DynRBLA: A High-Performance and Energy-Efficient Row Buffer Locality-Aware Caching Policy for Hybrid Memories

HanBin Yoon
hanbinyoon@cmu.edu

Justin Meza
meza@cmu.edu

Rachata Ausavarungnirun
rausavar@andrew.cmu.edu

Rachael Harding
rharding@andrew.cmu.edu

Onur Mutlu
onur@cmu.edu

Computer Architecture Lab (CALCM)
Carnegie Mellon University

SAFARI Technical Report No. 2011-005

December 16, 2011

Abstract

Phase change memory (PCM) is a promising memory technology that can offer higher memory capacity than DRAM. Unfortunately, PCM's access latencies and energies are higher than DRAM and its endurance is lower. DRAM-PCM hybrid memory systems use DRAM as a cache to PCM, to achieve the low access latencies and energies, and high endurance of DRAM, while taking advantage of the large PCM capacity. A key question is what data to cache in DRAM to best exploit the advantages of each technology while avoiding their disadvantages as much as possible.

We propose DynRBLA, a fundamentally new caching policy that improves hybrid memory performance and energy efficiency. Our observation is that both DRAM and PCM contain row buffers which cache the most recently accessed row. Row buffer hits incur the same latency in DRAM and PCM, whereas row buffer misses incur longer latencies in PCM. To exploit this, we devise a policy which tracks the access and row buffer misses of a subset of recently used rows in PCM, and caches in DRAM only rows which are likely to miss in the row buffer and be reused.

Compared to a cache management technique that only takes into account the frequency of accesses to data, our row buffer locality-aware scheme improves performance by 15% and energy efficiency by 10%. DynRBLA improves system performance by 17% over an all-PCM memory, and comes to within 21% of the performance of an unlimited-size all-DRAM memory system.

1 Introduction

Multiprogrammed workloads on chip multiprocessors require large amounts of main memory to support the working sets of many concurrently executing threads. Today, this demand is increasing as the number of cores on a chip continues to increase, and applications become more data-intensive. The main memory in modern computers is composed of DRAM. Though strides in DRAM process technology have enabled DRAM to scale to smaller feature sizes and thus higher densities (i.e., capacity per unit area), it is predicted that DRAM density scaling will become costly as feature size continues to reduce [28, 24]. Satisfying high memory demands exclusively with DRAM will quickly become expensive in terms of both cost and energy.

Phase Change Memory (PCM) offers a competitive alternative to DRAM. PCM is an emerging random-access memory technology that is expected to scale to smaller feature sizes than DRAM [24, 14, 13]. This is due to the fact that PCM stores information as varying electrical resistance, which is more amenable to extreme scaling than how DRAM stores information as a small amount of electrical charge. Furthermore, the non-volatility presented by PCM not only eliminates the need for periodic refresh of the memory cells as is required for DRAM, but also opens the door

to system-level opportunities that may exploit this characteristic (such as lowering OS overheads and increasing I/O performance through the use of persistent memory [4, 5, 1]).

However, PCM has a number of shortcomings which prohibit its adoption as a DRAM replacement. First, PCM exhibits higher read and write latencies compared to DRAM. Second, the dynamic energy consumed in reading from or writing to an individual memory cell is higher for PCM than it is for DRAM. Finally, PCM cells wear out as they undergo write operations. These disadvantages must be tackled in order for PCM to become a viable alternative for main memory.

Hybrid memory systems comprising both DRAM and PCM technologies attempt to achieve the low latency/energy and high endurance of DRAM while taking advantage of the large capacity offered by PCM. Previous proposals [21, 6, 30] employ a small amount of DRAM as a cache to PCM [21], or use it to store data that is frequently written to [6, 30]. A key question in the design of such a DRAM-PCM hybrid memory system is what data should be cached in DRAM to best exploit the advantages of each technology while avoiding its disadvantages as much as possible.

In this work, we develop new mechanisms for deciding what data should be cached in DRAM, in a DRAM-PCM hybrid memory system. Our main observation is that a memory bank, whether it is a PCM bank or a DRAM bank, employs row buffer circuitry [13]. This row buffer acts as a cache for the most recently activated row in the bank. A memory request that hits in the row buffer incurs the same latency in both PCM and DRAM, whereas a request that misses in the row buffer requires activating the requested row in the memory cell array, which incurs a much higher latency in PCM than in DRAM. Placing a row of data that mostly leads to row buffer hits in DRAM provides little benefit, whereas placing a heavily reused row of data that leads to frequent row buffer misses in DRAM saves the latency and energy of costly PCM accesses for that row.

Based on this new observation, we devise a new caching policy, DynRBLA: cache in DRAM only those rows of data that are likely to miss in the row buffer and also likely to be reused. To implement this policy, the memory controller keeps track of the row buffer access and miss counts of a subset of recently used rows in PCM, and places into the DRAM cache a row whose access and miss counts exceed certain thresholds, adjusted dynamically at runtime. We observe our mechanism (1) mitigates the high access latencies and energy costs of PCM, (2) reduces memory channel bandwidth consumption due to the movement of data between DRAM and PCM, and (3) balances the memory access load between DRAM and PCM.

Our evaluations on a wide variety of workloads on a 16-core system show that our proposal improves system performance by 41% over using DRAM as a conventional cache to PCM, while reducing maximum slowdown by 32% and increasing energy efficiency by 23%. Furthermore, our scheme shows 17% performance gain over an unlimited-size all-PCM memory system, and comes within 21% of the performance of an unlimited-size all-DRAM memory system. Compared to a row buffer locality-oblivious caching policy that selectively places data in a DRAM cache based on frequency of use [9, 23], our proposal significantly improves both performance and energy-efficiency.

Our contributions are (1) the identification of row buffer locality as a key metric in making caching decisions in a hybrid memory system, (2) a row buffer locality-aware hybrid memory caching policy based on this observation, (3) the evaluation of our mechanism against caching policies that are unaware of row buffer locality on a hybrid memory system, showing that our proposal provides significant improvements in performance and energy efficiency.

2 Background

2.1 Memory Device Architecture: Row Buffers

Memory devices have cell arrays that are typically organized into rows (cells sharing a common word line) and columns (cells sharing a common bit line), where accesses to the array occur in the granularity of rows (illustrated in Figure 1). During a read, a specified row is read by sense amplifiers and latched in peripheral circuitry known as the *row buffer*.

Once the contents of a row are latched in the row buffer, subsequent memory requests to that row are serviced quickly from the row buffer, without having to interact with the array. Such a memory access is called a *row buffer hit*. However, if a different row that is not in the row buffer is requested, then the requested row must be read from the array into the row buffer (replacing the row buffer's previous contents). Such a memory access incurs the latency and energy of activating the array, and is called a *row buffer miss*. *Row Buffer Locality (RBL)* refers to the reuse of a row while its contents are in the row buffer. Memory requests to data with high row buffer locality are serviced efficiently without having to frequently activate the memory cell array.

2.2 Phase Change Memory

Phase Change Memory (PCM) is a non-volatile memory technology that stores information by varying the electrical resistance of a material known as chalcogenide [29, 24]. A PCM memory cell is programmed by applying heat to the chalcogenide and then cooling it at different rates, depending on the data to be programmed. Quick quenching puts the chalcogenide into an amorphous state which has high resistance, representing a '0', and slow cooling puts the chalcogenide into a crystalline state which has low resistance, representing a '1'.

A key advantage of PCM is that it is expected to offer higher density than DRAM. While a DRAM cell stores information as charge in a capacitor, a PCM cell stores information in the form of resistance, which is expected to scale to smaller feature sizes. Also, unlike DRAM, PCM offers multi-level cell capability, which stores more than one bit of information per memory cell by achieving more than two distinguishable resistance levels. This further enhances PCM's opportunity for high capacity.

However, PCM has a number of drawbacks compared to DRAM. The long cooling period required to crystallize chalcogenide leads to higher PCM write latency, and read (sensing) latencies for PCM are also longer. A technology survey on nine recent PCM devices and prototypes [13] reported PCM read and write latencies at 4 to 6× and 6 to 32× that of DRAM, respectively. In addition, PCM read and write energies were found to be 2× and 10 to 140× that of DRAM, respectively. Furthermore, repeated thermal expansions and contractions of a PCM cell lead to finite endurance, which is estimated at 10^8 writes.

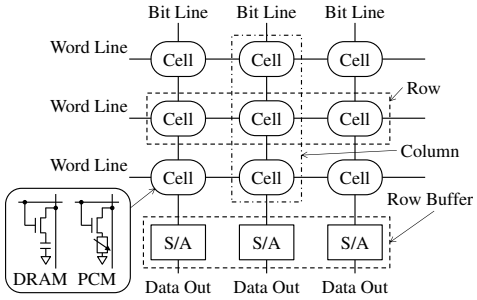


Figure 1: Array organized into rows and columns.

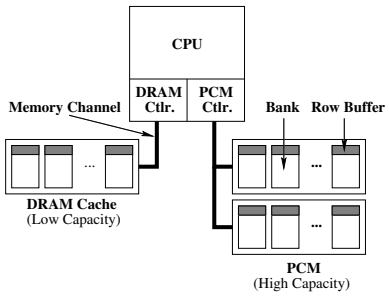


Figure 2: Hybrid memory organization.

2.3 DRAM-PCM Hybrid Memory

DRAM-PCM hybrid memory systems [21, 6, 30] aim to mitigate the drawbacks of PCM while taking advantage of its high capacity benefits by using DRAM as a cache. The increased memory capacity per cost leads to reduced page faults in a system, and the DRAM cache acts to mitigate the high latency and energy costs of PCM. This increases overall system performance and energy efficiency [21].

Figure 2 illustrates the organization of such a hybrid memory system. The OS page table tracks which rows are cached in DRAM and which are stored in PCM, to operate a large DRAM cache capacity (on the order of a gigabyte). Implementing the DRAM as an inclusive cache to PCM has the advantage that the dirty cache blocks can be tracked and selectively written back to PCM on the eviction of a row (known as line level writeback [21]). This reduces writes to PCM, thus enhancing its lifespan and reducing write energy (which is significantly higher than read energy on a per-cell basis).

2.4 Data Placement in a Hybrid Memory

A key question in designing a high-performance hybrid memory system is deciding what data to place in the DRAM cache. Unlike an on-chip cache, an off-chip DRAM cache exhibits significantly higher latencies on the order of hundreds of CPU clock cycles. Furthermore, data movements occupy higher memory bandwidth for migrating rows of data on the order of kilobytes at a time. Influenced by these additional factors, a hybrid memory system requires an effective DRAM cache management policy to achieve high performance.

3 Motivation

Because of the disparate array access characteristics of DRAM and PCM, the effectiveness of caching techniques in a hybrid memory system largely depends on how the pattern of memory requests is mapped between the DRAM and PCM devices. Take, for example, a simple program which loads data from sequential cache blocks in main memory. Because of the row buffer present in both DRAM and PCM devices, most of this program's accesses will be serviced at a low row buffer hit latency, except for the occasional row buffer miss. Such an access pattern exhibits *high* row buffer locality and the data it accesses can be placed on either DRAM or PCM without significantly affecting average memory access latency. Next, consider another program which loads data at a stride much larger than the row buffer size. In this case, it is likely that very few of the program's accesses will be serviced at a low row buffer hit latency. Such an access pattern exhibits *low* row buffer locality and the average memory access latency will be highly sensitive to which device the data it accesses is located on. Preserving row buffer locality will become an even more acute problem in future many-core architectures, as requests from many different access streams will interfere with one another and get serviced in main memory in a way which may not be amenable to row buffer locality.

Unfortunately, current hybrid memory and on-chip cache management proposals do not consider row buffer locality in making their caching decisions and instead seek to solely improve the reuse of data placed in the cache and reduce off-chip memory access bandwidth. As we will show in this paper, the row buffer locality of data placed between the DRAM and PCM devices in a hybrid main memory can play a crucial role in determining average memory access latency and system performance.

The example in Figure 3 illustrates how row buffer locality-oblivious data placement can result in suboptimal application performance. In this figure, there are twelve requests which arrive at the memory system in the order shown and we display the service timelines of these requests assuming conventional data mapping and row buffer locality-aware data mapping. For simplicity, we assume that requests from the processor access rows A and B infrequently in such a way that requests to those rows never hit in the row buffer (low row buffer locality) and access rows C and D in such a way that the requests to data in the same row occur one after the other and frequently hit in the row buffer (high row buffer locality). If these requests were issued from the same core, servicing them more quickly would allow the waiting core to stall less, resulting in improved performance; if these requests were from different cores, servicing them more quickly would allow other requests to in turn be serviced, improving system performance. Without loss of generality, we assume a single core system where DRAM and PCM each have a single bank.

With a *conventional data mapping* scheme which is unaware of row buffer locality (such as any prior proposed caching policy [23, 9]), it is possible to map rows A and B to PCM and rows C and D to DRAM (as shown in the top half of Figure 3). Given the access pattern described, requests to rows A and B will always miss in the PCM row buffer and access the PCM device at a high latency. Requests to rows C and D already hit frequently in the row buffer and thus do not receive much benefit from being placed in DRAM versus being placed in PCM where the row buffer hit latency is the same. Note that the processor stalls for six PCM device access latencies and can only proceed after all these misses are serviced.

In contrast, a *row buffer locality-aware* caching mechanism would cache rows with high row buffer locality in PCM and low row buffer locality in DRAM (bottom half of Figure 3). This caching decision leads to a reduced number of

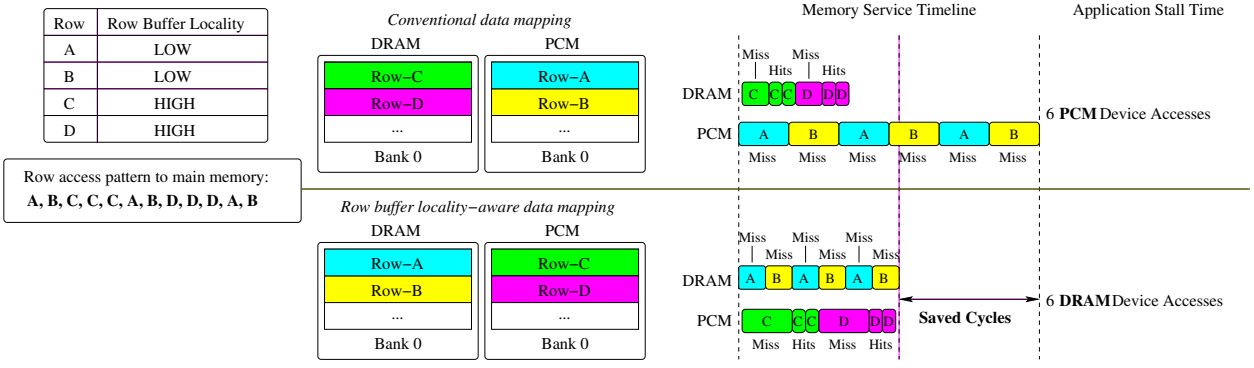


Figure 3: Conceptual example showing the importance of including row buffer locality-awareness in hybrid memory data placement decisions.

cycles the core is stalled while waiting for the last memory request to return. This is because the data for accesses which frequently lead to row buffer misses are placed in DRAM, where access latencies are less than in PCM, and system performance improves because these requests are serviced faster.

The crucial observation is that *both DRAM and PCM devices employ row buffers which can be accessed at similar latencies:* placing data which is frequently accessed in a row missing manner in DRAM can reduce application stall time, while leaving data which frequently hit in the row buffer in PCM will not increase application stall time by much more than if that data were placed in DRAM.

Our goal is to develop a mechanism that identifies rows with low row buffer locality which would benefit from being placed in the DRAM cache. Based on the observation that both DRAM and PCM devices employ row buffers, we design a dynamic caching policy for hybrid main memories which identifies data that frequently incur row buffer misses and caches such data in DRAM.

4 Row Buffer Locality-Aware Caching

Overview. Ideally, we would like to cache in DRAM rows which have little row buffer locality and exhibit high reuse. To do so, our *Row Buffer Locality Aware (RBLA)* mechanism tracks the number of row buffer misses and accesses a row in PCM experiences. The key idea is that the rows with more row buffer misses and accesses in a given amount of time will be those that have low row buffer locality and are frequently reused. In our mechanism, when the number of row buffer misses and accesses for a row exceed certain thresholds, the row is cached in DRAM. We find that static threshold numbers of misses and accesses are not sufficient for adapting to the varying workload and system characteristics of a many-core CMP, and so we develop a dynamic policy, *DynRBLA*, which is designed to choose appropriate thresholds at runtime.

4.1 Measuring Row Buffer Locality and Reuse

To measure row locality and reuse information, we use a small structure called the *statistics store*, or simply *stats store*, in the memory controller which monitors the number of row buffer misses and accesses which have occurred for each of a subset of most recently used rows in PCM. The stats store is organized similar to a cache, though it only contains only several bits of row buffer miss and access data per entry.

The stats store works as follows (Figure 4). Each memory request serviced in PCM has associated with it a row address and a row buffer hit or miss status determined by the PCM memory controller. Whether the request generates a row buffer hit or miss depends on the contents of the row buffer at the bank where the request is serviced at the time it is serviced. The row address of the request is used to index an entry in the stats store. If the entry is valid, the tag matches that of the request, and the request generates a row buffer hit (Figure 4a), the value of its access counter is incremented (shown as shaded entry with bold text in the figure). On the other hand, if the request generates a row buffer miss, (Figure 4b), the value of both its miss and access counters are incremented. If an entry for the corresponding row is not found in the stats store (Figure 4c), an invalid entry is allocated (or the least recently used entry is reassigned) and its miss and access counts are initialized to 1 for the row buffer miss that generated the stats store access.

The miss and access counters in the stats store approximate the row buffer locality and reuse of the row they tracks because rows with lower row buffer locality will miss more

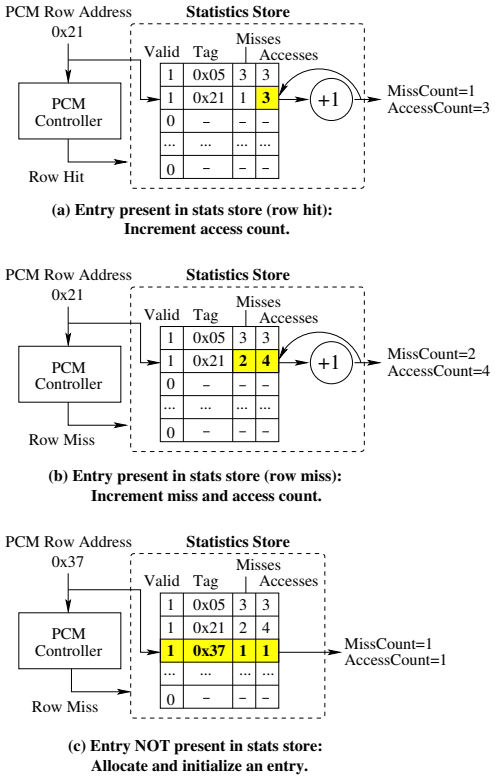


Figure 4: Stats store organization and operation.

frequently in the row buffer and have larger miss counter values while rows which have higher reuse will have larger access counter values. Rows which have both low row buffer locality and high reuse will have high values for both counters. We leverage this observation for designing a mechanism for caching rows.

4.2 Determining When to Cache Rows

RBLA caches a row when its number of row buffer misses and accesses (as reported by the stats store) exceed threshold values, `MissThresh` and `AccThresh`. At this point, the memory controller triggers a migration of the row to DRAM and invalidates the corresponding stats store entry.

The `MissThresh` parameter determines the amount of row buffer locality a row must exhibit in order to be cached. Larger values of `MissThresh` will favor caching rows with less row buffer locality. The `AccThresh` parameter, on the other hand, determines the amount of reuse a row must exhibit in order to be cached. Larger values of `AccThresh` will favor caching rows with more reuse. However, setting `MissThresh` or `AccThresh` too high can overly restrict the amount of data that is cached. Because the best threshold configuration for a system can change depending on co-running programs and system characteristics, in the next section we will describe a dynamic mechanism for adjusting our mechanism at runtime.

Caching rows based on their row buffer locality and reuse attempts to ensure that data is only migrated when it is useful to do so. This affects system performance in several ways. First, placing in DRAM rows which have low row buffer locality improves average memory access latency, due to DRAM's lower row miss latency. Second, by selectively caching data, RBLA reduces data movement while still caching beneficial data, reducing memory bandwidth consumption and allowing more bandwidth to be used to service demand requests. Third, caching data with high reuse helps balance the memory request load across both DRAM and PCM.

To prevent rows with little temporal reuse from gradually, over long amounts of time, building up large enough row miss and access counts in the statistics store to exceed `MissThresh` and `AccThresh`, we can periodically apply a decay factor to the values in the statistics store. We find resetting the miss and access counters every 10 million cycles to perform well.

4.3 Dynamic Threshold Adaptation

To improve the resiliency of RBLA to workload and system variations, we propose a technique which dynamically tunes our mechanism at runtime, DynRBLA. The key observation behind our technique is that the number of cycles saved by caching rows in DRAM should outweigh the bandwidth cost of migrating that data to DRAM. Our mechanism estimates, over an interval, the first order costs and benefits of employing the given threshold values and uses this information to make adjustments, aiming to maximize the net benefit.

Row migrations are costly because they occupy the shared memory channel for long amounts of time, preventing other requests from being serviced. However, they also have the benefit of potentially caching data which will be serviced from the DRAM cache, reducing average memory access latency. To take this into account when dynamically adjusting our mechanism, over the course of an interval (10 million cycles), we monitor the number of row migrations performed (a component of cost) as well as the number of reads and writes serviced by the DRAM cache (a component of benefit). We quantify the cost of using a given set of threshold values as the number of migrations generated times the latency per migration (Equation 1) and the benefit of using the threshold values as the aggregate number of cycles saved due to accessing data in DRAM at a lower latency instead of PCM (Equation 2).

$$Cost = Migrations \times t_{migration} \tag{1}$$

$$Benefit = Reads_{DRAM} \times (t_{read,PCM} - t_{read,DRAM}) + Writes_{DRAM} \times (t_{write,PCM} - t_{write,DRAM}) \tag{2}$$

A negative net benefit ($Benefit - Cost$) implies that with the current threshold values, more cycles are spent migrating data (preventing the memory channels from being used to service other requests) than the amount of latency saved as a result of data which has been placed in the DRAM cache. In other words, the small benefit of the data being cached does not justify the large cost of migrating that data. In such a situation, it is amenable to throttle back the number of migrations being performed while still migrating the data which achieve the most benefit from being placed in the DRAM cache. This can be done within RBLA by *increasing* the value of `MissThresh`, `AccThresh`, or both, which has the effect of requiring rows to have even lower row buffer locality and/or higher reuse before being cached.

Conversely, a positive net benefit implies that the data migrated under the current thresholds are accessed many times and save more cycles than their migration cost. This is beneficial for system performance, but there is still an opportunity cost associated with a given set of thresholds yielding a positive net benefit: migrating more data may expose even more benefit in terms of accesses to DRAM, perhaps for only a small amount of additional bandwidth cost. In such a situation, *decreasing* the value of `MissThresh`, `AccThresh`, or both, can allow more rows to be migrated by relaxing the constraints on row buffer locality and/or reuse. We leverage these observations to develop an algorithm to tune our mechanism.

To reduce the search space of our algorithm without reducing its effectiveness, we adjust only `AccThresh`. We empirically observed that for our benchmarks, setting `MissThresh` = 2 achieves the best average performance for all `AccThresh` and `MissThresh` values in the range 1 to 10. Intuitively, values of `MissThresh` > 2 require suffering additional row buffer misses in PCM before making a decision to cache a row, and those row buffer misses could have been serviced in DRAM had the row been cached more quickly. On the other hand, setting `MissThresh` = 1 does not consider row buffer locality information at all. We find setting `MissThresh` = 2 to offer a good indication of low row buffer locality without paying the cost of too many row buffer misses before making a caching decision.

Tying everything together, our DynRBLA technique uses a simple hill-climbing algorithm, evaluated at the end of each quantum, to adjust `AccThresh` in the direction of increasing net benefit, ultimately converging at point of maximum net benefit, as follows (Algorithm 1). If, during the previous quantum, net benefit is negative, the cost of using the current `AccThresh` outweighs the benefit of the accesses to the DRAM cache, and `AccThresh` is incremented to filter out rows which do not benefit as much from being placed in the DRAM cache. Else, during the previous quantum the net benefit was positive. If the net benefit from the previous quantum is greater than the net benefit from the quantum *before* that, it means that benefit is increasing and our algorithm increments `AccThresh` to explore if net benefit will continue increasing. Otherwise, if net benefit is decreasing, we decrement `AccThresh`, reverting to the previous threshold value (which, as of two quanta ago, achieved the largest net benefit). Our algorithm assumes net benefit as a function of `AccThresh` is concave, which, across all the benchmarks we have tested, is the case.

```
Each quantum (10 million cycles):
1  NetBenefit = Benefit − Cost
2  if NetBenefit < 0 then
3      AccThresh++
4  else
5      if NetBenefit > PreviousNetBenefit then
6          AccThresh++
7      else
8          AccThresh--
9      end
10 end
11 PreviousNetBenefit = NetBenefit
```

Algorithm 1: Dynamic Threshold Adjustment

5 Implementation and Hardware Cost

Row Migration: The DRAM cache is mapped as a region of the virtual address space and the OS is responsible for updating the virtual addresses of rows migrated to and from the DRAM cache. For example, when an entry for a row in the statistics store exceeds the `MissThresh` and `AccThresh` values, the memory controller raises an interrupt to the OS to trigger a migration of the row. The OS is in charge of managing the DRAM cache (we assume a 16-way set associative organization) and invalidating TLB entries for migrated pages.

Statistics Store: We present results for a stats store of unlimited size. We do this to observe the effects of different promotion policies in isolation of implementation-dependent stats store parameters. For reference, across 50 randomly-generated 16-core workloads, we find that a 16-way 32-set LRU-replacement stats store (2.3 KB total) with LRU replacement achieves performance within 4% of an unlimited-sized stats store.

6 Related Work

To our knowledge, this is the first work that observes row buffer hits are the same latency in different memory technologies, and devises a caching policy that takes advantage of this observation to improve system performance. No previous work, as far as we know, considered row buffer locality as a key metric that influences caching decisions.

Selective Caching Based on Frequency of Data Reuse: Jiang et al. [9] proposed only caching data (in block sizes of 4 to 8 KB) that experience a high number of accesses in an on-chip DRAM cache, to reduce off-chip memory bandwidth consumption. Johnson and Hwu [10] used a counter-based mechanism to keep track of data reuse at a granularity larger than a cache block. Cache blocks in a region with less reuse bypass a direct-mapped cache if that region conflicts with another that has more reuse. We propose taking advantage of row buffer locality in memory banks when dealing with off-chip DRAM and PCM. We exploit the fact that accesses to DRAM and PCM have same latencies for rows that have high row buffer locality.

Ramos et al. [23] adapted a buffer cache replacement algorithm to rank pages based on their frequency and recency of accesses, and placed the highest-ranked pages in DRAM, in a DRAM-PCM hybrid memory system. Our work is orthogonal, and we expect improved performance for similarly ranking pages based on their frequency and recency of row buffer miss accesses.

Improving Data Locality between On-Chip Caches: Gonzalez et al. [8] proposed predicting and segregating data into either of two last-level caches depending on whether they exhibit spatial or temporal locality. They also proposed bypassing the cache when accessing large data structures (i.e., vectors) with large strides to prevent cache thrashing. Rivers and Davidson [25] proposed separating data with temporal reuse from data without temporal reuse. If a block that has not been reused is evicted from the L1 cache, a bit indicating lack of temporal reuse is set for that block in the L2 cache. If blocks with such a bit set are referenced again, they are placed in a special buffer instead of the L1 cache to prevent them from polluting the L1 cache. These works are primarily concerned with L1/L2 caches that have access latencies on the order of a few to tens of CPU clock cycles, where off-chip memory bank row buffer locality is less applicable.

Hybrid Memory Systems: Qureshi et al. [21] proposed increasing the size of main memory by using PCM as main memory, and using a DRAM as a cache to PCM. The reduction in page faults due to the increase in main memory size brings performance and energy improvements to the system. We study the effects past the reduced page faults, and propose an effective DRAM caching policy to PCM.

Dhiman et al. [6] proposed a hybrid main memory system that exposes DRAM and PCM addressability to software (OS). If the number of writes to a particular PCM page exceeds a certain threshold, the contents of the page are copied to another page (either in DRAM or PCM), thus facilitating PCM wear-leveling. Mogul et al. [16] suggested the OS exploit metadata information available to it to make data placement decisions between DRAM and non-volatile memory. Similar to [6], their data placement criteria are centered around write frequency to the data (time to next write, or time between writes to page).

Bivens et al. [3] examined the various design concerns of a heterogeneous memory system such as memory latency, bandwidth, and endurance requirements of employing storage class memory (PCM, MRAM, Flash, etc.). Their hybrid memory organization is similar to ours and that in [21], in that DRAM is used as a cache to a slower memory medium, transparently to software. Phadke et al. [20] proposed profiling the memory access patterns of individual applications in a multicore system, and placing their working sets in the particular type of DRAM that best suits the application's memory demands. In constrast, our mechanism dynamically makes fine-grained data placement decisions at a row granularity, depending on the row buffer locality characteristics.

Exploiting Row Buffer Locality: Lee et al. [13] proposed using multiple short row buffers in PCM devices, much like an internal device cache. This works in synergy with our proposal, as multiple row buffers increase PCM row buffer hit rate and our mechanism migrates rows with low row buffer locality to DRAM.

Row buffer locality is commonly exploited in memory scheduling algorithms. The *First-Ready First Come First Serve algorithm (FR-FCFS)* [26, 31] prioritizes memory requests that hit in the row buffer, therefore improving the latency, throughput, and energy cost of servicing memory requests. Many other memory scheduling algorithms [18, 17, 11, 12] build upon this "row-hit first" principle.

7 Experimental Methodology

We use an in-house, closed-loop CMP simulator to evaluate our proposed mechanisms. Table 1 describes the details of the processor and memory models, and lists key system parameters of our simulations.

Our workload selection is from the SPEC CPU2006 benchmark suite. We simulate the representative phase of the program, generated by PinPoints [19]. We categorize benchmarks into two groups depending on the sizes of their working sets, which we regard as the aggregate size of rows an application accesses throughout its execution. Table 2 lists the benchmarks along with their application characteristics.

<table>
<tr><td>System organization</td><td>16 cores; 2 on-chip memory controllers (1 for DRAM, 1 for PCM).</td></tr>
<tr><td>Processor pipeline</td><td>Out-of-order issue; 128-entry instruction window; fetch/execute/commit 3 instructions per cycle (max. 1 memory op per cycle).</td></tr>
<tr><td>L1 cache</td><td>Private; 32 KB per core; 4-way set-associative; 128 B blocks.</td></tr>
<tr><td>L2 cache</td><td>Shared; 512 KB per core; 8-way set-associative; 128 B blocks.</td></tr>
<tr><td>Memory controller</td><td>DDR3 1066 MT/s; 128-entry request buffer; 128-entry write data buffer; FR-FCFS [26] scheduling; open row policy; row interleaving address mapping (consecutive cache blocks make up rows and rows are striped across banks); 512 cycles to migrate a 2 KB row.</td></tr>
<tr><td>DIMMs</td><td>1 DRAM rank; 2 PCM ranks (1 rank for 1, 2, or 4 cores in system); 8 devices on a DIMM.</td></tr>
<tr><td>DRAM</td><td>256 MB (reduced-scale) or 1 GB (full-scale); capacity scales with core count; 8 banks per chip; 2 KB row buffer per bank.
<table>
<tr><td>Latency</td><td>ns</td><td>cycles</td></tr>
<tr><td>Row buffer hit</td><td>40</td><td>200</td></tr>
<tr><td>Row buffer miss</td><td>80</td><td>400</td></tr>
</table>
<table>
<tr><td>Dyn. energy (pJ/bit)</td><td>Read</td><td>Write</td></tr>
<tr><td>Row buffer access</td><td>0.93</td><td>1.02</td></tr>
<tr><td>Cell array access</td><td>1.17</td><td>0.39</td></tr>
</table></td></tr>
<tr><td>PCM</td><td>8 GB (reduced-scale), or 32 GB (full-scale); capacity scales with core count; 8 banks per chip; 2 KB row buffer per bank.
<table>
<tr><td>Latency</td><td>ns</td><td>cycles</td></tr>
<tr><td>Row buffer hit</td><td>40</td><td>200</td></tr>
<tr><td>Row buffer miss (clean)</td><td>128</td><td>640</td></tr>
<tr><td>Row buffer miss (dirty)</td><td>368</td><td>1840</td></tr>
</table>
<table>
<tr><td>Dyn. energy (pJ/bit)</td><td>Read</td><td>Write</td></tr>
<tr><td>Row buffer access</td><td>0.93</td><td>1.02</td></tr>
<tr><td>Cell array access</td><td>2.47</td><td>16.82</td></tr>
</table></td></tr>
</table>

Table 1: Simulator model and parameters. Memory latencies and energies are based on [13]. The latencies conservatively include overheads for interconnect traversal from the L2 cache to the memory controller, synchronization and clocking overhead at the DRAM interface, and ECC overhead in DRAM.

Benchmark	RBHR	MPKI	WS (MB)	L/S	Benchmark	RBHR	MPKI	WS (MB)	L/S
milc	0.56	13.0	359.6	L	gobmk	0.48	0.60	8.0	S
astar	0.52	4.3	268.7	L	gromacs	0.61	0.65	6.3	S
GemsFDTD	0.41	13.1	255.3	L	gcc	0.46	0.16	4.9	S
lbm	0.86	25.0	180.4	L	bzip2	0.69	3.50	3.8	S
leslie3d	0.55	11.0	73.9	L	perlbench	0.59	0.05	3.0	S
sjeng	0.21	0.4	70.3	L	h264ref	0.79	0.99	2.9	S
omnetpp	0.10	18.1	54.7	L	hmmer	0.48	2.79	2.1	S
cactusADM	0.14	3.1	32.7	L	dealII	0.75	0.07	1.8	S
libquantum	0.94	13.2	32.0	L	namd	0.78	0.07	1.7	S
xalancbmk	0.44	15.1	29.0	L	wrf	0.80	0.14	1.4	S
soplex	0.73	22.6	22.3	L	calculix	0.67	0.03	1.0	S
mcf	0.13	57.0	22.1	L	povray	0.72	0.01	0.5	S
sphinx3	0.53	7.43	13.6	S	tonto	0.78	0.01	0.4	S

Table 2: Benchmark characteristics (200 M inst.; RBHR = Row Buffer Hit Rate; MPKI = last-level cache Misses Per Kilo Instructions; WS = Working Set size; L/S = large/small working set; sorted in descending order of working set size).

To identify trends over a large range of workloads within a reasonable amount of time, we perform most of our evaluations in reduced scale by using 200 million instructions per benchmark and using a 256 MB DRAM cache size. Using combinations of the benchmarks, we form multi-programmed workloads for multi-core evaluations. We form different workload groups by varying the proportion of large working set benchmarks in a workload (0%, 25%, 50%, 75%, 100%). We use 100 random combinations in each workload group to show the variation in performance and energy efficiency according to the aggregate working set size. We use 200 random combinations (not fixing the proportion of large benchmarks) for the various sensitivity studies.

To verify the trends we find, we also perform fewer large scale evaluations using 2 billion instructions per benchmark and a 1 GB DRAM cache size, and confirm that our reduced-scale evaluations yield similar results (as shown in Section 8.2.3). We use 8 random combinations of the large working set benchmarks in order to stress the DRAM cache.

7.1 Evaluation Metrics

To gauge multi-core performance, we use the *weighted speedup* metric [27]. The speedup of a benchmark is its *Instructions Per Cycle (IPC)* when executing simultaneously with other benchmarks on the system, divided by its IPC

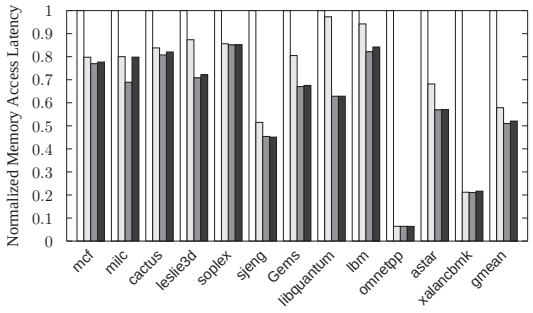


Figure 6: Average memory request access latency.

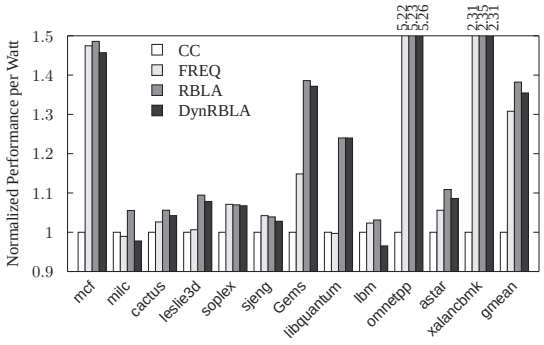


Figure 7: Single core energy efficiency.

when executing alone on the system, $Speedup = \frac{IPC_{together}}{IPC_{alone}}$. The weighted speedup is the sum of the speedups of all the benchmarks executing on the system, $WeightedSpeedup = \sum \frac{IPC_{together}}{IPC_{alone}}$.

For multi-core fairness, the *maximum slowdown* metric [2] is used. The slowdown of a benchmark is the reciprocal of its speedup, $Slowdown = \frac{IPC_{alone}}{IPC_{together}}$. The maximum slowdown is the highest slowdown experienced by any benchmark executing in the system, $MaxSlowdown = \max \frac{IPC_{alone}}{IPC_{together}}$. In addition, we report the *harmonic speedup* metric [15] (or the harmonic mean of individual benchmark speedups) that has been proposed to measure both multicore performance and fairness, where $HarmonicSpeedup = \frac{NumBenchmarks}{\sum \frac{IPC_{alone}}{IPC_{together}}}$.

Unless otherwise noted, IPC_{alone} is measured on a DRAM-PCM hybrid memory system that manages its DRAM cache using a conventional caching policy (cache a row into DRAM on the first access to the row).

8 Experimental Results

8.1 Single Core Case Study

We first analyze the benefit of using row buffer locality information to make caching decisions compared to previous caching techniques with a case study on a single core system. Aggregate results over 200 16-core workloads are provided in Section 8.2.

Figure 5 shows the normalized performance of each benchmark compared to different caching policies. We make several observations.

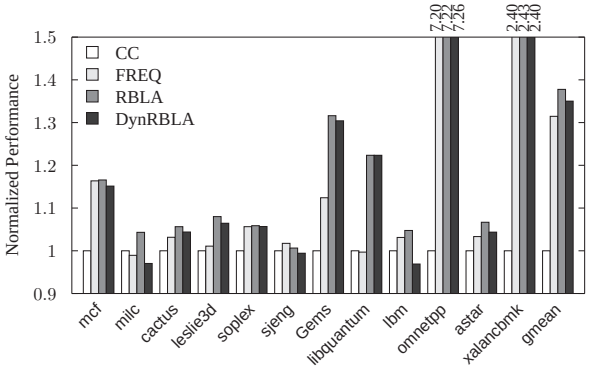


Figure 5: Single core workload performance.

Conventional Caching (CC): The commonly-used conventional caching policy (employed in most on-chip caches and some prior hybrid memory works such as [21]) simply caches every row that is accessed. Such a caching policy leads to a large amount of memory channel bandwidth spent migrating data. Figure 8 shows, for each benchmark, the fraction of time its DRAM and PCM channels were active for each of the caching policies. For all benchmarks, conventional caching leads to the most amount of time spent migrating data. This large channel bandwidth utilization can delay demand memory requests, increasing average memory access latency (shown normalized in Figure 6), and leading to low performance and low energy efficiency (Figure 7).

Frequency-Based Caching (FREQ): The frequency-based caching policy is similar in approach to caching policies employed by prior works in on-chip and hybrid memory caching, such as [23, 9], which try to improve spatial locality in the cache and reduce off-chip bandwidth consumption by monitoring the reuse of data before making a caching decision. The key idea is that data which are accessed many times in a short interval will likely continue to be accessed and thus would benefit from being cached. Therefore, FREQ caches a row when the number of accesses it receives exceeds a certain threshold value, `FreqThresh`. We statically profiled the benchmarks and show

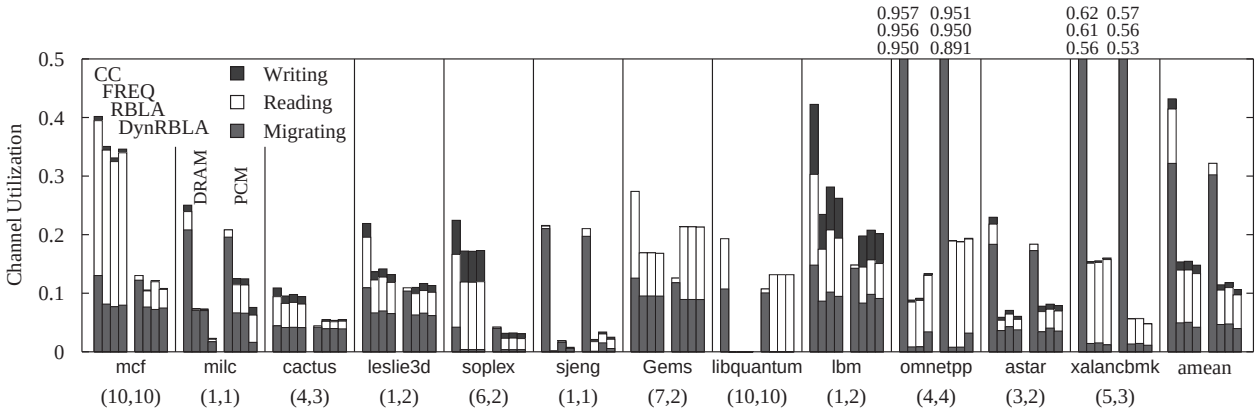


Figure 8: Memory channel utilization. Numbers below workload names indicate the best static `FreqThresh` for FREQ and `AccThresh` for RBLA. The best `MissThresh` for all benchmarks was 2. For bar heights which fall outside the plotted range, numeric values above the graph indicate the heights of the bars in the same order as in the legend.

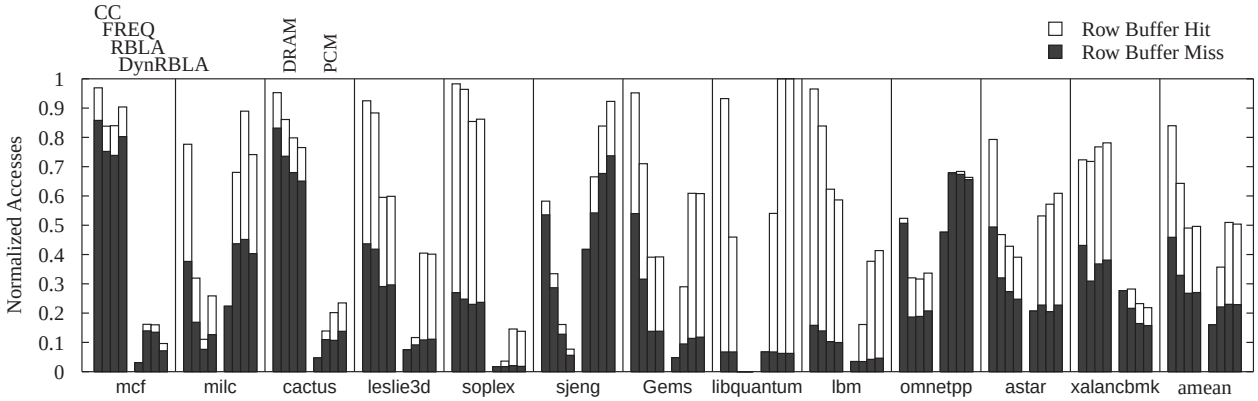
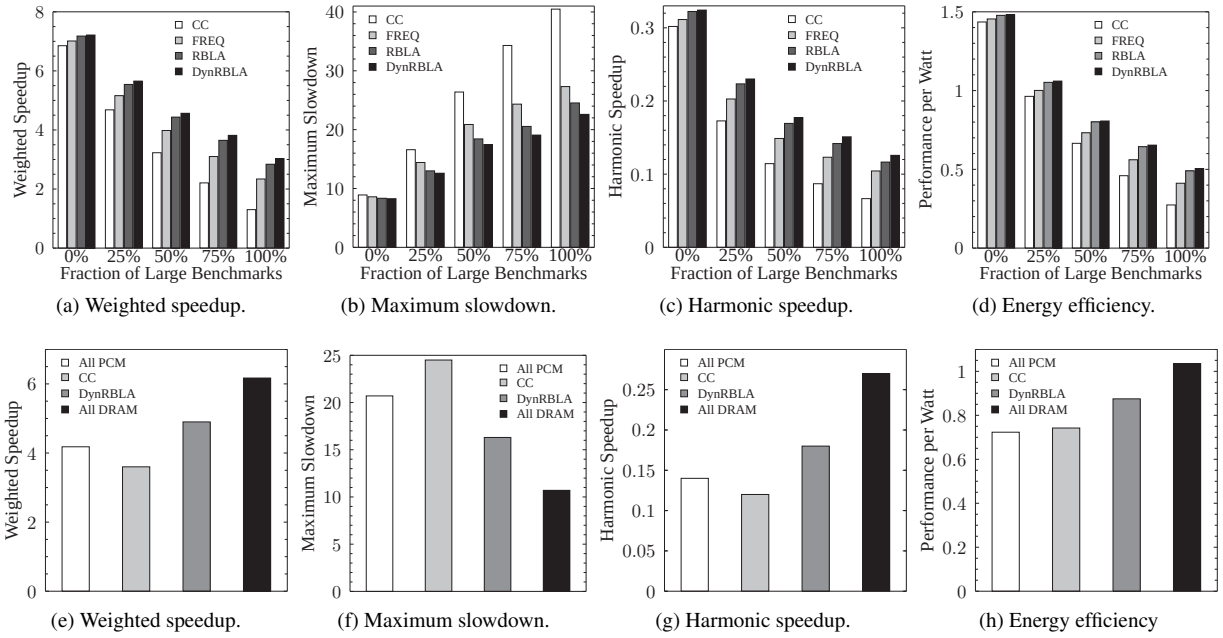


Figure 9: Normalized number of accesses to each memory device broken into row buffer hits and misses.

results for the best `FreqThresh` value for each benchmark. Compared to conventional caching, FREQ reduces the amount of time spent migrating data which generally improves average memory access latency and leads to improved performance and energy efficiency compared to conventional caching.

Row Buffer Locality-Aware Caching (RBLA): Our approach, RBLA, migrates less data, reduces average memory access latency more, and improves performance and energy efficiency more than the previous policies. Like FREQ, we statically profiled the benchmarks and only show results for the best `MissThresh` and `AccThresh` values for each benchmark. In addition to achieving more bandwidth reductions than FREQ, Figure 9 shows that RBLA simultaneously improves row buffer locality in PCM (seen by the increasing proportion of row buffer hits to row buffer misses in PCM), leading to a reduction in average memory access latency. Furthermore, RBLA more evenly balances the load of requests between the DRAM and PCM devices (shown by the total height of the bars in Figure 9). Energy efficiency benefits come not only from improvements in performance, but also from the fact that more requests in PCM are serviced at a lower energy from the row buffer and fewer data migrations occur.

Dynamic RBLA (DynRBLA): DynRBLA provides performance and energy efficiency similar to the best static RBLA configurations for each application by adjusting `AccThresh` at runtime. As we will show in the next section, the ability of DynRBLA to adapt to workload and system characteristics enables it to achieve the best performance, fairness, and energy efficiency on a 16-core system.



(a) Weighted speedup. (b) Maximum slowdown. (c) Harmonic speedup. (d) Energy efficiency.

(e) Weighted speedup. (f) Maximum slowdown. (g) Harmonic speedup. (h) Energy efficiency

Figure 10: Performance, fairness, and energy efficiency results for a 16-core system comparing the various caching techniques (a–d) and versus a system with all PCM and all DRAM (e–h).

8.2 DynRBLA on a 16-Core System

The main memory system will become an increasingly bigger bottleneck as the number of cores sharing it increases and the data set sizes of applications running on those cores increase. We next examine the performance of DynRBLA on a 16-core system. For the static threshold techniques, we set `FreqThresh`, `MissThresh`, and `AccThresh` to the best static values on average from the single core study, which we found to perform well.

Figure 10 shows performance, fairness, and energy efficiency for different caching policies for workloads grouped based on the fraction of large benchmarks, as well as for an all PCM and all DRAM system.

8.2.1 Analysis

As Figure 10(a) shows, DynRBLA provides the best performance among all evaluated caching techniques. We make several observations and provide insight into the resulting trends:

First, as the fraction of large benchmarks in the workload increases, the benefit of considering row buffer locality when making caching decisions increases. Though FREQ, when compared to conventional caching, reduces the amount of memory channel bandwidth consumed by not migrating data which has little reuse, row buffer locality-aware policies are able to offer further benefits at less bandwidth consumption by caching rows which exhibit high reuse and frequent row buffer misses, making better use of the available bandwidth and DRAM space.

Second, DynRBLA performs better than the static RBLA policy. This is because DynRBLA is able to adjust its caching policy to suit the needs of each workload at runtime as opposed to assuming one fixed threshold for every type of workload.

Third, the RBLA-based policies are beneficial even for workloads with small memory footprints because they balance the load of memory requests between both the DRAM and PCM devices. Servicing requests from PCM does not degrade performance for row buffer locality-aware policies because rows with high row buffer locality can be accessed at the same latency as DRAM.

DynRBLA is able to provide both the smallest maximum slowdown (Figure 10b) as well as the best balance between performance and fairness according to the harmonic speedup metric (Figure 10c)[1]. In general, the policies examined improve fairness by a greater amount as the fraction of large benchmarks in the workload mixes increase.

[1]DynRBLA also achieves the best *average* slowdown of 5.2, compared to 7.8 for CC, 6.1 for FREQ, and 5.5 for RBLA.

Such workloads access large amounts of data and cause greater interference between different programs accessing DRAM and PCM. Taking into account reuse alone, as FREQ does, can reduce the strain on the shared memory bandwidth, however, our row buffer locality-aware techniques are able to provide increased fairness by not only reducing the amount of bandwidth consumed due to migrations, but by also by reducing the average memory access latency, overall reducing resource contention among the threads.

Figure 10(d) shows that DynRBLA achieves the best main memory energy efficiency, in terms of performance per Watt, compared to the other examined caching techniques. This is because DynRBLA reduces the number of PCM array reads and writes by leaving in PCM data which frequently hit in the row buffer where they can be serviced at a lower energy, as well as due to the fewer number of data migrations which are performed

We conclude that DynRBLA is very effective in providing the best performance, fairness, and energy efficiency in 16-core systems.

8.2.2 Comparison to Systems Using All PCM or All DRAM

Figure 10(e–h) shows performance, fairness, and energy efficiency for DynRBLA compared to aggressive all PCM and all DRAM main memory systems. For the all PCM and all DRAM systems, we model infinite memory capacity, to fit the entire working sets of the workloads. Data are mapped to two ranks, totaling sixteen banks.

In the all PCM system, all row buffer miss accesses incur high latencies and energies to access PCM, while in DRAM, the opposite is true. Note that an ill-designed caching scheme—such as conventional caching—can have worse performance and fairness compared to an all PCM system. This is due to the high amount of channel contention introduced and the inefficient use of the DRAM cache. The same system with the DynRBLA policy makes efficient use of the DRAM, PCM, and memory channel bandwidth by caching data with low row buffer locality and high reuse.

DynRBLA achieves within 21% of the weighted speedup, 53% of the maximum slowdown, and 31% of the harmonic speedup of a system with an unlimited amount of DRAM. Compared to a system with an all PCM main memory, DynRBLA improves weighted speedup by 17%, reduce maximum slowdown by 21%, and improve harmonic speedup by 27%.

8.2.3 Sensitivity to System Configuration

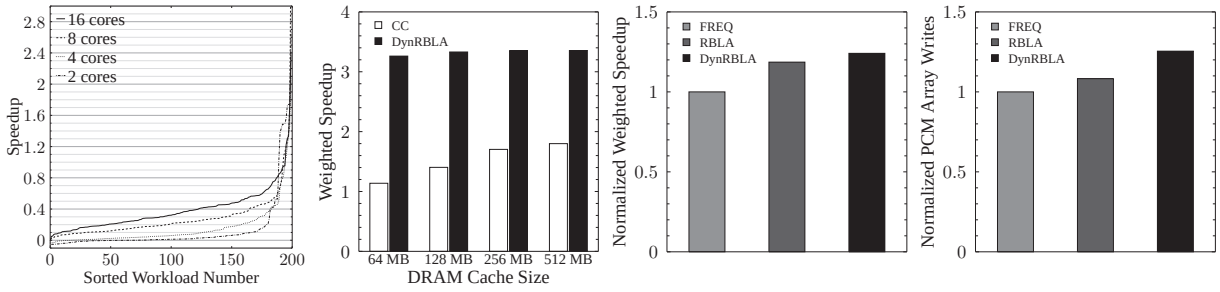
We varied the salient parameters of our hybrid main memory system and examined how they affected the performance of DynRBLA.

Number of Cores: Figure 11(a) shows the speedup of DynRBLA on 2, 4, 8, and 16-core systems, normalized to conventional caching. Results are sorted in terms of increasing speedup. Overall, DynRBLA achieves good performance improvements across a range of core counts and workloads and scales well by providing greater performance benefit as the number of cores increase.

DRAM Cache Size: Figure 11(b) shows the performance DynRBLA for DRAM cache sizes from 64 to 512 MB, averaged across workloads consisting entirely of large benchmarks, in order to exercise the DRAM cache. There are two things to note. First, even when a large portion of the working set can fit in the cache (e.g., the 512 MB bar in Figure 11b), DynRBLA outperforms conventional caching. This is because DynRBLA reduces the amount of channel bandwidth consumption and does a better job of balancing the load of memory requests between DRAM and PCM. Second, for small cache sizes, DynRBLA provides the most performance benefit compared to conventional caching. At smaller capacities or with more data intensive applications, prudent use of DRAM cache space becomes more important, and DynRBLA is able to dynamically adjust the amount of data cached, ensuring that only the data with low row buffer locality and high reuse get cached.

To verify that our conclusions hold at even larger DRAM cache sizes, we performed large scale simulations of a 16-core system running workloads composed of large benchmarks with a 1 GB DRAM cache. Our results are presented in Figure 11(c). We can see that our techniques still achieve similar performance improvements as in the reduced scale evaluations.

For the same large scale evaluation, we show PCM array writes in Figure 11(d) normalized to FREQ. As can be seen, one trade off of DynRBLA is that it may increase the number of writes performed in a given amount of time to the PCM array. There are two reasons why DynRBLA incurs more writes in PCM. First, in the same amount of time, DynRBLA executes more instructions and generates more write traffic than the other caching policies. Second, by leaving data with high row buffer locality in PCM, more accesses, and thus writes, must serviced in that device.



(a) Varying number of cores. (b) Varying DRAM size. (c) Larger-scale simulations. (d) PCM array writes.

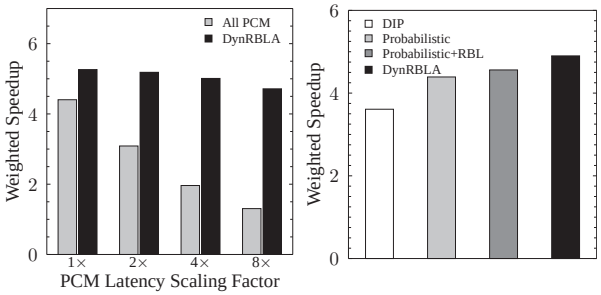
Figure 11: Sensitivity studies for 16-core workloads varying number of cores and DRAM cache size.

As future work, we plan on adapting our technique to take into account the read/write intensity of a row as another component of our caching policy.

PCM Latency: Figure 12(a) shows the performance of our technique as the technology-dependent latencies associated with PCM (array read and array write) are varied from to 1 to 8× the values in Table 1. We compare our technique to the all PCM system described in Section 8.2.2 and the IPC_{alone} component of weighted speedup is measured for the benchmarks run alone on the all PCM system. There are two trends to notice. First, as the latency required to access the PCM memory array increases, the performance of the all PCM system decreases less than linearly (compared to the scaling factor). This is because aspects of the memory system such as the row buffer and bank-level parallelism can help hide some of the long access latencies of PCM. Second, our technique performs well even when the latencies associated with PCM increase. For example, at 8× the PCM latency of our baseline system, our technique slows down by less than 11%. This is because DynRBLA caches data with low row buffer locality in DRAM, avoiding large PCM access latencies, and keeps in PCM data that have high row buffer locality which can be serviced at the same latency as DRAM.

8.2.4 Comparison to On-Chip Caching Policies

Dynamic Insertion Policy (DIP): The key idea of DIP [22] is that blocks which are not reused quickly should be removed from the cache as early as possible. To achieve this effect, DIP inserts blocks into the LRU position of a set with a high probability and only with a small probability into the MRU position. In Figure 12(b), we find DIP to perform similarly to conventional caching in a hybrid memory system (compare with the CC bar in Figure 10e). This is because although DIP is able to quickly evict rows that exhibit low reuse in the DRAM cache, it still promotes any row on its first access like a conventional cache. Thus, it is unable to reduce the memory channel contention that results due to data migrations that yield low benefits. We find that DynRBLA increases system performance by 36% compared to using DIP.



(a) Varying PCM latency. (b) Related techniques.

Figure 12: Sensitivity of DynRBLA performance to PCM latency and comparison on-chip caching techniques.

Probabilistic Caching [7] inserts data blocks into the cache with a certain probability which determines the expected number of times a block must be accessed before it is cached. We leverage our key observation to introduce a variant that is row buffer locality aware which probabilistically promotes rows only on a row buffer miss. This causes rows that generate frequent row buffer misses to have a higher chance of being inserted into the DRAM cache. Figure 12(b) shows that probabilistic caching performs better than DIP because it makes its decision not to cache data before the data is transferred. The RBL-aware probabilistic scheme outperforms the purely probabilistic scheme because it caches data with lower row buffer locality, however, DynRBLA outperforms both by caching rows based on a more concrete measure of their row buffer locality. DynRBLA improves system performance compared to probabilistic caching and its RBL-aware variant by 12% and 8% respectively.

We conclude that the runtime adaptation of our DynRBLA scheme allows it to be robust to workload and hybrid memory variations and perform better than on-chip caching policies in 16-core systems.

9 Conclusion

We presented row buffer locality-aware caching policies for a DRAM-PCM hybrid memory system. Our proposed mechanisms improve system performance, fairness, and energy efficiency compared to caching policies that are only aware of the frequency of data access. DynRBLA selectively caches to DRAM rows that generate frequent row buffer misses. This scheme allows rows that are mostly accessed as row buffer hits, to take advantage of the large PCM capacity while being accessed at low latency and energy. The limited DRAM capacity is effectively utilized by selectively filling it with data that repeatedly incur memory array accesses. This also has the effect of balancing the memory request load between DRAM and PCM. Furthermore, row migrations between DRAM and PCM that are unbeneficial for performance are pruned, thus decreasing memory bandwidth consumption.

We evaluated our proposed mechanism and showed that it consistently yields performance improvements. We also demonstrated its robustness to high PCM latencies and restricted DRAM capacities. Furthermore, DynRBLA is able to dynamically adapt to workloads and system configurations to maintain high system throughput. We conclude that DynRBLA is a high performance, energy efficient means to manage hybrid memories for multicore systems.

Acknowledgments

We gratefully acknowledge members of the SAFARI research group and CALCM for many insightful discussions on this work. HanBin Yoon was supported by a Samsung Scholarship PhD fellowship while conducting this work. This research was partially supported by an NSF CAREER Award CCF-0953246, NSF Grant CCF-1147397, Gigascale Systems Research Center, Intel Corporation ARO Memory Hierarchy Program, and Carnegie Mellon CyLab. We also acknowledge equipment and gift support from Intel, Samsung, and Oracle.

References

[1] K. Bailey et al. Operating system implications of fast, cheap, non-volatile memory. HotOS, 2011.

[2] M. Bender et al. Flow and stretch metrics for scheduling continuous job streams. Symp. on Discrete Alg., 1998.

[3] A. Bivens et al. Architectural design for next generation heterogeneous memory systems. Intl. Memory Workshop '10.

[4] J. Coburn et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS'11.

[5] J. Condit et al. Better I/O through byte-addressable, persistent memory. SOSP, pages 133–146. ACM, 2009.

[6] G. Dhiman et al. PDRAM: a hybrid PRAM and DRAM main memory system. DAC. ACM, 2009.

[7] Y. Etsion et al. L1 cache filtering through random selection of memory references. PACT, pages 235–244, 2007.

[8] A. González et al. A data cache with multiple caching strategies tuned to different types of locality. ICS '95.

[9] X. Jiang et al. CHOP: Adaptive filter-based dram caching for CMP server platforms. HPCA, pages 1–12, 2010.

[10] T. L. Johnson and W.-m. Hwu. Run-time adaptive cache hierarchy management via reference analysis. ISCA '97.

[11] Y. Kim et al. ATLAS:a scalable and high-performance scheduling algorithm for multiple memory controllers. HPCA'10.

[12] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. MICRO '10.

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

[14] B. C. Lee et al. Phase-change technology and the future of main memory. *IEEE Micro*, 30, January 2010.

[15] K. Luo et al. Balancing thoughput and fairness in smt processors. ISPASS, pages 164 –171, 2001.

[16] J. C. Mogul et al. Operating system support for NVM+DRAM hybrid main memory. HotOS, 2009.

[17] O. Mutlu et al. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. ISCA'08.

[18] O. Mutlu et al. Stall-time fair memory access scheduling for chip multiprocessors. MICRO, 2007.

[19] H. Patil et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. MICRO'04.

[20] S. Phadke et al. MLP aware heterogeneous memory system. DATE, pages 1–6, march 2011.

[21] M. K. Qureshi et al. Scalable high performance main memory system using phase-change memory technology. ISCA'09.

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

[23] L. E. Ramos et al. Page placement in hybrid memory systems. ICS '11, pages 85–95.

[24] S. Raoux et al. Phase-change random access memory: a scalable technology. *IBM J. Res. Dev.*, 52, July 2008.

[25] J. Rivers and E. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. ICPP '96.

[26] S. Rixner et al. Memory access scheduling. ISCA, pages 128–138. ACM, 2000.

[27] A. Snavely et al. Symbiotic jobscheduling for a simultaneous multithreading processor. ASPLOS, 2000.

[28] The International Technology Roadmap for Semiconductors. Process integration, devices, and structures, 2010.

[29] H. Wong et al. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[30] W. Zhang et al. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. PACT, pages 101–112. IEEE Computer Society, 2009.

[31] W. K. Zuravleff et al. Controller for a synchronous dram that maximizes throughput by allowing memory requests and commands to be issued out of order. U.S. Patent Number 5,630,096, 1997.