2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth

Nagendra Gulur, Mahesh Mehendale
Texas Instruments (India),
Bangalore, India
nagendra@ti.com,m-mehendale@ti.com

R. Manikantan, R. Govindarajan
Indian Institute of Science,
Bangalore, India
r.manikantan@gmail.com, govind@serc.iisc.ernet.in

***Abstract*—In this paper, we present *Bi-Modal Cache* - a flexible stacked DRAM cache organization which simultaneously achieves several objectives: (i) improved cache hit ratio, (ii) moving the tag storage overhead to DRAM, (iii) lower cache hit latency than tags-in-SRAM, and (iv) reduction in off-chip bandwidth wastage. The *Bi-Modal Cache* addresses the miss rate versus off-chip bandwidth dilemma by organizing the data in a *bi-modal* fashion - blocks with high spatial locality are organized as *large* blocks and those with little spatial locality as *small* blocks. By adaptively selecting the right granularity of storage for individual blocks at run-time, the proposed DRAM cache organization is able to make judicious use of the available DRAM cache capacity as well as reduce the off-chip memory bandwidth consumption. The *Bi-Modal Cache* improves cache hit latency despite moving the metadata to DRAM by means of a small SRAM based *Way Locator*. Further by leveraging the tremendous internal bandwidth and capacity that stacked DRAM organizations provide, the *Bi-Modal Cache* enables efficient concurrent accesses to tags and data to reduce hit time. Through detailed simulations, we demonstrate that the *Bi-Modal Cache* achieves overall performance improvement (in terms of Average Normalized Turnaround Time (ANTT)) of** 10.8%**,** 13.8% **and** 14.0% **in 4-core, 8-core and 16-core workloads respectively.**

I. Introduction

With increasing core counts in single-chip multiprocessors, off-chip memory has become a performance-limiting factor from both latency and bandwidth perspectives. Due to limited growth in pin counts, data access rates from off-chip DRAM systems have not scaled to match the demands of modern servers leading to the bandwidth wall problem [1]. 3D die stacking [2] has emerged as a promising alternative wherein DRAM memory dies are stacked on top of a processor die using high bandwidth through-silicon-vias (TSVs). Stacking offers 100s of MBs to even gigabytes of DRAM capacity at very high bandwidth alleviating the off-chip *memory wall* constraint.

Researchers have proposed to use this capacity as a very large capacity last level cache. The proposed solutions, based on the size of the DRAM cache block, fall under two categories: *fine-grained [3], [4]* - in which the cache is organized at the same block size as the last level SRAM cache[1](typically 64 or 128 bytes), and *coarse-grained [5], [6]* - in which the DRAM cache blocks have much larger sizes (typically 2048 or 4096 bytes, not exceeding DRAM page size). Fine-grained organizations incur prohibitively high metadata storage overhead (of the order of many megabytes) and thus forces the metadata to be stored in the stacked DRAM itself. This increases the hit latency as the accesses to tag and then data happen serially, incurring multiple (at least two) DRAM accesses. The small block size also fails to exploit the abundant spatial locality inherent at this level, incurring higher cache miss rates. On the other-hand, the fine-grained organization uses off-chip bandwidth and cache capacity efficiently.

The coarse-grained organizations are characterized by higher cache hit rates and lower metadata storage needs. Metadata can be stored on SRAM thereby enabling faster access times. Anticipating that stacked DRAM capacities will grow, we argue that SRAM-based metadata will become unaffordable, even for large block size. For example, a DRAM cache of 1GB size organized as 1024 byte blocks needs metadata storage of 4 MB, assuming a per-block metadata overhead of 4 bytes. Further, the large block size incurs wasted bandwidth by fetching un-used data into the cache. This also causes under-utilization of cache space.

Going forward, the DRAM caches in scalable multi-core architectures of the future would need to achieve higher hit rates, have lower hit (and miss) latency, reduce off-chip memory bandwidth wastage, and improve cache space utilization. Meeting these objectives together is quite challenging, and [5] proposes a scheme to have them all. But can we have more?

Towards this goal, in this paper we propose the *Bi-Modal Cache*. *Bi-Modal Cache*, as the name suggests organizes the data with high spatial locality as large blocks and the rest as small blocks. *Bi-Modal Cache* reduces wasted bandwidth and improves cache space utilization by (i) accurately identifying the spatial locality at the level of cache blocks and storing them appropriately and (ii) learning the spatial locality at the application level and identifying a judicious mix of large and small blocks that matches the application requirement.

Second, *Bi-Modal Cache* stores metadata on DRAM. We overcome the DRAM tag access latency issue and improve the average hit latency (compared to both tags-in-SRAM and tags-in-DRAM) by two optimizations: (i) we introduce

[1] Abbreviated *LLSC* throughout the rest of this paper.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.36

a small SRAM based *way locator* which helps avoid making DRAM accesses for tags in a large majority of accesses, and (ii) we perform parallel tag and data accesses on DRAM. The resulting *Bi-Modal Cache* achieves high hit rate, low hit latency, high cache-space utilization, and reduced (off-chip) memory bandwidth.

Through extensive evaluation of the cache organization on 4, 8 and 16-core workloads, we demonstrate that our organization achieves lower average *LLSC* miss penalty than other proposed organizations. *Bi-Modal Cache* achieves an improvement in ANTT of 10.8%, 13.8% and 14.0% in 4-core, 8-core and 16-core workloads respectively. These improvements are over an aggressive baseline, *AlloyCache* [4]. Further, compared to the recent *Footprint Cache* [5], *Bi-Modal Cache* achieves an average reduction in latency of 12% and an ANTT improvement of 4.9%.

II. The Design Space of DRAM caches

A. Overview of DRAM caches

By virtue of stacking and the inherent density of DRAM, a DRAM cache provides a large capacity (typically 64MB to even gigabytes) offering an unprecedented opportunity to hold critical workload data on chip. The DRAM cache is typically organized as a last level shared cache behind a hierarchy of SRAM caches[2]. A DRAM cache offers large capacity caches at lower power unlike L1, L2 caches that are implemented using SRAM. However DRAM cache design requires careful attention to access latency since a typical DRAM access requires activating a row of cells, sensing this charge and finally transmitting the sensed data over a bus. Since row activation has drained the corresponding capacitors, a *precharge* operation is required to restore the charge back on these capacitors.

B. Design Space

Performance of DRAM cache is determined by the following parameters:

Metadata Storage: The large capacities offered by DRAM caches require higher metadata (tags, valid, dirty bits, recency bits etc) storage requirements which can run into multiple megabytes. For instance, assuming a tag overhead of 4 bytes per cache block, a 256MB cache organized at 64B block-size requires as much as 16MB of metadata to be stored. Obviously committing this much storage in SRAM is costly and energy expensive[3]. Several researchers have proposed to address this issue by techniques such as larger cache block sizes or by moving the metadata storage to the DRAM cache itself [3], [4].

Cache Block Size: Larger block sizes (e.g., 2KB or 4KB) reduce metadata overhead and also exploit spatial locality very well. However, they suffer from two issues: (i) cache misses consume large off-chip bandwidth to fetch large blocks, and (ii) in the absence of sufficient spatial locality, they waste cache capacity.

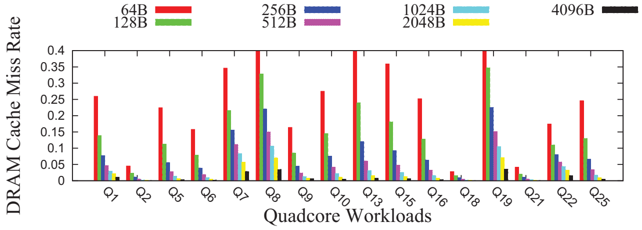


Figure 1: Miss Rates Fall with Increasing Block Sizes

Set Associativity: Associativity generally improves cache hit rates by reducing conflict misses, thereby improving the likelihood of retaining useful blocks longer. In the context of DRAM caches, we find that associativity itself does not have a significant bearing on hit rate – an observation that other researchers have also made [7], [4]. However, associativity is important in the sense of providing the substrate for implementing non trivial block allocation and replacement policies.

These parameters interact to influence cache hit rate, row-buffer hit rate, hit latency, and off-chip bandwidth. We discuss these interactions below:

Achieving High Cache Hit Rates: In Figure 1, we plot the miss rates observed in several quad-core workloads[4] at 7 different block sizes: 64B, 128B, 256B, 512B, 1024B, 2048B and 4096B. For most workloads, the miss rate nearly halves with doubling of block size indicating that DRAM caches should be organized with large block sizes to leverage spatial locality.

Reducing Cache Hit Time: Hit time is governed by tag lookup as well as data access times. In the works of [4], [3], metadata is placed along with data on DRAM rows. In [3], the cache is organized as 29-way set-associative. Through *Compound Access Scheduling*, both the tags and data are accessed via multiple column accesses once the row has been activated. In [4], a direct mapped organization with co-located tag and data is proposed to reduce hit latency but requires a larger burst to retrieve both in a single access. Further, it strictly limits the organization to 64B block size and direct-mapping. In [5], the metadata is placed on SRAM. Thus, tag lookup is performed first and then a single DRAM access is made to retrieve data (if there was a tag match). Even here, tag and data accesses are sequential, incurring high latency. Further, a large tag RAM store incurs latency of several cycles even on SRAM making the serialized accesses undesirable. In a recent paper [8][5], the tags are held on DRAM in a configuration similar to that

[2] In this work, the DRAM cache is located behind a cache-coherent shared cache and thus does not handle coherence related events.

[3] The tag storage overhead may even exceed the total size of the last-level SRAM cache.

[4] Details of workloads and experimental setup are presented in Section IV.

[5] The work in [8] is independent and concurrent to ours.

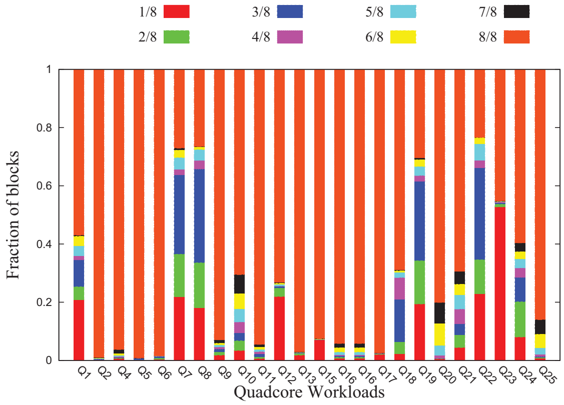


Figure 2: Distribution of Blocks with Different Utilizations

in [3] and a small SRAM tag cache is introduced to cache all tags of recently accessed sets. If the tag cache achieves high hit rate, and is small in size, then the hit time can be reduced.

Improving Cache Space Utilization: Small blocks use cache space effectively but at the cost of high metadata storage. They also do not exploit spatial locality and incur frequent cache misses. Large blocks on the otherhand may "commit" space in the cache without fully using every byte allocated to it. This block-internal fragmentation results in inefficient utilization of capacity and causes more misses with resulting increase in off-chip traffic. Thus we argue that cache space utilization also needs to be improved to effectively reduce the off-chip bandwidth.

In order to understand the extent of internal fragmentation, we tracked the utilization of each 64B sub-block in 512B DRAM cache blocks and computed the fractions of blocks that had different amounts of utilization. Utilization varies from $\frac{1}{8}$ (corresponding to just 1 sub-block being referenced) all the way upto 1 (when all 8 sub-blocks were referenced by the CPU). Figure 2 plots the fractions of blocks of different utilizations. While some workloads (noticeably $Q2$, $Q4$ and $Q5$) have over 90% of blocks with 100% utilization, others (noticeably $Q7$, $Q8$, $Q19$ and $Q23$) have $< 30\%$ of such blocks. Thus always allocating large blocks results in wasted cache space. It also leads to wasted over-fetch from main memory.

This motivates our *bi-modal* organization that uses two block sizes to achieve the dual objectives of improving hit rate, and improving space utilization/bandwidth reduction.

Associativity has only a minor impact: Lastly, we examine the role played by set associativity on large last level caches. Much like prior work [7], [3], we find that hit rates do not significantly improve with higher associativity. Thus, given the cost of implementing high set associativity (multiple tag comparisons, and maintaining recency information), we argue that a reasonably modest associativity is good enough to capture any additional cache hits to be gained. Having some associativity proves to be beneficial in constructing adaptive cache management schemes - such as being able to control block replacement, block insertion and block sizing.

III. *Bi-Modal Cache* ORGANIZATION

A. *Overview*

Bi-Modal Cache stores metadata on DRAM anticipating the need to manage very large DRAM caches with multi-megabyte metadata storage requirements. The cache organization boosts performance by two independent optimizations. First, in contrast to fixed sized blocks, our organization is *bi-modal* - some big blocks and some small blocks. By predicting a block's spatial utilization, the cache is able to decide and fill the cache miss as a big way or as a small way. This not only reduces off-chip bandwidth demand but also reduces block-internal fragmentation. Second, *Bi-Modal Cache* improves the average access latency over both the tags-in-DRAM as well as the tags-in-SRAM organizations. It does so by looking up the metadata and data in parallel (much like conventional SRAM based L1 caches) by storing metadata on its own DRAM bank/channel. This design also helps improve row-buffer locality for metadata accesses. Further, by introducing an efficient and accurate (high hit-rate) *Way Locator* in SRAM for tag look-up, this organization ensures that only data accesses are performed on DRAM for the majority of accesses. The *Way Locator* caches the tags and data locations of the most recently accessed blocks for a subset of heavily accessed sets.

Access to *Bi-Modal Cache* starts with a tag look-up in the Way Locator. A hit in the Way Locator determines the location in DRAM where the data is available; a subsequent access fetches the data from DRAM. A miss in Way Locator leads to accessing the tag and data from DRAM. Here, *Bi-Modal Cache* enables efficient access of tags and the opening of the corresponding data page in DRAM concurrently.

Figure 3 compares *Bi-Modal Cache* with other related schemes in terms of latency breakdown (per access). The *AlloyCache* provides a low-latency baseline by fetching tag and data together in a single big burst (72 bytes). However this comes at the cost of a higher cache miss rate due to the small block size (see Figure 1). The tags-in-SRAM organization (Footprint-Cache) has a slightly higher latency than the *AlloyCache* scheme since large SRAM storage access is costlier than an additional data transfer. While the ATCache achieves low hit latency corresponding to tag-cache hits, it suffers from low tag-cache hit rates due to high associativity and 64B block size. Thus its average latency is high. In *Bi-Modal Cache* the way locator achieves a high hit rate ($>$ 90% in quad and 8-core) resulting in very few DRAM accesses for tags. On way locator misses, *Bi-Modal Cache* leverages the high RBH in the metadata bank to reduce tag+data access time by issuing parallel tag and data accesses. In Section V we show that in practice a

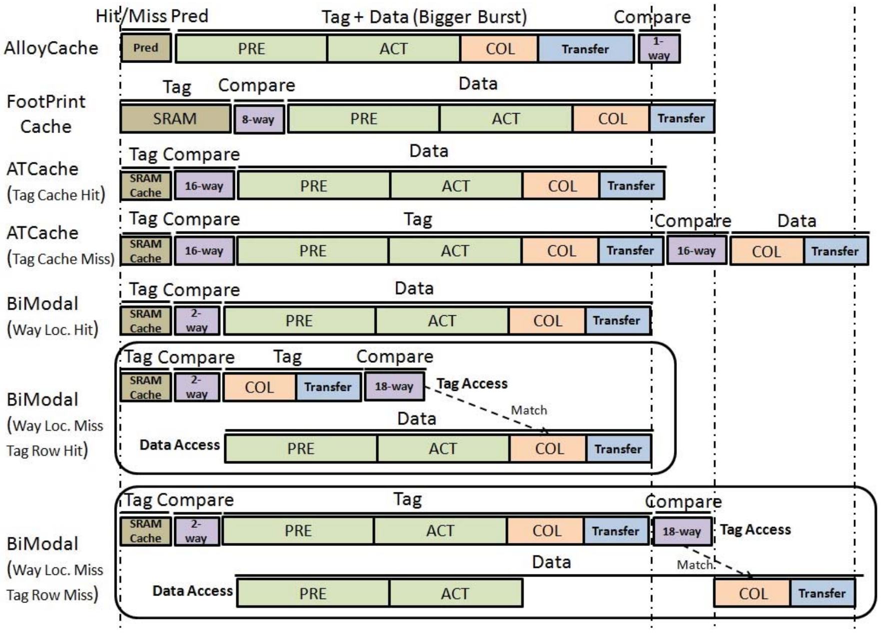


Figure 3: Latency Reduction with *Bi-Modal Cache*

combination of high DRAM cache hit rate, way locator hit rate, and high RBH in the metadata bank combine to achieve a lower *LLSC* miss latency than all the other schemes.

Taken together, these optimizations simultaneously achieve improvements in hit latency, cache hit rate, row-buffer hit rate, off-chip bandwidth and cache space utilization. This results in achieving the lowest average *LLSC* miss penalty.

Table I summarizes these features and compares *Bi-Modal Cache* to prior organizations.

B. Bi-Modal Blocks: Design for Bandwidth and Cache Space Utilization

1) Cache Organization: We propose to organize the data at two granularities - big blocks (512B) and small blocks (64B). Intuitively, big blocks are preferred for data that has high spatial utilization and small blocks for the rest. A static partitioning of the cache to hold a fixed number of big and small blocks would be sub-optimal and very rigid considering that different workloads have widely different spatial utilizations (see Figure 2) and these utilization levels are likely to vary over time even for a single workload. Thus we organize each cache set to be *bi-modal* - i.e., each cache set can hold X big and Y small blocks upto the size of the set. Thus the "state" of a set at any time is denoted as (X, Y).

A set in state (X, Y) has $(X + Y)$ ways and is thus $(X + Y)$-way associative. Given the limited benefit of high associativity for large DRAM caches [4], [7], we prefer to limit the maximum associativity permitted by restricting the number of small ways. For a set of size 2KB, the allowed set states are $\{(4, 0), (3, 8), (2, 16)\}$. The size of a set is chosen to map its contents to reside within one DRAM page to avoid having to access multiple DRAM pages[6]. Similarly, for a 4KB set, the allowed states are $\{(8, 0), (7, 8), (6, 16), (5, 24), (4, 32)\}$.

Set Data Layout: A set's data maps to a DRAM page. Data for the big ways are stored adjacent to each other followed by data for the small ways. Further, in order to correctly identify the column address of a way, we number the big ways left-to-right with way-0 in the leftmost columns; small ways are numbered right-to-left with way-0 in the rightmost columns of the page.

Cache Indexing and Metadata Layout: In our organization, the number of sets is fixed while set associativity can vary over time on a per set basis. Thus the indexing scheme that maps an address to set index is the same as in a conventional cache. Specifically, for a cache of size C, and set size S, there are $2^M = \frac{C}{S}$ sets. For a physical address space that uses A bits, the least 9 bits are used for offset (since the big block size is 512B), followed by next M set-index bits, and the remaining $(A - M - 9)$ bits being the tag bits. For small blocks, in addition to these tag bits, the 3 high-order offset bits are also stored to ensure correct match. In the metadata bank, for each set, we store its state (X, Y) followed by tag bits for the big blocks followed by the tag bits for the small blocks. There are upto 18 tags per set as the maximum number of blocks per set is 18 when the set in the $(2, 16)$ state. Figure 4 shows the data layouts in the

[6] Accesses to multiple DRAM pages within a bank introduces additional access latency.

Attribute	Loh-Hill	AlloyCache	ATCache	Footprint Cache	*Bi-Modal Cache*
Block Size	64B	64B	64B	2048B	Two sizes (512B, 64B)
Associativity	Yes (29 way)	Only direct-mapped	Yes (16-way)	Yes (Fixed)	Yes (4–18 ways)
Metadata	On DRAM	On DRAM	On DRAM	On SRAM	On DRAM
Metadata Overhead	High	High	High	Low	Low
SRAM storage (for caches/predictors)	High	Low	Low	Low	Low
DRAM cache Hit Latency	High - (Multiple DRAM Accesses for Tag then Data)	Low - (1 DRAM access with a larger burst for tag+data)	High - (Moderate Tag Cache Hit Rate, Large Associative Search)	Moderate - (Sequential Tag, then Data)	Low - (High Way Locator Hit Rate, 2-Way Search, Parallel Tag+Data on Miss)
DRAM cache Hit Rate	Low	Low	Low	High	High
Avg LLSC Miss Latency	High	Moderate	High	Low	Low
Wasted Off-Chip Bandwidth	No	No	No	Low	Low
Block Internal Fragmentation	No	No	No	High	Reduced

Table I: How *Bi-Modal Cache* compares to existing DRAM cache organizations

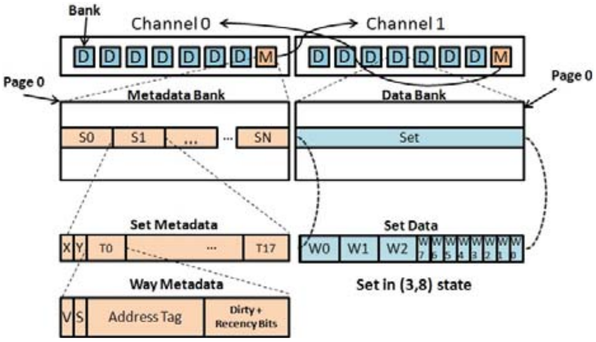


Figure 4: Data and Metadata Layouts in DRAM

metadata and data banks. The figure shows 2 channels, 8 banks per channel and with a bank in each channel holding metadata for the data in the other channel. Each page of the data banks holds a set of size 2KB. A sample set in the $(3, 8)$ state is shown.

2) Data and Metadata in Separate Banks: Unlike prior schemes wherein the metadata is interleaved with data on the same DRAM rows, we propose to store the metadata on a separate DRAM bank on-chip. Stacked DRAM organizations provide tremendous bandwidth as well as capacity and we leverage these to dedicate one of the banks to hold all the metadata per channel. A typical DRAM stack has multiple (4–8) data channels that can access many banks (8–16 per channel), and thus it permits two concurrent accesses on two different channels. By mapping the metadata for data banks belonging to one channel onto a bank of another channel, concurrent access of metadata and data can be achieved. We issue a tag access operation on the metadata bank in parallel to activating the row in the bank that holds the corresponding set data[7]. This avoids the "tags-then-data" serializtion. This organization requires no hardware modification and can be implemented over existing stacked DRAMs.

This organization has an important advantage. It significantly improves the row-buffer hit rate of the metadata bank(s). By keeping only metadata in the DRAM pages of a bank, the "density" of metadata per DRAM page goes up increasing the likelihood of finding more row-buffer hits. To quantify this, consider an organization of the cache with 64B block size, DRAM page size of 2KB and 4B metadata per cache block. If the metadata is stored alongside data in the same pages, then 29 blocks (and their metadata) could be stored per page (as in [4]). A channel typically has 8–16 banks, and thus the interleaved scheme will have only 232–464 metadata entries in open row-buffers per channel[8]. On the other hand, if the metadata was stored separately, we can store 512 metadata entries per page increasing the likelihood of getting more row-buffer hits. As shown in Section V-E this scheme achieves higher RBH for metadata resulting in hit latency reduction.

3) Block Size Predictor: The decision on whether to fetch big or small blocks is facilitated by the *block size predictor*. The block size predictor comprises of two components: a *tracker*, to measure the actual spatial utilization levels seen, and a *predictor* which uses the information supplied by the tracker to make predictions for future cache misses.

Tracking Spatial Utilization: Spatial utilization is measured by tracking the utilization of 64-byte sub-blocks allocated in sets that the block size predictor samples. In particular, it allocates a utilization bit vector (8 bits for a 512 byte block, one for each 64B sub-block) for each sampled way and sets a bit to true whenever the corresponding sub-block is accessed by the CPU. When a way gets evicted, its utilization bit vector is used to update the *Block Size Predictor*. The utilization bit vector is then cleared to obtain utilization data for the incoming block. To reduce the storage

[7] On the data bank, we only open the row in anticipation of a DRAM cache hit, we do not make a data access until tags are checked. This is unlike traditional SRAM based parallel tag/data accesses.

[8] With larger sized blocks, the number of metadata entries per DRAM page in the interleaved organization falls further.

overhead of tracking, we use the idea of set-sampling [9]. The tracker monitors the utilization of all the big blocks in these sampled sets. We monitored about 4% of the sets resulting in a storage overhead of $\approx 20KB$ for a 256MB Cache.

Block Size Predictor: The size predictor uses the utilization bit vector to decide if the sampled way is to be classified as a big or small block. It does so by comparing the number of bits set to true against a configurable threshold level, T. If the number of set bits is $\geq T$, then the way is classified as big, else classified small. A high value of T requires higher utilization levels for blocks to be classified big. In our setup, we set T to 5 (maximum is 8 since there are eight 64B blocks in a 512B block)[9].

The predictor is implemented as a table in SRAM comprising 2^P entries indexed by P bits from the N tag and set index bits. Each entry contains a 2-bit saturating counter. If successive updates to an entry are in the same direction, then the counter is decremented to saturate at "00" (predict *small*) or incremented to saturate at "11" (predict *big*). The storage requirements are quite modest: a predictor with $P = 16$ needs only $2 \times 2^{16} = 128K$ bits (i.e., $16KB$).

4) Adapting Associativity in Each Set: Next we describe how the number of big and small blocks in each set is adapted.

Adapting Cache-wide State: The DRAM cache controller maintains a cache-wide global state (X_{glob}, Y_{glob}) denoting the number of big and small blocks to maintain on a per-set basis. (X_{glob}, Y_{glob}) is initialized to $(4, 0)$ and is periodically updated using a pair of counters - D_{big}, D_{small} - which keep track of the demand for big and small blocks respectively. We update the global state after each interval comprising of $1M$ DRAM cache accesses. Demand is measured as the number of DRAM cache misses suffered for each type of block size and is updated at corresponding miss events. We let $R = W \times \frac{D_{small}}{D_{big}}$ where W denotes a weight. R is compared to the current ratio of small-versus-big ways $\frac{Y_{glob}}{X_{glob}}$ to adapt the global state. The weight W helps control the preference for big/small blocks. Setting $W < 1$ boosts the preference for bigger blocks. We found that in practice, setting $W = 0.75$ provided a good tradeoff. The controller updates its global state using the rules below:

- *If* $R > \frac{Y_{glob}}{X_{glob}}$*:* then increase the quota for small blocks, i.e., set $X_{glob} = X_{glob} - 1$;$Y_{glob} = Y_{glob} + 8$
- *If* $R < \frac{(Y_{glob}-8)}{(X_{glob}+1)}$*:* then increase the quota for big blocks, i.e., set $X_{glob} = X_{glob} + 1$;$Y_{glob} = Y_{glob} - 8$
- Otherwise the state remains as before at (X_{glob}, Y_{glob}).

The storage overhead in implementing this control is negligible: two counters to track demand, two counters to maintain current prediction of cache state, and a register to store the weight.

Comparison Outcome	**Predicted: Big Block**	**Predicted: Small Block**
$X_s = X_{glob}$, $Y_s = Y_{glob}$	Replace a big block	Replace a small block
$X_s < X_{glob}$, $Y_s > Y_{glob}$	Evict 8 small blocks and insert big block.	Replace a small block
$X_s > X_{glob}$, $Y_s < Y_{glob}$	Replace a big block	Evict a big block and insert small block.

Table II: Block Replacement in *Bi-Modal Cache*

Adapting Per-Set State: The controller also initializes all cache blocks as big blocks - i.e., state (X_s, Y_s) of each set S is $(4, 0)$. At the time of a cache miss, the global state (X_{glob}, Y_{glob}) is compared to the set state (X_s, Y_s). Based on the outcome of the state comparison, appropriate allocation and replacement decisions are taken as shown in Table II. These steps essentially try to align the state of the set to the global state. We note that if the set state has to change, then the evicted/allocated block(s) must be the highest numbered way(s). Since the off-chip miss is the long pole, implementing the above replacement scheme will not become latency critical.

5) Handling Writebacks: Writebacks from the DRAM cache to the main memory are handled at 64B granularity by maintaining dirty bits for every 64B block inside the 512B block. Thus when a big dirty block is evicted, the writebacks are performed only for the 64B sub-blocks that are dirty. Note however that the entire big block is removed from the cache.

C. Way Locator - Design for Hit Latency Reduction

We introduce a *Way Locator* in SRAM which caches the way IDs of the most recent accesses to DRAM cache sets. The key observation that guides the design of our way locator is that in a large low-level cache, most cache hits are to the most recently used ways. This observation is supported by Figure 5 which shows the fraction of cache hits at various MRU (Most Recently Used) positions in an 8-way associative cache for several eight-core workloads . On average, more than 94% of hits are to the top 2 MRU ways. Similar observations were made even on 16-core workloads sharing the DRAM cache. Thus, it suffices to record the 2 most recently accessed ways for each set.

1) Way Locator Design: The way locator is a small 2-way set associative cache. It is a table indexed using K (out of N) bits drawn from the tag and set index bits of the incoming address. We chose not to use the PC of the instruction causing the access since it requires the PC value (and associated core-id) to be passed through 2-3 levels of memory hierarchy to the DRAM cache controller. As shown in Figure 6, for each index there are 2 entries, with each entry consisting of a valid bit, a block size bit (to denote big/small), remaining set+tag bits as well as the 3 leading bits of the offset, and a way identification number. For every access, the way locator is looked up using the

[9] T could be adjusted at run-time but that is beyond the scope of this work.

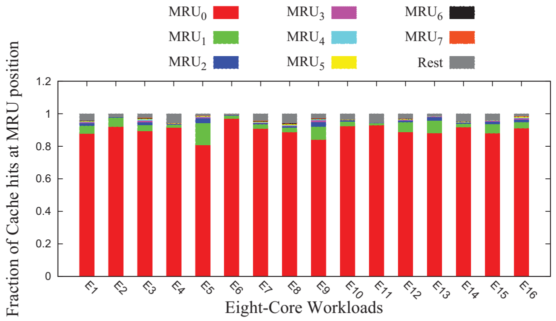


Figure 5: Most Cache Hits are to the Top-2 MRU Ways

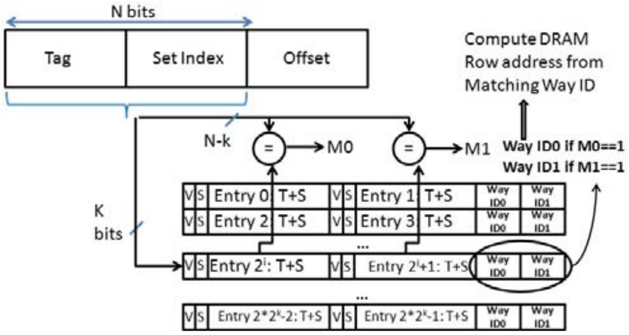


Figure 6: Design of the Way Locator

K-bit index and the 2 entries at that index are compared against the incoming address. If a match is found, then the corresponding way identification number is used to compute the column location of the data on the DRAM row and a DRAM access is initiated. Note that our way locator design ensures that it *never makes any wrong predictions*. By comparing all of the required address bits against stored entries, it ensures that there are no mis-predictions and hence no wasted DRAM accesses.

The way locator is updated whenever it misses. The way id of the accessed block is inserted to the way locator. In case of a cache block eviction, its way information is evicted from the way locator.

2) Way Locator Storage Requirements: Way Locator latency is governed by the size of the SRAM storage needed. In Table III we list the storage needs and latencies at various table sizes and DRAM cache sizes. The latency values are obtained using CACTII [10] at 22nm. The latencies of way locator lookup are smaller than those associated with looking up large SRAM tag stores [5] (6 cycles for 1MB, 7 for 2MB and 9 cycles for 4MB in CACTII 22nm).

Thus, while the techniques employed in the *Bi-Modal Cache* are well-known, they are orchestrated in an effective manner to design a flexible DRAM cache organization that achieves significant performance and that can continue to benefit with scaling DRAM cache size.

Num. entries (2×2^K)	$128M$ Cache, $4GB$ mem	$256M$ Cache, $8GB$ mem	$512M$ Cache, $16GB$ mem
K=10, 2K entries	5.9KB 1 cycle	6.14KB 1 cycle	6.4KB 1 cycle
K=12, 8K entries	21.5KB 1 cycle	22.5KB 1 cycle	23.5KB 1 cycle
K=14, 32K entries	77.8KB 1 cycle	81.9KB 1 cycle	86KB 1 cycles
K=16, 128K entries	278.5KB 2 cycles	294.9KB 2 cycles	311.3KB 2 cycles

Table III: Way Locator Storage and Latency

D. *Accessing the* Bi-Modal Cache

Putting things together, in this section, we describe how an access takes place in the *Bi-Modal Cache*. There are three distinct cases, depending on whether the access is a hit in way-locator, miss in way-locator but a DRAM cache hit, or miss in the DRAM cache.

1) Way Locator Hit: If the way locator indicates a match, we just access the corresponding data bank's way on DRAM.

Eliminating Metadata Accesses: The way locator enables an important optimization - a way locator hit can altogether eliminate DRAM metadata access. Since a hit prediction is always correct (i.e., it is a DRAM cache hit and data is indeed present in the indicated way), there is no need for reading the metadata. Metadata *updates* may still be needed like in any cache management scheme (recency information or setting a dirty bit in case of a write). In our implementation, we do not maintain strict LRU and thus we do not update the metadata on every access. Since the way locator provides the top 2 MRU ways, our replacement scheme is "random-not-recent" - randomly replace a way that is not one of the top 2 MRU ways. It may be noted that since the way locator may have fewer entries than the number of sets, it can not provide MRU data for every set. In cases where the way locator does not hold top-2 MRU locations for a given set, a random way of that set is replaced. This scheme does well given the limited demand for older ways in large low level caches. With this, the *Bi-Modal Cache* eliminates accesses to the metadata bank whenever there is a way locator hit for DRAM cache reads. For writes, we update the dirty bits in the metadata bank but this is not in the critical path of data access.

2) Way Locator Miss, DRAM cache Hit: If there was a way locator miss, then the DRAM metadata bank is accessed, and all the tags are read. Since our design limits the highest associativity to 18 (in $(2, 16)$ state), we are able to read all the tags and associated attributes (big/small bits, additional offset bits for small ways) from DRAM in 2 DRAM bursts (each burst fetches 64 bytes)[10]. In parallel, the row in the bank that holds data for this set is activated. Once the tags are matched and a match is found, a DRAM column access is issued to the open row.

[10] In case of 4KB sets, the max. associativity is 36, and requires 3 DRAM bursts

3) DRAM cache Miss: If the access is a cache miss[11], then the DRAM cache controller first predicts the size of the block to be fetched from main memory using the block size predictor. Based on the predicted block size, the appropriate fetch is initiated from the off-chip memory.

4) Hit Latency Reduction with Way Locator: In order to understand the hit latency reduction of *Bi-Modal Cache* we have to examine the average tag access time which is dependent on the SRAM storage size (which determines the tag lookup time t_{tag_hit}), SRAM way locator hit rate h_{tag_hit}, and DRAM cache metadata lookup time t_{tag_miss}. We may express the average tag access latency as:

$$\begin{aligned} t_{tag\_access} &= h_{tag\_hit} * t_{tag\_hit} + (1 - h_{tag\_hit}) * t_{tag\_miss} \\ t_{tag\_miss} &\approx r_{tag\_row\_hit} * t_{tag\_col\_read} + (1 - r_{tag\_row\_hit}) * \\ &\qquad (t_{tag\_precharge} + t_{tag\_row\_open} + t_{tag\_col\_read}) \end{aligned}$$

For illustration, consider a 256MB DRAM cache over a 40-bit address space. A tags-in-SRAM organization can be modeled using the above equations as a tag store with $h_{tag\_hit} = 100\%$ and $t_{tag\_hit} = 7$ cycles. The way locator has a smaller SRAM storage and thus incurs a smaller t_{tag_hit} of 1 cycle at a table size of $120KB$. At a DRAM access timing of 10ns (32 cycles of a 3.2GHz processor), the way locator needs to achieve h_{tag_hit} of atleast 78% to perform better than the tags-in-SRAM organization. This model reveals the importance of achieving high h_{tag_hit} as well as reducing t_{tag_miss} for a cached-metadata organization to outperform the tags-in-SRAM. *Bi-Modal Cache* achieves $h_{tag\_hit} > 90\%$ by leveraging spatial locality and caching only the top-2 MRU blocks of sets (see Section V-F). Further, *Bi-Modal Cache* reduces t_{tag_miss} by over 30% (compared to a co-located tags and data scheme) by issuing tag reads to a dedicated metadata bank which has a higher average RBH ($r_{tag\_row\_hit}$) compared to data banks (see Section V-E). With a high hit-rate in *Way Locator* and high RBH for metadata accesses, *Bi-Modal Cache* achieves an average tag access latency of 3.6 cycles, which is nearly half the average latency of a tags-in-SRAM organization.

IV. Evaluation Methodology

We evaluated the performance benefits of *Bi-Modal Cache* using multiprogrammed workloads running on the GEM5 [12] simulation infrastructure to which we integrated detailed models of stacked DRAM caches and off-chip memory. The memory models faithfully account for all the significant timing and functional characteristics including hierarchical DRAM organization, key timing parameters (including refresh), data bus widths, and clock frequencies. The memory controller models implement all the key parameters, including command & data queues, request scheduling algorithms, and signalling commands to the memory. For quad-core workloads, timing simulations were run for 1 billion instructions on each core after fast-forwarding the first 10 billion instructions to allow for sufficient warm-up. As is the norm, when a core finishes its timing simulation, it continues to execute until all the cores have completed[12]. In case of 8 and 16-core workloads, due to the amount of simulation time required, we collected statistics on timing runs of $500M$ and $250M$ instructions per core respectively. In all cases, the total instructions simulated across all the cores amount to more than 4B.

Processor	3.2 GHz OOO Alpha ISA
L1I Cache	32kB private, 64B blocks, Direct-mapped, 2 cycle hit latency
L1D Cache	32kB private, 64B blocks, 2-way set-associative, 2 cycle hit latency
L2 Cache	For 4/8/16 cores: 4MB/8MB/16MB, 8-way/16-way/32-way, 128/256/512 MSHRs, 64-byte blocks, 7/9/12 cycles hit latency
AlloyCache (Baseline)	For 4/8/16 cores: 128MB/256MB/512MB, **Direct-Mapped**, **64-byte blocks**, MAP-I Predictor (1KB), Memory in 2/4/8 Channels Total of 16/32/64 DRAM banks, 2KB page, 128-bit bus width, 1.6GHz, CL-nRCD-nRP=9-9-9
BiModal DRAM cache	**512/64B blocks**, **2KB Set Size**, Way Locator: 77.8KB/81.9KB/86KB Size Predictor and Tracker: 32KB/36KB/46KB All the rest same as AlloyCache Baseline (Note: BiModal did not employ MAP-I miss predictor)
Memory Controller	For 4/8/16 cores: 1/2/4 off-chip data channels Each MC: 64-bit interface to channel, 256-entry command queue FR_FCFS scheduling [11], open-page policy Address-interleaving: row-rank-bank-mc-column
Off-Chip DRAM	For 4/8/16 cores: 4GB/8GB/16GB main memory using: DDR3-1600H, BL (cycles)=4, CL-nRCD-nRP=9-9-9 in 2/4/8 ranks, 16/32/64 banks Refresh related: T_{REFI} of 7.8us and T_{RFC} of 280nCK

Table IV: CMP configuration

In addition to timing simulations, we explored several design parameters/design choices of interest using a trace-based DRAM cache simulator. Traces were collected from GEM5 simulations of 4, 8, and 16-core architectures running for 75 billion instructions on each core. This has resulted in 120M – 450M accesses to the DRAM cache, with an average of 310M DRAM cache accesses per workload. The trace-driven simulator facilitated a comprehensive analysis of cache hit rate, RBH, way locator hit rate, block size predictor accuracy, cache space utilization and bandwidth across a wide range of DRAM cache parameters including cache size, block size, associativity, predictor table size and thresholds.

Our workloads are comprised of programs from SPEC 2000 and SPEC 2006 benchmark [13] suites. The 4, 8 and 16-core multiprogrammed workloads are listed in Table V. These benchmarks were carefully combined to create workloads with high, moderate and low levels of *memory intensity*[13] to ensure a representative mix. Workloads marked with a "*" in Table V have high memory intensity (*LLSC* miss rate $\geq 10\%$). We also measured the footprints of these

[11] We have not used a miss predictor. Works in [3], [4] propose SRAM based miss predictors which we could also deploy. This is an orthogonal optimization aimed at miss latency.

[12] The statistics are collected only during the first 1 Billion instructions.

[13] Intensity was measured in terms of the last-level SRAM cache miss rate.

Quad-Core Workloads
*Q1:(462,459,470,433), *Q2:(429,183,462,459), *Q3:(181,435,197,473),
Q4:(429,462,471,464), *Q5:(470,437,187,300), *Q6:(462,470,473,300),
*Q7:(459,464,183,433), Q8:(410,464,445,433), Q9:(462,459,445,410),
*Q10:(429,456,450,459), Q11:(181,186,300,177), Q12:(168,401,435,464),
Q13:(434,435,437,171), *Q14:(444,445,459,462),Q15:(401,410,178,177),
Q16:(300,254,255,470), *Q17:(171,181,464,465), Q18:(464,450,465,473),
*Q19:(453,433,458,410), Q20:(462,471,254,186),Q21:(462,191,433,437),
Q22:(197,168,179,187), Q23:(401,473,435,177),Q24:(416,429,454,175)
Q25:(254,172,178,188)
Eight Core Workloads
E1:(462,459,433,456,464,473,450,445), *E2:(300,456,470,179,464,473,450,445),
*E3:(168,183,437,401,450,435,445,458), *E4:(187,172,173,410,470,433,444,177),
E5:(434,435,450,453,462,471,164,186), E6:(416,473,401,172,177,178,179,435),
*E7:(437,459,445,454,456,465,171,197), E8:(183,179,433,454,464,435,444,458),
*E9:(183,462,450,471,473,433,254,168), *E10:(300,173,178,187,188,191,410,171),
*E11:(470,177,168,434,410,172,464,171),E12:(459,473,444,453,450,197,175,164),
E13:(471,462,186,254,465,445,410,179),*E14:(187,470,401,416,433,437,456,454),
*E15:(300,458,462,470,433,172,191,471),E16:(183,473,401,435,188,434,164,427)
Sixteen Core Workloads
*S1:(462,459,433,456,464,473,450,445,453,179,183,168,416,434,444,191)
*S2:(435,465,471,164,186,434,416,256,172,177,178,437,454,171,197,458)
S3:(462,473,254,168,183,453,300,173,187,178,188,410,171,434,470,191)
*S4:(470,177,464,171,172,168,434,410,175,164,444,450,254,465,179,471)
S5:(410,433,189,187,177,173,300,255,254,471,458,456,454,437,444,434)
S6:(191,189,177,183,179,168,470,164,470,464,459,450,435,256,416,445)

Table V: Workloads

workloads in terms of the number of distinct $64B$ blocks accessed. The average memory footprints in 4-core and 8-core workloads are 990MB and 2.1GB respectively. We also found that on average 87% of all the DRAM cache misses are due to capacity/conflict. Thus our workloads are sufficiently exercising the DRAM cache.

System performance is measured using the *ANTT* [14] metric, defined as: $ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{C_i^{MP}}{C_i^{SP}}$, where C_i^{MP} and C_i^{SP} denote the cycles taken by the i^{th} program when running in a multi-programmed workload and when running standalone, respectively.

The details of our baseline architecture and variants explored are listed in Table IV. Our baseline DRAM cache architecture is the *AlloyCache* organization [4].

V. Results

In this section, we evaluate the performance of the *Bi-Modal Cache* and compare it with other schemes.

A. System Performance

Figure 7 shows the performance improvement that *Bi-Modal Cache* achieves over the baseline in 4-core, 8-core and 16-core workloads. Note that the baseline is aggressive - it is a low-latency direct-mapped organization with interleaved tags and data to achieve efficient read-out of both from a single DRAM row. *Bi-Modal Cache* achieves on average, gains in ANTT of 10.8%, 13.8% and 14.0% respectively in 4, 8 and 16-core workloads. In order to understand the sources of these performance gains, we ran experiments with 2 additional configurations: *Bi-Modal-Only* - this configuration implements only bi-modal caching and no way location, and *Way-Locator-Only* - this configuration implements only way location on fixed sized (512B) blocks, and no bi-modality. Figure 8(a) shows the improvements in performance for all 3 configurations on 8-core workloads. As we can see, both components of the *Bi-Modal Cache* design, namely its ability to support and adapt to two different cache block sizes and its ability to locate the way quickly and access the data (without tag search) for a large majority of the accesses, independently yield significant performance benefits.

B. Improving Cache Hit Rate

The baseline scheme (*AlloyCache*) is organized as 64B blocks and as shown earlier in Figure 1, this has significantly lower cache hit rates. A fixed 512B block size greatly improves hit rates (average gain: 29%) by leveraging inherent spatial locality. As shown in Figure 8(b), the *Bi-Modal Cache* further improves hit rates (average gain: 38%) via improved cache space utilization.

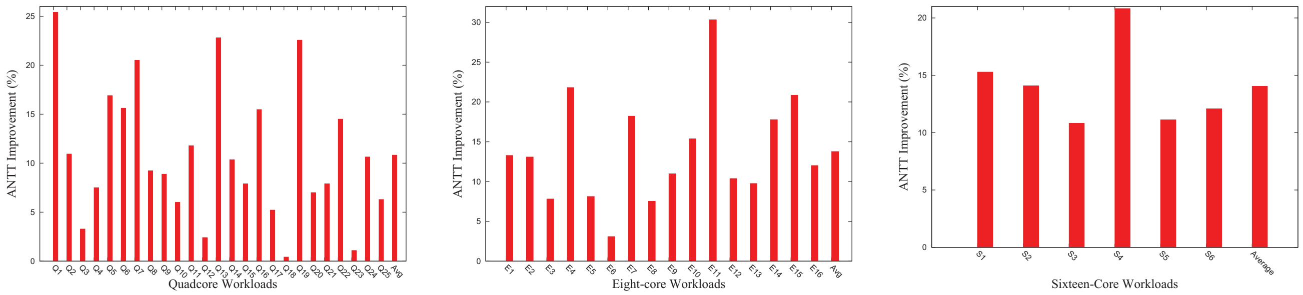
C. Reducing Access Latency

We measured the cache hit and miss latencies observed at the DRAM cache controller, including the time for way location and delays caused by contention. We compare the average access latency (i.e., average *LLSC* miss penalty) with several other schemes in Figure 8(c). *Bi-Modal Cache* achieves lower average latency (22.9%) over the baseline by virtue of a higher cache hit rate despite having nearly the same hit-latency as baseline.

1) Comparison with Footprint-Cache and ATCache: Footprint Cache [5] (*FPC*) organizes data in the form of large (1024B or 2048B) blocks and thereby manages to store metadata on SRAM. Only predicted sub-blocks of these large blocks are fetched into the cache to reduce wasted off-chip bandwidth. Further, it bypasses blocks predicted to have just one CPU reference to it. *Bi-Modal Cache* has 2 benefits over *FPC*. One, *FPC* incurs a higher tag lookup latency. Two, *FPC* commits large block space in the cache whenever its predictor indicates a utilization of $\geq$ 2 fine-grain blocks (64B size). Given that a good fraction of blocks (average of 18% for our workloads) have utilization levels $\geq$ 2 but $<$ 8 (see Figure 2), this causes internal wastage within a block and results in additional cache misses due to a virtually smaller cache. *Bi-Modal Cache* achieves average latency improvement of 12% over *FPC* resulting in 4.9% in ANTT (details not shown due to space restriction).

Finally, as compared to the *ATCache* [8][14], our scheme achieves a higher way locator hit rate than the tag-cache hit rate in the *ATCache*, higher DRAM cache hit rate (due to larger blocks) and lower DRAM hit latency resulting in significant latency improvement (26.5%). This translates to ANTT improvement of 14.8%. Thus *Bi-Modal Cache* achieves latency reduction over both tags-in-SRAM and tags-in-DRAM organizations by simultaneously improving cache hit rates and reducing hit latency.

[14] Our implementation used $PG = 8$.

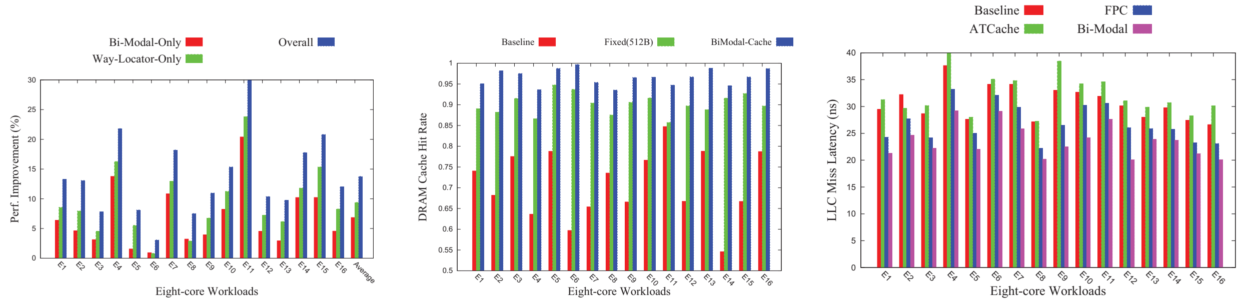


(a) Qaud-Core

(b) Eight-Core

(c) Sixteen-Core

Figure 7: Overall System Performance Improvement with *Bi-Modal Cache*



(a) Sources of Performance Gains

(b) Hit Rate

(c) *LLSC* Miss Latency (Lower is Better)

Figure 8: Understanding Performance Improvement with *Bi-Modal Cache*

D. Reducing Wasted Bandwidth

Bi-modality achieves a significant reduction in wasted bandwidth (more than 60%) compared to a fixed block size organization. Figure 9(a) plots the wastage in the fixed-512B block size organization and that incurred in the *Bi-Modal Cache* for 8-core workloads. In particular, the workloads $E8, E12, E14$ and $E15$ that suffered significant wastage in the fixed block size configuration have considerably benefited. On average, *Bi-Modal Cache* achieves savings of 67%, 62% and 71% in 4-core, 8-core and 16-core workloads over the fixed 512B block-size organization. These savings are substantial both from reducing contention on the off-chip bus (and thereby reducing miss latency) as well from an energy point-of-view.

As compared to the 64B baseline [3] that incurs no wasted off-chip bandwidth, the *Bi-Modal Cache* incurs only an additional 3.7% and 4.4% bandwidth in quad-core and eight-core respectively. Thus, *Bi-Modal Cache* is able to leverage a large block size without incurring significant additional bandwidth. A stricter threshold ($T > 5$) can be used to reduce this additional bandwidth consumption. In comparison to *FPC* [5], our organization reduces off-chip bandwidth consumption by 7.2% and 7.7% in quad-core and eight-core respectively. These savings are a result of improving cache utilization.

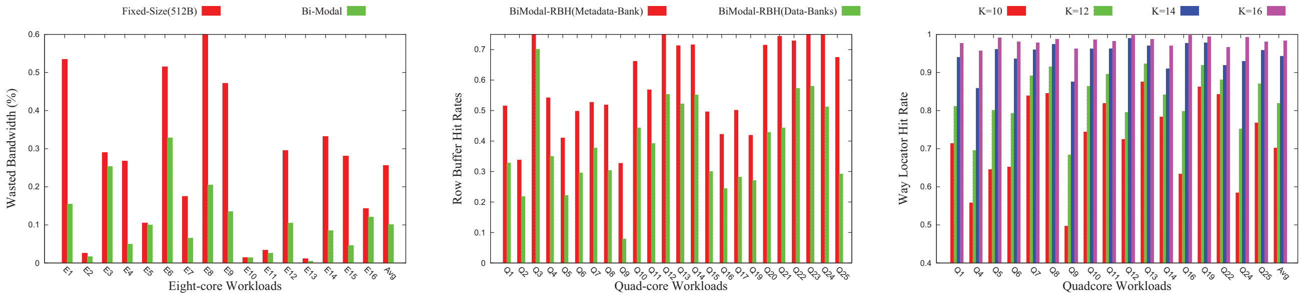
E. Improving the Metadata Row-Buffer Hit Rate

As discussed in Section III, by separating out the metadata into its own bank the DRAM row-buffer hit rates improve. Figure 9(b) shows this effect for several quad-core workloads[15]. On average, the metadata bank gains 37% hit rate improvement over one where data and tags are co-located in the same rows. Similar results are seen in 8-core and 16-core workloads (but not shown here due to space restriction). What this means from a latency perspective is that even when the way locator suffers a miss, this organization would suffer less latency by eliminating a good fraction of row-buffer activations and precharges.

F. Way Locator Hit Rates

The way locator's hit rate plays a key role in ensuring that most requests require just a single DRAM access. Figure 9(c) plots the way locator hit rates at different table sizes for selected quad-core workloads. A table size of $K = 14$ provides a good trade-off between hit rates (average: 95%) and table size ($77.8K$ for 128MB cache - refer Table III). Thus only a small fraction of accesses incur an additional DRAM access for metadata. At this table size, 8-core workloads achieve an average hit rate of 91%.

[15] In all cases, the averages are computed over all the workloads even if not all have been plotted for space reasons.



(a) Wasted Bandwidth (8-Core)

(b) *BiModal* RBH (Metadata bank)

(c) Way Locator Hit Rates

Figure 9: Bandwidth, RBH and Way Locator Hit Rate Improvements in *Bi-Modal Cache*

G. Bi-Modal Adaptation

Bi-modality enables different workloads to tailor the use of cache space suitably depending on the access characteristics. Figure 10 shows the fraction of accesses that go to small blocks. There is a wide variation across workloads with $Q17$ having only 1% of its accesses to small blocks while $Q23$ has 48% of its accesses to small blocks. This indicates that the *Bi-Modal Cache* adapts to workload characteristics well.

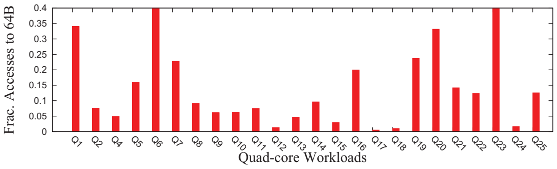


Figure 10: Fraction of Accesses to Small Blocks

H. Energy Savings

We computed the energy consumed using the number of accesses, DRAM cache hit rate, way locator hit rate, row buffer hit rates in the cache and main memory, and the amount of data transferred. *Bi-Modal Cache* saves off-chip energy by improving DRAM cache hit rate, and by leveraging higher spatial locality in off-chip accesses. While the baseline (*AlloyCache*) does not incur any wasted transfers, it suffers from low spatial locality and causes a high proportion of off-chip DRAM page activations and precharges. Further, its direct-mapped organization incurs more evictions.

The fraction of DRAM cache accesses that miss in the way locator ($< 5\%$ in quad-core) may increase energy by opening two banks. Of these, on average, only about 17% of accesses require two row-buffer activations (i.e., $< 0.85\%$ of all accesses in quad-core).

Figure 11 plots energy saving realized by *Bi-Modal Cache* for 8-core workloads. It achieves overall memory energy (DRAM cache + main memory) reduction of 11.8% for 8-core workloads (14.9%, and 12.4% on average in quad, and 16-core workloads respectively) over the baseline.

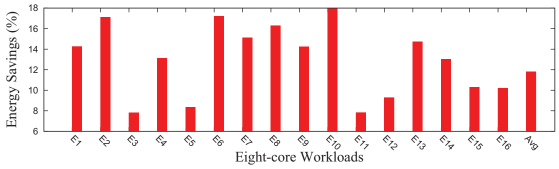


Figure 11: Off-Chip Energy Savings in 8-Core

N	*PREF_NORMAL*	*PREF_BYPASS*
1	9.8%	10.4%
3	8.7%	9.3%

Table VI: Performance (ANTT) Improvement Over Prefetch-Enabled Baseline

I. Sensitivity

Interaction with Prefetch: Prefetching can potentially hide the long latency of a cache miss. However, prefetchers also introduce their own complexity, e.g., wasted bandwidth if prefetched data was not actually needed, and cache pollution by evicting useful data early. We explored the effect of a hardware prefetcher on *Bi-Modal Cache* by adding a *next-N-lines* prefetcher [15] between the *LLSC* and the DRAM cache. This prefetcher observes the misses in the *LLSC* and initiates prefetching of the next N spatially adjacent cache blocks (64B) if these blocks are not already present in the *LLSC*. We introduced such a prefetcher in both the baseline (*AlloyCache*) as well as *Bi-Modal Cache*. We explored two settings of N, a conservative prefetcher at $N = 1$ and an aggressive prefetcher at $N = 3$. For the *Bi-Modal Cache*, we explored two different implementations of prefetch in the DRAM cache: (i) prefetch requests are treated exactly like normal accesses (*PREF_NORMAL*), and (ii) prefetch requests bypass the DRAM cache if they are misses in the DRAM cache (*PREF_BYPASS*). Table VI reports the performance improvements observed in *Bi-Modal Cache* relative to the respective prefetch-enabled baselines in quad-core workloads. This shows that the benefits of *Bi-Modal Cache* hold even in the presence of prefetching, with the smallest average gain being 8.7%.

Cache Size, Block Size and Associativity: We explore the performance benefit of *Bi-Modal Cache* with different cache sizes, block sizes and associativity. Figure 12 shows that this organization is able to exhibit performance benefit at both smaller (64MB) and larger (512MB) caches, with smaller (256B) and larger block sizes (1024B) and at higher associativity (8-way). The notation *BiModal(X-Y-Z)* refers to the *Bi-Modal Cache* of size X, big block size Y and big block associativity Z. All the improvements are over corresponding-sized *AlloyCache* configurations.

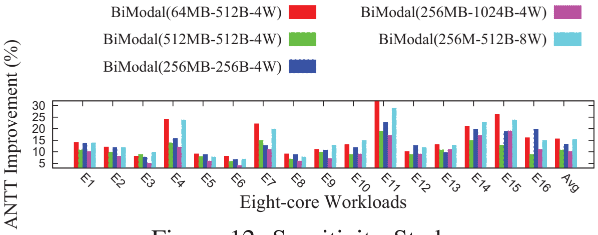


Figure 12: Sensitivity Study

VI. Related Work

In Sections II and III-A, we have already contrasted our work with recent DRAM cache organizational studies [8], [3], [4], [5]. The work in [6] aims to filter out infrequently used pages and cache only the few *hot* pages. In our work, we found that the proportion of hot pages can be substantial (as seen from Figure 2) thereby reducing or even eliminating opportunities for significant filtering. In [16], a DRAM cache organization that balances out cache and off-chip bandwidths is proposed to prevent a disproportionate load on the DRAM cache. This work is orthogonal to the ideas discussed in our proposal. The works in [17], [18] address SRAM cache line sizing in the presence of stacked DRAM main memory. Techniques related to SRAM cache line sizing assume that tag overheads are low, access times are small and that data movement/layout changes are affordable. These are all severe constraints in the DRAM cache space. There are quite a few works that discuss 3D stacked memory issues such as TSV bandwidth, resilience, and power [19], [20], [21], [22] that are orthogonal to the issues addressed in our work.

There are a number of studies around SRAM cache organizations [23], [24], [25] with the goals of achieving better hit rates, and latencies in multi-core systems. In [23] a variable-granularity cache organization at the $L1, L2$ levels is proposed to utilize space more efficiently. The proposed mechanisms are costly for implementation at the DRAM-Cache level both from a metadata storage as well as from hit, and miss evaluation cost. In [24], block utilization is tracked and dead blocks are used to retain useful victims from other sets. Similarly, the work in [25] proposes to organize the likely eviction candidates at small granularity retaining only the likely useful sub-blocks. At the DRAM cache level, we found very little benefit of retaining evicted (or likely to be evicted) blocks in a victim cache since there was very little temporal reuse.

Cache way prediction/memoization has been applied at the L1 and L2 levels [26], [27], [28], [29] to reduce energy at some trade off to access latency. Although the idea of way-prediction is old and well-known, we believe it has more role to play in the realm of DRAM caches and ours is the first work to leverage this idea at the DRAM cache level to reduce cache access latency significantly.

VII. Conclusions

In this work, we presented a DRAM cache organization that achieves improved cache hit rate, row-buffer hit rate, hit latency and off-chip memory bandwidth. By separating out the metadata into its own bank and using a way locator to point to the correct data location, the *Bi-Modal Cache* achieves hit latency reduction while also enabling metadata to reside on DRAM. This separation has the added benefit of improving the row-buffer hit rate. By organizing the cache sets to accomodate two sizes, the *Bi-Modal Cache* reduces wasted off-chip bandwidth as well as internal fragmentation. These combine to deliver improved performance. Overall, the *Bi-Modal Cache* improves ANTT by 10.8%, 13.8% and 14% for 4, 8, and 16-core workloads, over an aggressive baseline.

Acknowledgements

We would like to thank Prof. Mainak Chaudhuri and the anonymous reviewers of this paper for their insightful comments and feedback.

References

[1] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the bandwidth wall: Challenges in and avenues for CMP scaling,” *Proceedings of the 36th Annual International Symposium on Computer Architecture*, vol. 37, no. 3, pp. 371–382, 2009.

[2] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, “Die stacking (3D) microarchitecture,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 469–479.

[3] G. H. Loh and M. D. Hill, “Efficiently enabling conventional block sizes for very large die-stacked DRAM caches,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 454–464.

[4] M. K. Qureshi and G. H. Loh, “Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 235–246.

[5] D. Jevdjic, S. Volos, and B. Falsafi, “Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? Have it all with footprint cache,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 404–415.

[6] 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.” in *International Symposium on High Performance Computer Architecture*, 2010, pp. 1–12.

[7] M. D. Hill, “A case for direct-mapped caches,” *Computer*, vol. 21, no. 12, pp. 25–40, 1988.

[8] C.-C. Huang and V. Nagarajan, “ATCache: Reducing DRAM-cache latency via a small SRAM tag cache,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[9] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer, “Set-dueling-controlled adaptive insertion for high-performance caching.” *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, vol. 28, pp. 91–98, 2008.

[10] S. J. E. Wilton and N. P. Jouppi, “Cacti: An enhanced cache access and cycle time model,” *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 677–688, 1996.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” *Proceedings of the 27th Annual International Symposium on Computer Architecture*, vol. 28, no. 2, pp. 128–138, 2000.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[13] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[14] S. Eyerman and L. Eeckhout, “System-level performance metrics for multiprogram workloads,” *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[15] S. P. Vanderwiel and D. J. Lilja, “Data prefetch mechanisms,” *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, Jun. 2000.

[16] J. Sim, G. H. Loh, H. Kim, M. O’Connor, and M. Thottethodi, “A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 247–257.

[17] T. Ono, K. Inoue, and K. Murakami, “Adaptive cache-line size management on 3D integrated microprocessors,” in *SoC Design Conference*, 2009, pp. 472–475.

[18] K. Inoue, K. Kai, and K. Murakami, “Dynamically variable line-size cache exploiting high on-chip memory bandwidth of merged DRAM/logic LSIs.” in *International Symposium on High Performance Computer Architecture*, 1999, pp. 218–222.

[19] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee, “An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth.” in *International Symposium on High Performance Computer Architecture*, 2010, pp. 1–12.

[20] J. Sim, G. H. Loh, V. Sridharan, and M. O’Connor, “Resilient die-stacked DRAM caches,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 416–427.

[21] G. H. Loh, “3D-stacked memory architectures for multi-core processors,” *Proceedings of the 35th Annual International Symposium on Computer Architecture*, vol. 36, no. 3, pp. 453–464, 2008.

[22] L. Zhao, R. R. Iyer, R. Illikkal, and D. Newell, “Exploring DRAM cache architectures for CMP server platforms.” in *ICCD*. IEEE, 2007, pp. 55–62.

[23] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, “Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 376–388.

[24] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi, “Using dead blocks as a virtual victim cache,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 489–500.

[25] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, “Line distillation: Increasing cache capacity by filtering unused words in cache lines.” in *International Symposium on High Performance Computer Architecture*, 2007, pp. 250–259.

[26] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, “Reducing set-associative cache energy via way-prediction and selective direct-mapping,” in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001, pp. 54–65.

[27] K. Inoue, T. Ishihara, and K. Murakami, “Way-predicting set-associative cache for high performance and low energy consumption,” in *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, 1999, pp. 273–275.

[28] T. Ishihara and F. Fallah, “A way memoization technique for reducing power consumption of caches in application specific integrated processors,” in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, 2005, pp. 358–363.

[29] B. Calder, D. Grunwald, and J. Emer, “Predictive sequential associative cache,” in *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 1996.