Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management

Justin Meza* Jichuan Chang† HanBin Yoon* Onur Mutlu* Parthasarathy Ranganathan†
*Carnegie Mellon University †Hewlett-Packard Labs
{meza,hanbinyoon,onur}@cmu.edu {jichuan.chang,partha.ranganathan}@hp.com

Abstract—Hybrid main memories composed of DRAM as a cache to scalable non-volatile memories such as phase-change memory (PCM) can provide much larger storage capacity than traditional main memories. A key challenge for enabling high-performance and scalable hybrid memories, though, is efficiently managing the metadata (e.g., tags) for data cached in DRAM at a fine granularity. Based on the observation that storing metadata off-chip in the same row as their data exploits DRAM row buffer locality, this paper reduces the overhead of fine-granularity DRAM caches by only caching the metadata for recently accessed rows on-chip using a small buffer. Leveraging the flexibility and efficiency of such a fine-granularity DRAM cache, we also develop an adaptive policy to choose the best granularity when migrating data into DRAM. On a hybrid memory with a 512MB DRAM cache, our proposal using an 8KB on-chip buffer can achieve within 6% of the performance of, and 18% better energy efficiency than, a conventional 8MB SRAM metadata store, even when the energy overhead due to large SRAM metadata storage is not considered.

Index Terms—Cache memories, tag storage, non-volatile memories, hybrid main memories.

1 Introduction

As feature sizes continue to shrink, future chip multi-processors are expected to integrate more and more cores on a single chip, increasing the aggregate demand for main memory capacity. Satisfying such a demand with DRAM alone may prove difficult due to DRAM scaling challenges [9]. To address this problem, recent work has proposed using DRAM as a cache to large non-volatile memories, such as phase-change memory (PCM), which are projected to be much more scalable than DRAM at comparable access latencies [6]. A key challenge in scaling hybrid main memories is how to efficiently manage the *metadata* (such as tag, LRU, valid, and dirty bits) for data in such a large DRAM cache at a fine granularity.

Most prior hardware-based approaches toward large, fine-granularity DRAM caches have either (1) stored metadata for each cache block in a large SRAM structure, limiting scalability and increasing cost (e.g., [11]), or (2) stored metadata in a contiguous region of DRAM, requiring additional accesses and reducing performance ([1, 15]). **Our goal** is to achieve minimal performance degradation compared to large on-chip SRAM metadata structures, but with orders of magnitude lower hardware overhead.

2 Hybrid Main Memories

Fig. 1 shows the organization of a hybrid memory using DRAM as a cache to PCM. Both DRAM and PCM are composed of multiple banks organized as rows and columns of memory cells. Each bank is equipped with a row buffer that stores the contents of the most recently accessed row of data. Accesses to row buffer data (*row buffer hits*) can be serviced more quickly than accesses to the memory array (*row buffer conflicts*).

A hybrid memory controller located on-chip is responsible for managing the placement of data, scheduling accesses, and performing data movement between the DRAM and PCM devices.

Metadata Lookup. Tracking whether data are located in the DRAM cache requires some form of metadata storage. While tracking data at a large granularity (e.g., 4KB) is possible,

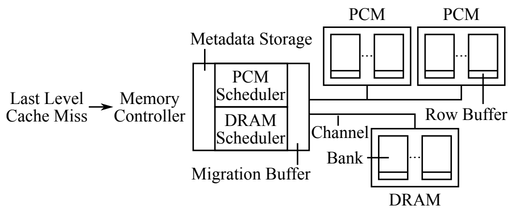


Fig. 1: A hybrid DRAM-PCM main memory organization.

doing so can cause underutilization of the DRAM cache and waste bandwidth, making large-granularity migrations inefficient and undesirable if most data in the large blocks are not accessed. On the other hand, tracking data at a fine granularity can incur large storage overheads (e.g., 8MB required to track all 128B blocks in a 512MB DRAM cache). Others have proposed storing metadata in the DRAM cache itself, alongside data [1, 8, 15]. While this mitigates the storage overhead and scalability limitations induced by on-chip metadata stores, it instead requires additional metadata accesses to DRAM, increasing main memory latency and bandwidth consumption.

Request Scheduling. Based on the retrieved metadata, the request is placed in either the DRAM or PCM scheduler. To maximize throughput, requests are scheduled using a *first ready, first-come first-served (FR-FCFS)* scheduling policy [12, 16].

Data Movement. After data arrive at the memory controller, if they should be cached into (or evicted from) the DRAM cache, a special migration request is inserted into the destination scheduler, which writes them into the destination device. During this brief transient state, in-flight data are placed in a migration buffer in the memory controller, where they can be accessed at a latency similar to an on-chip cache. For each demand request to PCM, the memory controller may issue multiple data requests to support large migration granularities.

We would like to achieve the benefits of storing metadata in DRAM while minimizing the latency of metadata accesses. To this end, we propose a technique for storing metadata in DRAM coupled with a new architecture for reducing DRAM metadata accesses using a small, on-chip metadata buffer.

3 A Fine-Grained DRAM Cache Architecture

Overview. Independent to [8], we also observe that metadata can be stored in DRAM in the same row as their data, reducing

• *Manuscript submitted: 20-Jan-2012. Manuscript accepted: 16-Feb-2012. Final manuscript received: 23-Feb-2012.*

the access latency from two row buffer conflicts (one for the metadata and another for the datum itself), to one row buffer conflict and one row buffer hit (if the datum is located in DRAM). Based on this observation, to further mitigate metadata lookup latency, we cache the metadata for the most recently accessed rows in DRAM in a small, on-chip buffer. The key idea is that the metadata needed for data with temporal or spatial locality will likely be cached on-chip, where they can be accessed at the same latency as an SRAM tag store.

3.1 Storing Metadata Alongside Data

Fig. 2 shows how data and metadata are laid out in the same row: Each row uses one cache block (referred to as the *metadata block*) to store the metadata for the remaining cache blocks. On a memory access, the row index of the request is used to retrieve the metadata block. The cache block address of the request is used to index this metadata block to retrieve its metadata. We call this metadata organization *tags-in-memory (TIM)*. Fig. 3 compares TIM with storing metadata in a DRAM region separate from their data (Region), where TIM reduces memory access latency by accessing data after its metadata lookup, at a lower latency by hitting in the row buffer.

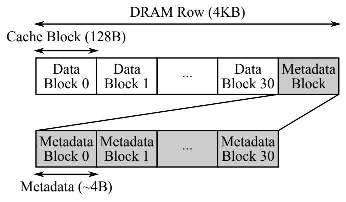


Fig. 2: A method for efficiently storing tags in memory (TIM). Metadata are stored in the same row as their data.

Fig. 4(a) shows the performance of various DRAM cache management techniques which we will refer to throughout the paper (our simulation methodology is explained in Section 4). Comparing a system with all DRAM cache metadata stored in an on-chip SRAM (SRAM), a simple metadata storage scheme where metadata are stored contiguously in a separate DRAM region (Region), and the TIM optimization just discussed (TIM), two observations are worth noting. (1) The simple, region-based in-memory tag storage mechanism increases average memory access latency by 29% and requires additional metadata accesses, degrading performance by 48% compared to the SRAM tag store. (2) While the TIM approach improves average memory access latency by 19% over Region, there is still a significant 30% performance gap in between TIM and the SRAM tag store design due to the additional off-chip tag lookups.

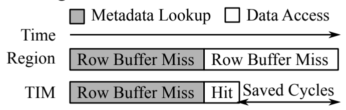
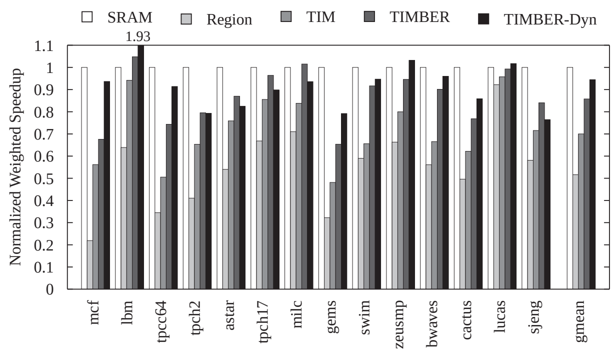


Fig. 3: Unlike storing metadata in a region, storing them in the same row as their data allows subsequent accesses to hit in the row buffer, reducing memory latency.

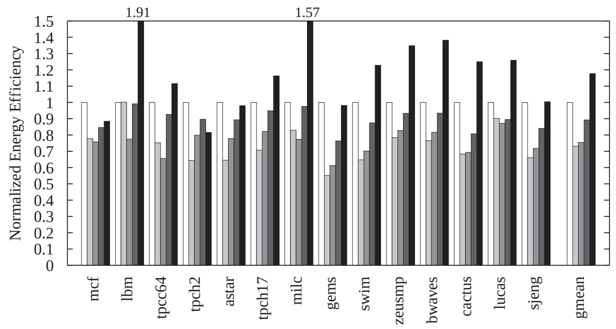
3.2 A Buffer for Recently-Accessed Metadata Blocks

To help mitigate the performance deficiency of TIM without requiring the large hardware overheads of an SRAM metadata store, we propose caching metadata for a small number of recently accessed rows in a buffer, called TIMBER. The key insight behind TIMBER is that caching *metadata* (i.e., tags) for data with good locality in a small buffer allows most metadata accesses to be serviced at SRAM latency, and at a low storage overhead. TIMBER is organized as a cache, where entries are tagged by DRAM row indices and the data payload contains the metadata block for a particular row, as shown in Fig. 5.

Fig. 5 also shows how metadata are looked up under TIMBER: An incoming memory request's address is used to determine its row index to access TIMBER. If the row index *does not match* the TIMBER tag entry, the row's metadata must be fetched from DRAM and inserted into TIMBER. When the row index *matches* the TIMBER tag entry, the tag data in the TIMBER entry's payload is further compared with the request block's address tag. The cache block is located in DRAM if the tags match, and located in PCM otherwise.



(a)



(b)

Fig. 4: Multi-core (a) performance and (b) energy efficiency.

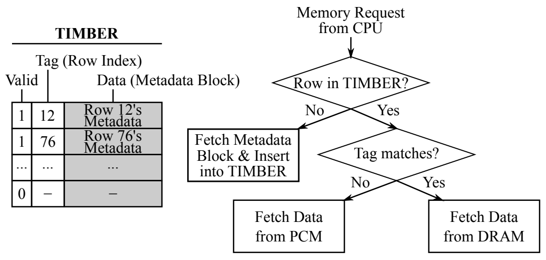


Fig. 5: TIMBER organization and metadata lookup.

3.3 Implementation and Hardware Cost

For a system with a 44-bit physical address, each TIMBER entry requires 4B for TIMBER tag storage, 128B for metadata storage, and 1 bit for validity for a total overhead of around 8KB for 64 entries. For comparison, the amount of SRAM overhead required to store all metadata for a 512MB DRAM cache is 8MB (i.e., roughly 1000× larger).

4 METHODOLOGY

We developed a cycle-level memory simulator as part of an in-house x86 multi-core simulator, whose front-end is based on Pin, and executes representative phases of benchmarks as profiled by PinPoints. Table 1 shows the major system parameters used in our study. We warmed up the system for one billion cycles and collected results for one billion cycles.

For our 8-core workloads, we run one single-threaded instance of a SPEC CPU or TPC-C/H benchmark[1] on each core

1. Benchmarks were chosen based on their large memory footprints.

TABLE 1: Baseline simulation parameters adapted from [6]. ‡Note that TIM can also support set-associative DRAM caches.

Processor	8 cores, 4GHz, 3-wide issue (maximum 1 memory operation per cycle), 128-entry instruction window
L1 cache	Private 32KB per core, 4-way, 128B blocks
L2 cache	Shared 512KB per core, 8-way, 128B blocks, 16 miss buffers
Memory controller	128-/128-entry read/write request queues and 128-entry migration buffer per controller; FR-FCFS scheduler [12, 16]
TIMBER	64-entry, direct-mapped
Memory	2 controllers (DRAM and PCM), each with a 64-bit channel and 1 rank with 8 banks. 512MB direct-mapped‡, write-back, no-write-allocate DRAM cache. Block-level write-back of dirty data to PCM. Open-row policy.
Timing	DRAM: row buffer hit (conflict) = 40 (80) ns. PCM: row buffer hit (clean conflict/dirty conflict) = 40 (128/368) ns.
Energy	Both: row buffer read (write) = 0.93 (1.02) pJ/bit. DRAM: array read (write): 1.17 (0.39) pJ/bit. PCM: array read (write) = 2.47 (16.82) pJ/bit.

TABLE 2: Consolidated multi-core workload characteristics.

Workload	MPKI	Footprint (MB)	Avg. Dynamic Migr. (B)
mcf	65.18	2481.0	128
lbm	64.68	1703.1	700
astar	21.55	2171.6	128
tpcc64	19.44	1150.6	169
tpch2	17.97	1501.5	128
tpch17	11.72	1696.4	178
milc	11.04	1273.6	128
gems	8.84	1819.9	160
swim	8.29	1072.3	376
zeusmp	8.68	1176.0	361
bwaves	7.42	1506.0	128
cactus	4.02	783.7	881
lucas	2.15	1002.2	417
sjeng	0.85	893.9	128

for a total of eight benchmark instances per workload, representative of many consolidated workloads for large CMPs. Table 2 characterizes our workloads on an 8-core all-SRAM metadata store system in terms of last-level cache misses per kilo instruction (MPKI), amount of referenced data (footprint), and under TIMBER for a metric we will discuss in Section 5.5.

5 Evaluation

5.1 Performance Evaluation

Fig. 4(a) shows the performance of various DRAM cache management techniques using the weighted speedup metric [2]: the sum of the speedups of the benchmarks when run together compared to when run alone on the same system with SRAM metadata storage.

The addition of a 64-entry TIMBER (we study the sensitivity of our mechanism to TIMBER size in Section 5.3) improves performance over TIM by more than 22%. This performance boost is due to TIMBER's ability to issue accesses to the same row just as quickly as an SRAM tag store if the row's metadata are cached in TIMBER. We find that 62% of accesses hit in TIMBER (34% with data located in DRAM and 28% in PCM), and 38% of accesses miss in TIMBER and need to first access metadata from DRAM.

5.2 Energy-Efficiency Evaluation

Fig. 4(b) compares the dynamic main memory energy efficiency (performance per watt) of the different techniques (higher is better). Note that we do not consider memory controller energy, thus we do not penalize techniques with large SRAM storage. While Region and TIM are around 25% less energy efficient than an all-SRAM metadata store system due to their increased number of DRAM lookups, TIMBER is able to service many such lookups from its small on-chip cache, achieving energy efficiency within 11% of an all-SRAM system.

5.3 Sensitivity to Tag Buffer Size

Fig. 6(a) shows the sensitivity of performance and TIMBER miss ratio to the size of TIMBER. As TIMBER size increases, performance improves and TIMBER miss ratio decreases, though with diminishing marginal returns, as the metadata cached in TIMBER cover a large portion of the DRAM cache.

5.4 Sensitivity to Number of Cores

Fig. 6(b) shows how our TIMBER technique scales with different numbers of cores. For this study, we scale the number of cores, keeping the amount of DRAM proportional to the cores, and plot the number of TIMBER entries needed to achieve around 6% of the performance of an all-SRAM metadata system. TIMBER size needs to scale proportionally with core count, but the absolute storage overhead of TIMBER remains three orders of magnitude smaller than an SRAM tag store.

5.5 Potential Benefits Enabled by Fine-Granularity

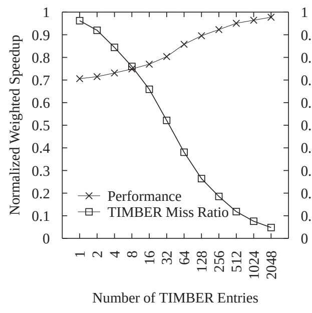
Managing a DRAM cache at a fine granularity provides the opportunity for migrating different amounts of data from PCM to DRAM based on runtime characteristics. For example, applications which have high data reuse may benefit from caching more data, increasing DRAM cache hit rate. On the baseline SRAM system, we found that simply caching 4KB of data per migration could improve DRAM cache hit rate by 20%, but cause performance to degrade by 75% due to the increase in bandwidth consumption on the DRAM and PCM channels by 55% and 140%, respectively. To balance this tradeoff between locality and bandwidth consumption, we developed a simple policy which dynamically adjusts the migration granularity.

Our mechanism is inspired by that of Qureshi et al. [10], where certain "leader" cache sets follow fixed replacement policies and the remaining "follower" cache sets follow the replacement policy that leads to the lowest cache miss rate among the leader sets. We divide main memory into 256 *sets of rows* consisting of seven leader row sets which employ a fixed granularity for migration—128B, 256B, 512B, 1KB, 2KB, 4KB, and no migration—and 249 follower row sets. At the end of each ten-million–cycle quantum, we compute per thread, per leader set: (1) average memory access latency (also counting migration buffer accesses), using latency counters per outstanding request, whose sum is divided by request count, and (2) number of cache blocks migrated.

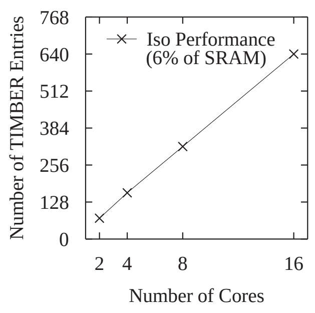
We determine a thread's migration granularity for the next quantum as the migration granularity of the leader set with the *smallest product of average request access latency and number of cache blocks migrated*[2]. The key idea is that follower sets should mimic leader sets with low access latencies and few migrations to improve system performance. Table 2 shows the average granularity measured for workloads under our policy.

Fig. 4(a) shows the performance of a 64-entry TIMBER with our dynamic migration granularity technique (TIMBER-Dyn). Several observations are in order. First, we find that TIMBER with our dynamic migration granularity technique can achieve within 6% of the performance of the SRAM metadata storage system due to its ability to adjust its migration granularity to workload characteristics, which in some cases involves not migrating data at all during a quantum. This improves both

2. For the "no migration" set, we only use average request latency, and our policy only considers sets with at least one access.



(a) Number of TIMBER entries.



(b) Core count, scaling DRAM.

Fig. 6: Performance sensitivity to system characteristics.

DRAM cache efficiency and TIMBER efficiency. We find that 45% of TIMBER accesses result in hits for data in the DRAM cache, 12% of TIMBER accesses result in hits for data in the PCM backing store, and 43% of accesses miss in TIMBER and must access metadata from DRAM. Second, as Fig. 4(b) shows, being more prudent about migrations improves the DRAM cache utilization and reduces bandwidth contention, leading to energy efficiency benefits of 18% over the SRAM-tag baseline.

Although we use TIMBER as a substrate to efficiently implement our dynamic migration granularity technique, other metadata storage techniques could be used in conjunction with our dynamic migration granularity mechanism[3]. Also note that this is an early design of a dynamic granularity mechanism and we are working on ways to improve its efficiency.

6 Related Work

Prior approaches toward reducing metadata storage overhead have managed the DRAM cache using large cache lines on the order of kilobytes [1, 4] or represented the presence of smaller sectors of a large cache block as a bit vector, eliminating the need to store full tag metadata [7, 13]. These techniques still store metadata for all of DRAM, increasing bandwidth consumption and pollution of the DRAM cache, increasing false-sharing probability, and limiting scalability. CAT reduced tag storage overhead by observing that cache blocks with spatial locality share the same high order tag bits and thus one tag can represent multiple cache blocks [14]; their cache for partial tags requires an associative search and cache elements must be invalidated on partial tag eviction. While TIMBER also caters to accesses with spatial locality (at the row granularity), we take a fundamentally different approach by storing tags in DRAM and caching full tag information in a small buffer.

Concurrent to this work, Loh and Hill also proposed storing metadata in the same row as their data and exploit this technique with an on-chip cache that stores a bitmap representing the presence in DRAM of a fixed number of recently accessed regions. When a region is evicted from the metadata cache, its data must also be evicted from DRAM, so a large metadata cache is needed to reduce such data evictions (the authors use 2MB of SRAM for a 512MB DRAM cache) [8]. The key difference of our work is that we cache *full metadata* for a *small subset* of rows in DRAM and *retain both the metadata and data for entries evicted from TIMBER, in the DRAM cache*. This improves DRAM cache utilization, avoids metadata lookups for TIMBER hits, and reduces costly writeback traffic to PCM.

Prior techniques to determine the fetch size in traditional caches (e.g., [3, 5]) differ from our dynamic policy in two key ways. (1) They consider the *benefit* side of different caching granularities, whereas we also consider the *cost* side in terms of data movement contention, a factor that significantly affects system performance in a hybrid memory. And (2) they track reuse information using large structures whose size must scale with the cache size, whereas our technique only requires structures whose size scale with the number of outstanding memory requests which is much smaller than the size of the DRAM cache.

3. For example, on an all SRAM metadata system, our technique improves performance by 3% and improves energy efficiency by 43%.

7 Conclusions

We introduced an efficient architecture for managing a large DRAM cache. Leveraging the observation that metadata can be stored in the same row as their data, we designed a new architecture which caches recently-used metadata to provide the benefits of a large SRAM metadata store for accesses with temporal and spatial locality. Building upon our technique, we also explored the design of a new caching policy for hybrid memories which determines the migration granularity that leads to low access latency and few migrations. Our results show that our technique and caching policy can achieve similar benefits to a large SRAM metadata store at much lower storage overhead (8KB compared to 8MB) and 18% better energy-efficiency due to fewer migrations, even when the energy overhead due to large SRAM metadata storage is not considered.

Acknowledgements

We thank the members of the SAFARI research group and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of an NSF CAREER Award CCF-0953246, NSF EAGER Grant CCF-1147397, and the Gigascale Systems Research Center. Part of this work was done while Justin Meza and HanBin Yoon were interns at Hewlett-Packard Labs.

References

[1] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. SC '10.
[2] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 2008.
[3] K. Inoue, K. Kai, and K. Murakami. Dynamically variable line-size cache exploiting high on-chip memory bandwidth of merged DRAM/logic LSIs. HPCA '99.
[4] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive filter-based DRAM caching for CMP server platforms. HPCA '10.
[5] T. L. Johnson and W.-m. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. ISCA '97.
[6] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. ISCA '09.
[7] J. Liptay. Structural aspects of the System/360 Model 85, II: The cache. *IBM Syst. J.*, 1968.
[8] G. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. MICRO '11.
[9] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM J. Res. Dev.*, 2002.
[10] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. ISCA '06.
[11] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. ISCA '09.
[12] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. ISCA '00.
[13] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. ISCA '94.
[14] H. Wang, T. Sun, and Q. Yang. CAT - caching address tags - a technique for reducing area cost of on-chip caches. ISCA '95.
[15] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM cache architectures for CMP server platforms. ICCD '07.
[16] W. K. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. U.S. patent 5630096, '97.