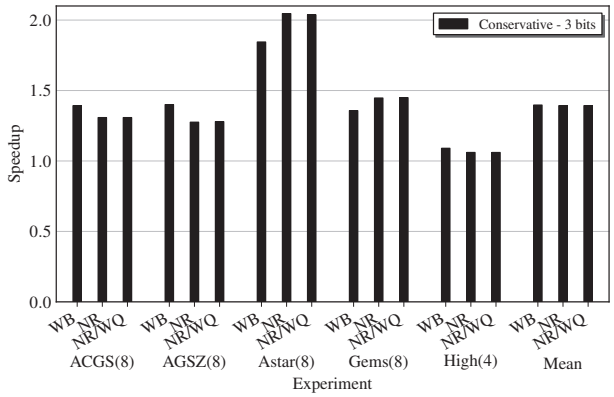


Figure 13: Sensitivity to memory controller issue policy.

rate, but are insufficient to eliminate most write bursts experienced by *limited-2*, while they are sufficient for *conservative 3-bits*.

PCM read and write latencies: For the next sensitivity experiments, we varied the latencies of reads and writes in PCM. We simulated read latencies 50% and 200% as long as the default latency (30 cycles, or 60ns) and write latencies as long as 25%, 50%, and 200% of the default latency (500 cycles, or 250ns).

Figure 10 shows that as read latency grows the difference between *conservative 3-bits* and *limited-2* slowly decreases. As reads get faster, they are serviced more rapidly by the memory controller, reducing contention and avoiding write bursts in some cases.

Faster writes have a much more significant effect on performance, as Figure 11 shows. At low write latencies, the performance of *conservative 3-bits* and *limited-2* is roughly the same. As write latencies increase, so does the performance difference between *conservative 3-bits* and *limited-2*. In fact, *conservative 3-bits* sometimes increases its performance advantage over *limited-2* superlinearly. This is due to the higher number of write bursts caused by slower writes. For example, at 200%, ACGS experiences write bursts 70.7% of the time, compared to 46.6% of the time with the default write latency and 10% of the time at 50% of the write latency.

Memory controller queue size: We next show the performance differences in varying the memory controller queue size. We simulated queues with 8, 16 and 32 entries, in addition to the default 24 entries. Figure 12 shows that, as the number of entries increases, so does the speedup of *conservative 3-bits* compared to *limited-2*.

With only 8 entries, all policies experience significant amounts of stalls due to write bursts, although *conservative 3-bits* already shows better performance. As the queue size increases to 16 and 24, the performance of *conservative 3-bits* grows more quickly than *limited-2* because *conservative 3-bits* experiences fewer write bursts due to its higher write concurrency capacity. The difference in performance between these two policies further increases when the number of entries grows from 24 to 32, although more slowly.

Memory controller queue handling: The final sensitivity experiments explore different issue policies for reads and writes in the memory controller. The default policy (*WB*) is to issue one read per cycle if there are reads in the queue, unless the write queue is full, in which case the write queue is completely drained before any read can proceed (write bursts). The first policy against which we compare (*NR*) also issues one read per cycle if there are any in the queue, but does not drain the write queue when it gets full (no write bursts). The second policy (*NR/WQ*) is similar to *NR*, but issues the write at the head position if the write queue is full, even if the read queue is not empty.

At first glance, Figure 13 shows no clear trend on the effects of disabling write bursts. However, a more careful look reveals that *conservative 3-bits* benefits the most from disabling write bursts for workloads with periods of high write intensity (*e.g.*, Astar(8)),

while *limited-2* benefits the most when these periods are not present (*e.g.*, AGSZ(8)). Under high intensity, write bursts completely drain the write queue, so the queue does not fill up again for a long time. Without write bursts, the probability of the write queue to fill up and cause LLC stalls is higher, more so for *limited-2* than for *conservative 3-bits*. Under low intensity, *limited-2* does not benefit from write bursts as much because after the write queue is drained there are not many more writes to service. Speedups of *conservative 3-bits* over *limited-2* are high with any of the memory controller issue policies, showing the value of power tokens.

5. CONCLUSION

The use of Phase Change Memory as a replacement for DRAM in main memory opens up many new challenges with respect to wear, performance, and power. Writes are the shared culprit among these problems as the very nature of a memory cell that is heated and cooled on every bit flip opens up a host of problems. While the problems of wear and performance have been studied extensively, we believe this paper is the first to treat the problem of variable power allocation that is inevitable under such conditions. The heart of this problem is the lack of a single point in the system that knows everything that is needed to make optimal power scheduling decisions. The memory controller has a global view of the entire system and, as the shepherd of all memory accesses, is in the perfect position to optimize the allocation of resources across the entire system. This is especially true for power constraints that affect many banks across a rank. Unfortunately, the memory controller has a very poor view of how much power is actually *needed* at each bank in the system. That information is kept within an individual die, as it is not known until the memory is read, and then compared to the incoming bits.

The central idea behind our approach is to build a good yet conservative estimator by which the memory controller can make safe yet optimized decisions regarding write concurrency. There is a spectrum of design points in this space, with decisions being made anywhere between the memory controller and the end banks themselves. As one moves decisions away from the memory controller, however, the problem of coordination and communication becomes even harder. If the memory system of the future continues to look like the traditional DIMM topology, the cost of that communication can be quite high. Standard memory protocols such as SDRAM and RDRAM assume that the controller has a *perfect* understanding of the state of the entire memory. If this assumption changes, it would mean a radical change to those architectures. In fact, such a change may already be overdue. That being said, even with the fairly conservative power estimation scheme we have proposed, large improvements are possible. Our experiments show that a power-aware memory controller unleashes speedups of up to 84%.

6. REFERENCES

[1] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *International Symposium on Microarchitecture*, December 2009.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Symposium on Operating Systems Principles*, 2009.

[4] I. T. W. Group. ITRS 2009 edition. Technical report, International Technology Roadmap for Semiconductors, 2009.

[5] J. L. Henning et al. SPEC CPU2006 benchmark descriptions. *Computer Architecture News*, 34(4), September 2006.

[6] E. Ipek, J. Condit, E. Nightingale, D. Burger, and T. Moscibroda. Dynamically replicated memory: Building resilient systems from unreliable nanoscale memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, March 2010.

[7] K. Kim et al. Technology for sub-50 nm DRAM and NAND flash manufacturing. *IEDM Tech. Dig*, 144, 2005.

[8] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase-change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture*, June 2009.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddy, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[10] Micron Technology, Inc. *Micron Technical Note TN-41-01: Calculating Memory System Power for DDR3*, 2007.

[11] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31. ACM, 2003.

[12] M. Qureshi, M. Franceschini, and L. Lastras. Improving read performance of phase change memories via write cancellation and write pausing. In *International Symposium on High-Performance Computer Architecture*, January 2010.

[13] M. Qureshi, M. Franceschini, L. Lastras, and J. Karidis. Morphable memory system: A robust architecture for exploiting multi-level phase change memories. In *International Symposium on Computer Architecture*, June 2010.

[14] M. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture*, June 2009.

[15] M. K. Qureshi, M. Fraceschini, V. Srinivasan, L. Lastras, B. Abali, and J. Karidis. Enhancing lifetime and security of phase change memories via start-gap wear leveling. In *International Symposium on Microarchitecture*, December 2009.

[16] M. K. Qureshi, M. Franceschini, and L. A. L.-M. no. Improving read performance of phase change memories via write cancellation and write pausing. In *International Symposium on High Performance Computer Architecture*, 2010.

[17] S. Schechter, G. Loh, K. Strauss, and D. Burger. Use ECP, not ECC, for hard failures in memories. In *International Symposium on Computer Architecture*, June 2010.

[18] N. H. Seong, D. H. Woo, and H.-H. Lee. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *International Symposium on Computer Architecture*, June 2010.

[19] N. H. Seong, D. H. Woo, V. Srinivasan, J. Rivers, and H.-H. Lee. SAFER: Stuck-at-fault error recovery for memories. In *International Symposium on Microarchitecture*, December 2010.

[20] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A low power phase-change random access memory using a data-comparison write scheme. In *International Symposium on Circuits and Systems*, June 2007.

[21] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *International Symposium on High Performance Computer Architecture*, 2011.

[22] W. Zhang and T. Li. Characterizing and mitigating the impact of process variations on phase change memory systems. In *International Symposium on Microarchitecture*, December 2009.

[23] W. Zhang and T. Li. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2009.

[24] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *International Symposium on Computer Architecture*, June 2009.