Baryon: Efficient Hybrid Memory Management with Compression and Sub-Blocking

Yiwei Li[†] and Mingyu Gao[†‡]
Tsinghua University[†] Shanghai Qi Zhi Institute[‡]
liyw19@mails.tsinghua.edu.cn gaomy@tsinghua.edu.cn

2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071115

***Abstract*—Hybrid memory systems are able to achieve both high performance and large capacity when combining fast commodity DDR memories with larger but slower non-volatile memories in a heterogeneous way. However, it is critical to best utilize the limited fast memory capacity and slow memory bandwidth in such systems to gain the maximum efficiency. In this paper, we propose a novel hybrid memory design, Baryon, that leverages both memory compression and data sub-blocking techniques to improve the utilization of fast memory capacity and slow memory bandwidth, with only moderate metadata overheads and management complexity. Baryon reserves a small fast memory area to efficiently manage and stabilize the irregular and frequently varying data layouts resulted from compression and sub-blocking, and selectively commits only stable blocks to the rest fast memory space. It also adopts a novel dual-format metadata scheme to support flexible address remapping under such complex data layouts with low storage cost. Baryon is completely transparent to software, and works with both cache and flat schemes of hybrid memories. Our evaluation shows Baryon achieves up to 1.68× and on average 1.27× performance improvements over state-of-the-art designs.**

***Index Terms*—Hybrid memory, compression, sub-blocking**

I. Introduction

With the surge of big data [40] and deep learning [41], the performance and energy efficiency of processing these memory-intensive applications in modern computer systems are largely dominated by the prominent DRAM-based main memory. To overcome the slowdown of DRAM scaling, a large variety of non-volatile memory (NVM) technologies have been developed [10], [25], [39], [42]. They are expected to offer larger storage capacity thanks to better density scaling and cost efficiency [42], [76], but also exhibit worse latency and bandwidth than existing DRAM.

Consequently, architects start to heterogeneously combine the regular DRAM-based *fast* memory and the emerging NVM-based *slow* memory into a *hybrid memory system* [14], [16], [30]–[32], [36], [38], [46], [51]–[53], [55], [62], [65]–[67], [75], with the hope to achieve both high performance and large capacity. Such hybrid memory systems can be managed in different ways. We can either use the fast memory as a cache [16], [30]–[32], [52], [65], [75], or as part of the physical memory space [14], [36], [46], [51], [53], [55], [62], [66]. We can coordinate data placement and migration with the help of the operating system [2], [33], [37], [38], [46], [53], [69], [71], [75], or hide these details completely in hardware [14], [16], [30]–[32], [36], [51], [52], [55], [62], [67].

However, the performance and efficiency of hybrid memory systems are largely restricted by the limited *fast memory capacity* and *slow memory bandwidth*, which must be carefully utilized. Two techniques could help. *Memory compression* exploits the value similarities among neighboring data and condenses multiple blocks into one, therefore reducing the data size when fetched from the slow memory and stored in the fast memory [5], [7], [17], [19], [20], [50], [57]–[59], [64], [73], [74]. *Sub-blocking* (a.k.a., *sectoring* [54]) divides a data block into multiple smaller chunks, and only fetches and stores the actually demanded sub-blocks to improve bandwidth and capacity utilization [32], [55], [67]. Consequently, the reduced data size from compression and the better utilization from sub-blocking both allow us to put more useful data in the fast memory and transfer fewer data from the slow memory.

Nevertheless, efficient support for compression and sub-blocking in hybrid memories is challenging. Both techniques call for a much more fine-grained management due to the smaller basic unit of data. With the same total memory capacity, using a smaller data tracking granularity would significantly increase the number of metadata entries for address remapping in hybrid memory systems, resulting in higher **metadata storage** overheads. This is unacceptable because the metadata cost in existing hybrid memories is already quite high and they can only be stored off-chip and partially cached on-chip. Further increasing would result in both storage and performance problems. In addition, the **compressed and sub-blocked memory layout** would become highly irregular and complex, with various-sized data from multiple different source blocks squeezed in one space. When data are constantly accessed and modified, the layout would also frequently change, making it even more difficult to efficiently cache and/or migrate data between the two memory tiers and accurately track their status.

In this paper, we propose Baryon, *a novel hardware-based hybrid memory architecture that leverages both compression and sub-blocking to better utilize both fast memory capacity and slow memory bandwidth, with only moderate metadata overheads and management complexity*. Baryon is completely transparent to system- and application-level software. It also supports common hybrid memory schemes that either use the fast memory as a cache or as part of the physical memory space, or even a static combination of both. To our best knowledge, Baryon is the first work to apply both compression and sub-blocking to hybrid memory systems.

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

We design Baryon based on a key insight. *While the compressed and sub-blocked data layouts are highly irregular and complex, once the critical subset of data from a specific block have been transferred to the fast memory, in most cases their footprints and compression schemes would quickly stabilize and do not significantly change afterwards*. Consequently, we reserve a small **stage area** in the fast memory to manage and stabilize the compressed and sub-blocked layout of each data block (i.e., the fetched block/sub-block IDs and their sizes after compression) for a short period of time. We **selectively commit** the block to the ordinary fast memory space based on its final layout stability. Most complicated operations due to compression and sub-blocking are restricted in the small stage area, with efficient fetch and replacement policies. Furthermore, we use a **dual-format metadata scheme** to balance between fine-grained data remapping and metadata storage cost. The stage area uses a highly flexible metadata format with large entries, storing sub-blocks from arbitrary blocks within a super-block. The stage area is small and only incurs moderate overheads. In contrast, the rest of the memory uses a more compact metadata format to save storage cost, but sacrifices flexibility without affecting much performance, because committed block layouts rarely change.

We evaluate Baryon against state-of-the-art hybrid memory and DRAM cache designs [31], [67], [74] using various benchmarks [9], [21], [29], [48] with GB memory footprints. Baryon achieves up to 1.68× and on average 1.27× speedups over the DRAM cache baselines [31], [74], and up to 2.50× and on average 1.18× over a recent hybrid memory system [67]. The speedups mainly come from the higher chances of locating data in the fast memory due to better capacity utilization, and the reduced memory traffic to both fast and slow memories. These results demonstrate that Baryon is highly effective for improving the fast memory capacity and slow memory bandwidth utilization under fine-grained management.

II. Background and Motivation

Hybrid memory is an emerging architecture that potentially offers both high access performance and large storage capacity. It smartly combines the conventional commodity DRAM technology with new non-volatile memories (NVMs). In this paper, we call the traditional DRAM as *fast memory*, and the emerging NVM as *slow memory*.

A. Design Space of Hybrid Memory

Cache vs. flat schemes. Hybrid memory systems can be organized in two forms: the cache (or vertical) scheme, and the flat (or horizontal) scheme. In the *cache* scheme, the fast memory effectively becomes the last-level (DRAM) cache after the processor's cache hierarchy, buffering the most frequently used data from the slow memory [16], [30]–[32], [52], [65], [75]. Such a scheme usually uses simple hardware management, and is made transparent to the operating system (OS) where only the slow memory is OS-visible. However, the lost capacity of fast memory may be crucial for applications with large memory footprints. To fully utilize all memory spaces, in the *flat* scheme, both memories are OS-visible, composing a large and unified physical address space [14], [46], [51], [53], [55], [62], [66]. The tradeoff is that the flat scheme requires more expensive *swaps* to migrate data in both directions between the two memories, potentially doubling the traffic and resulting in additional write-after-read dependencies [68]. Given these tradeoffs, our work aims to support both schemes.

OS-based vs. hardware-based management. In both cache and flat schemes, we need to dynamically identify the most critical data, and cache or migrate them into the fast memory. An address *remapping* mechanism is thus needed to flexibly move and locate a data block at different places in fast and slow memories. Such remapping could be done in the OS or in hardware. The OS-based solutions directly change the physical addresses in the page table by reusing the virtual-to-physical address translation [2], [33], [37], [46], [53], [69], [71]. This method has several limitations due to substantial software-level overheads and coarse-grained 4 kB page-level management. In contrast, OS-transparent, hardware-based hybrid memory management typically keeps physical addresses unchanged, and introduces another level of indirection with a physical-to-"device" address *remap table* [14], [16], [30]–[32], [38], [51], [52], [55], [62], [67]. When migrating data, only the device addresses in the remap table are updated. These hardware designs adapt more quickly to access patterns, and are fine-grained to more efficiently utilize bandwidth and capacity. Thus in this work we focus on hardware-based approaches.

B. Design Opportunities and Challenges

While the total memory capacity in the flat scheme is larger than that of the cache scheme, the precious fast memory capacity is kept the same, and usually more difficult to expand than the slow memory as technologies advance further. In addition, there is a significant bandwidth gap between the two memory tiers. Therefore, to optimize the hybrid memory performance, we must efficiently utilize the **fast memory capacity** as well as the **slow memory bandwidth**.

However, in existing designs, these two utilizations usually trade one for the other. In order to best utilize the fast memory capacity, frequent data migration and replacement are needed to keep only the most critical data and to adapt quickly to access pattern changes [43], [51], [53], causing significant traffic to the slow memory. On the other hand, some solutions aim to balance the overall bandwidth usage with bypass, selective, or epoch-based replacement policies [15], [16], [24], [55], [75], which may sacrifice the fast memory hit rate.

In this work, we leverage two techniques to achieve better utilization for *both* fast memory capacity and slow memory bandwidth. First, we apply *sub-blocking*, which divides a data block into multiple small sub-blocks. Based on runtime access characteristics, it only fetches the actually needed subset of sub-blocks into the fast memory, and thus saves the bandwidth consumption for the unneeded data (Fig. 1(a)). Traditional sub-blocking [32], [55], [67] leaves the fast memory space of the unfetched sub-blocks unused; a recent sub-blocking proposal, named micro-sector cache [12], improves the fast memory

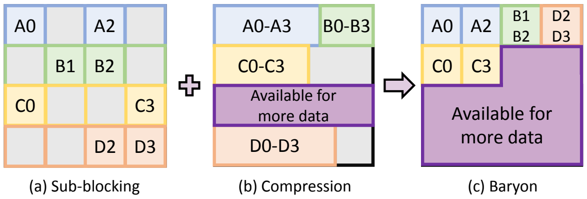


Fig. 1. Illustration of the fast memory data layouts with the three techniques. (a) Sub-blocking leaves unfetched sub-blocks unused. (b) Compression wastes space if blocks are not sufficiently compressible. (c) Baryon integrates sub-blocking and compression to enable more available fast memory space.

capacity utilization by further allowing the fetched sub-blocks from different blocks to share the same physical block.

Then, we increase the effective fast memory capacity using data *compression* (Fig. 1(b)). Compression has been widely studied in main memories [7], [17], [19], [20], [50], [64], [73] and caches [5], [22], [57]–[59], [63], [74]. It exploits the value similarities among neighboring data, a.k.a., the *compression locality*, and condenses multiple data blocks in a single physical block space. The same fast memory capacity can thus store more data, usually by a factor of 1.5 to 2 [17], [57], [59], [60]. Furthermore, when workloads exhibit spatial locality, transferring multiple compressed data blocks as one physical block also saves bandwidth usage [50], [61], [74].

Challenges. However, practically applying compression and sub-blocking to hybrid memory still faces challenging issues. First, by compressing or splitting a large block, both compression and sub-blocking effectively manage data in smaller units, typically a few hundreds of bytes or even 64 B cachelines. Such a granularity is much smaller than OS pages, and may result in significant **metadata storage overheads**. Particularly, the remap table, which is already large and cannot fit in on-chip SRAM, now needs to keep one metadata entry for each compressed sub-block of 64 to 256 B instead of one full block (say, 2 kB). This immediately results in up to 32× growth in size and can easily reach a few GB for even moderately large memory capacities, eating up a non-negligible part of the precious fast memory. Moreover, it also reduces the coverage of the on-chip *remap cache* [38], [51], [62], [67], making it less efficient to avoid off-chip metadata lookups.

Second, compression and sub-blocking also result in **complex, irregular, and varying data layouts**, making system management much more difficult. As data are fetched into the fast memory in the sub-block granularity, and further compressed into even smaller sizes, the layout of a physical block can be quite complex and irregular, containing various sub-blocks from different blocks, and with different compression factors (Fig. 1(c)). Furthermore, as more data are continuously accessed and modified, the set of cached/migrated sub-blocks need to be adaptively updated, and may also be recompressed into a different size that no longer fits in the original space. Essentially, it is difficult to foresee how many sub-blocks would be cached/migrated and what their compressed sizes would be, making it challenging to efficiently manage data layouts in the fast memory.

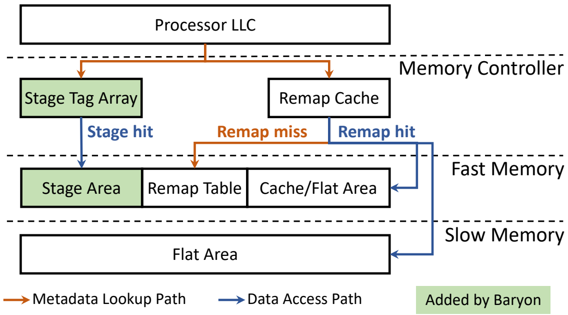


Fig. 2. Design overview of Baryon. New changes in Baryon are highlighted.

III. Design

We propose Baryon, which to our best knowledge is the first hybrid memory architecture that supports both *memory compression* and *data sub-blocking*, in order to maximize the utilization of both fast memory capacity and slow memory bandwidth. Baryon achieves better performance with only *moderate metadata overheads and management complexity*. The key ideas in Baryon are to use a novel *dual-format metadata scheme* to support both compression and sub-blocking without excessive metadata storage, and to reserve a small *stage area* in the fast memory with efficient *fetch, replacement, and commit policies* to flexibly manage and stabilize the compressed and sub-blocked data layouts.

Baryon is completely transparent to both system and application software. It only modifies the hardware memory controller, and manages data caching/migration through a hardware-based physical-to-device address remap table. Unmodified OS'es and applications could directly benefit from Baryon. Baryon also supports both cache and flat schemes.

A. Baseline Hybrid Memory System

Fig. 2 shows a typical hybrid memory architecture [38], [51], [62], [67], on top of which Baryon adds its new extensions. The fast memory can be flexibly (but statically) partitioned into *cache* and *flat* areas. We use 2 kB data blocks [38], [55], [62], [67], to align with DRAM pages and reduce the remap table size. The physical address of a data block is translated through the hardware-managed *remap table*, into the actual device address that may be in the fast memory cache area (invisible to OS) or in the (either fast or slow) flat area. The remap table resides in the fast memory, but its entries can be cached in an on-chip SRAM *remap cache*. The remap table uses a simple linear format: each data block in the entire physical address space has an entry that contains a pointer to the actual device address. In this baseline, each memory access first probes the remap cache (and the off-chip remap table if missed) to get the remapped device address, from which it then accesses the actual data. If the data block is currently in the slow memory, after responding to the processor last-level cache (LLC), we may further cache or migrate the block into the fast memory, according to the system policies.

To reduce complexity, the hybrid memory is organized as set-associative, i.e., divided into multiple sets where each includes a certain number of fast and slow blocks. We define *associativity* as the number of fast blocks in a set. Caching and migration only happen among blocks within a set. Therefore the remap entry only needs to store a relative position within a set. With a lower associativity (e.g., 1 to 4 fast blocks) [31], [38], [52], [55], [62], this pointer can be much shorter than the full address, and each remap entry is only a few bits. On the other hand, highly/fully associative designs [44], [51], [67] reduce conflicts at the expense of higher storage and lookup overheads. For simplicity, Baryon uses a moderate associativity of 4. We discuss higher associativities in Section III-F.

Note that in this design, the remap table covers both the cache and the flat area. Pure DRAM caches may use conventional tag arrays (essentially the inverted mapping from device to physical addresses) either in SRAM or embedded in DRAM [23], [28], [45], [52], and may use a way predictor to replace the remap cache [31], [55], [72]. These designs reduce metadata lookup latency, but are limited to low associativities. Baryon chooses not to use them.

B. Baryon Overview

Baryon extends the above baseline with compression and sub-blocking. Baryon divides each 2 kB block into eight 256 B sub-blocks. We choose 256 B as our sub-block size to balance between spatial locality and bandwidth consumption [67]. We evaluate other sub-block sizes, e.g., 64 B [32], [55], in Section IV. Baryon allows a subset of a block's sub-blocks to be cached or migrated into the fast memory. These cached/migrated sub-blocks are organized into one or multiple *contiguous and aligned* ranges [58], each of which is individually compressed and must fit in one sub-block space, realizing a dense layout. This works well in practice because nearby data usually have similar contents and can be effectively compressed, known as the compression locality [50], [57]. Baryon therefore saves both fast memory capacity and slow memory bandwidth for the rest rarely accessed data.

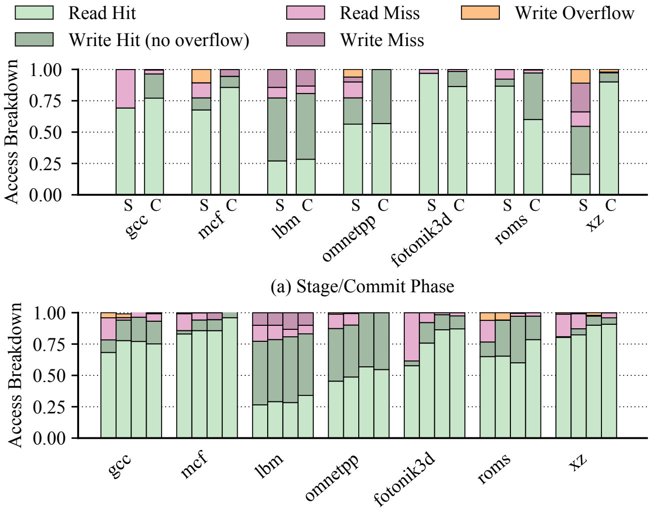
Following prior work [74], Baryon uses two compression algorithms, FPC [4] and BDI [49], which have efficient hardware implementations and offer a balance between compressibility and cost. We feed to-be-compressed data into both hardware modules and accept the one with the higher compression factor (CF). Alternative schemes [6], [13], [34] can also be used and the exact choices are orthogonal to our design. We support three CFs of 1, 2, or 4, following prior observations [26], [27], [57]–[59]. Therefore, each physical sub-block space may store one uncompressed sub-block (CF = 1), two sub-blocks with CF = 2, or four sub-blocks with CF = 4. Decompression is on the critical path of fast memory access, but usually costs only 1 to 5 cycles [4], [13], [49], [74]. This is much lower than the difference between typical fast and slow memory access latencies, so Baryon decides to always enable compression in the fast memory. Selective compression is promising to further reduce this overhead, but may need software semantic hints with extra design complexity. We leave it for future work.

Key innovation: stage area and selective commit. To address the layout challenge in Section II-B, Baryon introduces a small *stage area* in the fast memory (highlighted in Fig. 2) to buffer (called *stage*) incoming data that are just fetched from the slow memory and have not reached a stable layout. It only takes a small portion (e.g., 64 MB) of the fast memory, and is set-associative with 8192 sets and 4 ways per set. Each stage set is shared by multiple cache/flat sets. Only after a staged block is sufficiently stabilized, do we move it into the fast memory cache or flat area (called *commit*). Commits are *selective*; blocks that are still unstable when evicted out of the stage area are put back to the slow memory.

The stage area relies on a key novel observation. *While the overall sub-blocked and compressed data layouts constantly change, once all the sub-blocks of a specific block that should be cached/migrated have been moved into the fast memory after the initial cold misses, both its sub-block footprints and compression factors would likely remain fairly stable afterwards.* In other words, when a data block is just fetched into the fast memory, the unstable layout causes many data replacements and misses to the slow memory. After we spend a short period of time accumulating its frequently accessed data in the stage area, very few later accesses to this block would require a sub-block outside the accumulated subset, or make the accumulated sub-blocks significantly less compressible to not fit in their slots anymore. Therefore the number of misses would significantly reduce. Previous work leveraged the stabilization of simple sub-block footprints [32]; we extend the observation to also cover compression and more complex data layouts, with novel hardware designs.

To verify the above observation and the effectiveness of the stage area, we analyze the SPEC CPU2017 workloads [48]. Fig. 3(a) shows that when a block is just staged (the "S" bars), there are high chances for sub-block read/write misses, as well as write overflows where updated data cannot fit in the compressed space anymore. After commit (the "C" bars), both of these ratios decrease, to less than 5% and 1% on average, respectively. For example, the overflow rate of xz reduces from 10.9% to 2.0%. Fig. 3(b) further uses different stage area sizes from 16 MB to 128 MB. Larger stage area sizes reduce the miss/overflow chances of commit blocks, while 64 MB is generally sufficient. In Section IV, we demonstrate the end-to-end performance benefits of using a stage area, which could be up to 1.24× (Fig. 13(c)).

Fig. 4 further illustrates how the layout stability changes across time during the stage phase, i.e., when blocks are in the stage area. We sample 1k blocks and measure their stage area MPKI values, and use the reduction of misses to reflect the layout stability improvement. We aggregate the results of all blocks at the same *relative* time of their stage phases, i.e., the x-axis of each block is normalized to its unique stage phase length so that $x = 1$ indicates the time when the block is commit/evicted out of the stage area. We can see that, blocks have higher MPKI initially at $x = 0$, but the MPKI distribution quickly reduces by an order of magnitude, and keeps low after $x = 0.5$, i.e., halfway through the stage phase. This means that the layouts of most blocks can sufficiently stabilize before commit ($x = 1$). However, there still exist a small number of



(a) Stage/Commit Phase

(b) Multiple Stage Area Sizes

Fig. 3. Breakdown of access types when using a stage area. SimPoint [70] is used to identify a representative execution phase of each workload. Hit means accessed data are in the fast memory. Write overflow means updated data cannot be compressed to the original size. (a) When blocks are just staged (S) and later just committed (C), using a 64 MB stage area. (b) When blocks are just committed out of the stage area, using different stage area sizes (16 MB, 32 MB, 64 MB, 128 MB).

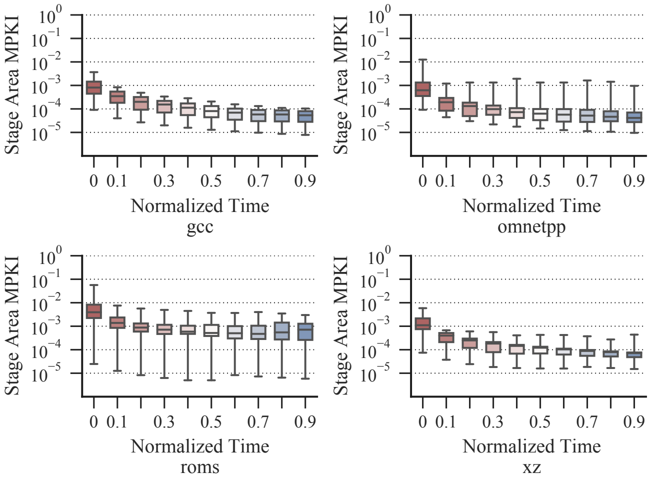


Fig. 4. Stage area MPKI distribution trends of 1k sampled blocks observed across their stage phases. The x-axis of each block is normalized to its unique stage phase length so that $x = 1$ indicates the time of commit/eviction out of the stage area. The box boundaries show the 25%/75% quartiles and the whiskers show 5%/95% tails.

unstable blocks with high MPKI values even at $x = 0.9$ (the 95% tail is notably higher than the average), which motivates a selective commit policy as discussed in Section III-E.

Key innovation: dual-format metadata scheme. Baryon also uses a *dual-format metadata scheme* to alleviate the metadata overheads without sacrificing flexibility. First, the stage area has a relatively small size (e.g., 64 MB), but the metadata need to be frequently accessed and updated in response to the rapidly changing data layouts. We therefore use an on-chip *stage tag array* in the memory controller (highlighted in Fig. 2), with one entry per stage area block. Each entry uses a flexible and fine-grained metadata format that is friendly to rapid data layout changes but requires significant size. In order to reduce the stage tag array size, we further adopt a hierarchy of multiple data granularities, with a few moderate restrictions on which sub-blocks can be cached/migrated to which locations. Eventually, each entry fits in 14 B and the total stage tag array is 448 kB, with the identical set-associative structure as the stage area.

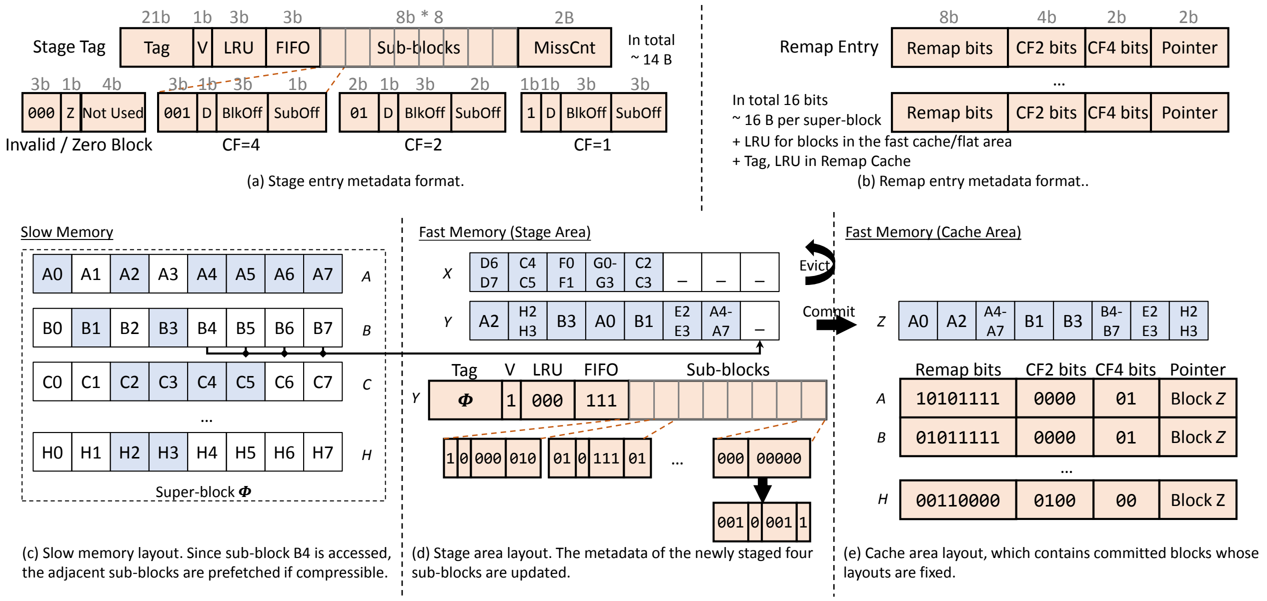
On the other hand, the committed blocks in the cache/flat areas use a more compact metadata format, similar to conventional remap tables but also supporting sub-blocking and compression. We again rely on the aforementioned insight that committed block layouts are fairly stable, and translate the metadata (and also the data layout of the block) into a format that disables any further layout change, but saves over 7× size than the stage tag (from 14 B to 2 B). This makes the full remap table occupy only 0.1% of the total system memory capacity, well fitting in a small portion of the fast memory. We further use a customized remap cache of 32 kB, which can achieve typical hit rates of over 90%. Together with the stage tag array, the total SRAM usage in the memory controller is 480 kB, comparable with previous works [38], [51], [62], [67] (see Table I for configuration details).

C. Data Layouts and Metadata Formats

Most previous sub-blocking designs only allow one fast memory block space to contain sub-blocks from a single data block [32], [55], [67], in order to share the same tag and remap pointer to reduce metadata overheads, but with the drawback of wasting the spaces for absent sub-blocks. Such space waste would be even more significant if compression makes the cached/migrated data even smaller.

To fully exploit the fast memory capacity benefits enabled by sub-blocking and compression, Baryon allows many sub-blocks *from multiple data blocks* to co-exist in one physical block space (Fig. 1(c)). Naively doing so would unfortunately enlarge the metadata size, as one entry for the physical block needs to include multiple long tags and/or pointers to indicate which blocks the sub-blocks come from. Baryon deals with this challenge by adopting a *dual-format metadata scheme*. The scheme uses two formats, for the stage tag array and the remap table, respectively, with different tradeoffs between flexibility and size. It also enforces a few moderate restrictions on which sub-blocks can be cached/migrated to which locations, and whether the data layouts are allowed to change.

Stage tag format. First, Baryon adopts an extra granularity, by organizing multiple blocks into a *super-block*. Each physical block space can only contain the sub-blocks from a single super-block (**Rule 1**), and only one super-block tag is needed [54], [58]. Based on the common statistics, typical compression factors (CFs) are up to 4 [17], [26], [27], [57]–[60], and usually less than half of the sub-blocks in one block are cached/migrated [32]. We thus group eight blocks into a 16 kB super-block. We explore other super-block size choices in Fig. 13(b). Note that Baryon still allows the data from one super-block to occupy more than one physical blocks,



21b
1b
3b
3b
8b * 8
2B
Stage Tag
Tag
V
LRU
FIFO
Sub-blocks
MissCnt
In total ~ 14 B
3b 1b 4b
000 Z Not Used
Invalid / Zero Block
3b 1b 3b 1b
001 D BlkOff SubOff
CF=4
2b 1b 3b 2b
01 D BlkOff SubOff
CF=2
1b 1b 3b 3b
1 D BlkOff SubOff
CF=1
(a) Stage entry metadata format.
8b 4b 2b 2b
Remap Entry
Remap bits
CF2 bits
CF4 bits
Pointer
...
In total 16 bits
~ 16 B per super-block
+ LRU for blocks in the fast cache/flat area
+ Tag, LRU in Remap Cache
(b) Remap entry metadata format..
Slow Memory
A0 A1 A2 A3 A4 A5 A6 A7 A
B0 B1 B2 B3 B4 B5 B6 B7 B
C0 C1 C2 C3 C4 C5 C6 C7 C
...
H0 H1 H2 H3 H4 H5 H6 H7 H
Super-block Φ
(c) Slow memory layout. Since sub-block B4 is accessed, the adjacent sub-blocks are prefetched if compressible.
Fast Memory (Stage Area)
X D6 D7 C4 C5 F0 F1 G0-G3 C2 C3 – – –
Y A2 H2 H3 B3 A0 B1 E2 E3 A4-A7 –
Tag V LRU FIFO Sub-blocks
Y Φ 1 000 111
1 0 000 010
01 0 111 01
...
000 00000
001 0 001 1
(d) Stage area layout. The metadata of the newly staged four sub-blocks are updated.
Evict
Commit
Fast Memory (Cache Area)
Z A0 A2 A4-A7 B1 B3 B4-B7 E2 E3 H2 H3
Remap bits CF2 bits CF4 bits Pointer
A 10101111 0000 01 Block Z
B 01011111 0000 01 Block Z
...
H 00110000 0100 00 Block Z
(e) Cache area layout, which contains committed blocks whose layouts are fixed.

Fig. 5. Data layouts and metadata formats in the stage area and the cache/flat areas. A, B are data blocks; A0, B0 are sub-blocks; X, Y are physical blocks.

e.g., physical blocks X and Y for data from super-block Φ in Fig. 5(d). Such cases are rare (1.12% in our experiments), but nevertheless helpful if some super-blocks have many hot data that are less compressible.

Additionally, we further require that the cached/migrated sub-blocks are organized in *contiguous and aligned* ranges (**Rule 2**). With three possible CFs in Baryon, the ranges could be a single uncompressed sub-block (e.g., A0 in Fig. 5(d)), two aligned sub-blocks with a CF of 2 (e.g., H2-H3), or four aligned sub-blocks with a CF of 4 (e.g., A4-A7). Under such restrictions, we can store shorter sub-block offsets and no explicit sub-block numbers in the metadata, both implied by the CF. These restrictions have minor impacts in practice (Fig. 12), thanks to the high spatial locality and value compression locality in many workloads [50], [57].

Consequently, the metadata format of the stage tag array is shown in Fig. 5(a). Each entry represents a 2 kB physical block in the stage area. There is a single super-block Tag (Rule 1, $48 - \log(16\text{kB}) - \log(8192\,\text{sets}) = 21\,\text{bits}$) and a valid bit V. Each of the eight sub-block spaces can store a contiguous and aligned range of sub-block data (Rule 2), with CF as 1, 2, or 4. Each range is encoded into 8 bits following one of the four types, with the CF code, a dirty bit D, the block offset within this super-block (BlkOff), and the starting sub-block offset within this block (SubOff). The number of sub-blocks is implicitly determined by the CF. A special encoding with the Z bit supports all-zero blocks. Finally, the entry uses a 3-bit LRU field and a 3-bit FIFO field for two-level replacements, and a 2 B MissCnt for selective commits, both discussed in Section III-E. In total, each entry needs 108 bits or 14 B.

Fig. 5(d) also shows an example. The physical block Y contains only sub-blocks from the super-block with tag Φ. Its sub-block space, e.g., the second one for H2-H3, is encoded with 01 (for CF = 2), 0 (clean), 111 (the 8th block H), and 01 (the 2nd aligned range of 2 sub-blocks).

Remap entry format. While the small stage area can use a longer and more flexible format, the large number of remap table entries must use a more compact representation to save space. In a conventional remap table, each data block entry contains one pointer to its cached/migrated location [38], [55]. We enforce the same requirement, i.e., all sub-block data from the same block, if present in the fast memory, must reside in the same physical block space (**Rule 3**). Furthermore, we also leverage the observation that committed data have relatively stable layouts that rarely change. Therefore, when a block is committed from the stage area into the fast memory cache/flat areas, we sort its compressed sub-blocks and fix the sorted layout to disallow any further layout change (**Rule 4**). Therefore, the remapped sub-block location could be determined by *how many sub-blocks from the blocks before this one in the same super-block are also remapped to this physical block.*

As a result, a more compact but less flexible format is used for the remap entry, as shown in Fig. 5(b). We still associate one remap entry with each block. Eight Remap bits indicate which sub-blocks have been cached/migrated, whose locations are recorded by the single Pointer field (Rule 3); other sub-blocks remain in their original locations. The Pointer is short in low-associative designs (2 bits for 4-way). If it points to an OS-visible address, this is a migrated block to the flat area; otherwise it is in the cache area. Additional CF2 and CF4 bits denote the corresponding contiguous and aligned ranges (Rule 2). A special invalid state of CF2 and CF4 (i.e., all 1's) encodes the all-zero case if the Z bit is set. Note that we may also need dirty bits and LRU information for replacement and writeback, but these metadata are per physical block (blocks X, Y, Z in Fig. 5), in contrast to the remap entry which is per logical data block (blocks A, B, C in Fig. 5). So we store them in another table, separately from the remap entries.

To look up the location of a cached/migrated data sub-block, we first get all entries for the blocks in the same super-block (only tens of bytes). We keep those that are *before* the target block and have the same Pointer as the target block. We then check the Remap and CF2/CF4 bits in these entries to detect how many sub-blocks are in the same physical block pointed by Pointer, and how much space they occupy before the target sub-block. Then we can know the position of the target sub-block given the sorted layout. As an example, to search B3 in Fig. 5(e), we realize both A and B blocks have sub-blocks in the same physical block Z. The Remap and CF2/CF4 bits say A0, A2, A4-A7, and B1 each takes one sub-block space. So B3 is in the 5th sub-block of Z. Generally, the remapped location is equal to the number of valid remap bits, minus valid CF2 bits, and minus 3× valid CF4 bits, all from the remap entries of the super-block before the current block.

The remap cache is correspondingly modified to better support this format. First, we organize the remap cache in the super-block granularity instead of individual blocks. This means each line contains eight entries (16 B, which are read together as mentioned before) plus a super-block tag. Then we add the logic to do the above sub-block position calculation from one super-block line, which is essentially eight parallel decoders and a prefix sum unit. Such logic has a minor delay. Such changes neither significantly impact the hit rate nor the hit latency. Our 32 kB remap cache is still comparable to the conventional one, with over 90% hit rates.

D. Access Flow

We now illustrate the detailed access flow of Baryon, especially the operations to the newly introduced stage area and stage tag array described above.

To access a physical address, we start by looking up the associated metadata in the stage tag array and the remap cache. We use the super-block tag to associatively search these two on-chip SRAM structures *in parallel* (the orange arrows in Fig. 2). For all matched entries (usually one but occasionally multiple) in the stage tag array, we further examine the eight sub-block fields. Because the stage tag array entries and the stage area blocks are one-to-one corresponded, a hit/miss in the stage tag array guarantees the sub-block is in or not in the stage area. On the other hand, if we miss in the remap cache, we need to probe the off-chip remap table in the fast memory. The Remap bit of the target sub-block determines whether it is remapped to the location pointed by Pointer, or in the original place. Stage tag array hits have higher priority than remap cache/table entries. Based on the metadata outcome, we have the following five cases as Fig. 6.

Case 1: block in stage area, and sub-block hit. The data are already in the stage area, so we can directly perform read or write. For writes, we recompress the updated data, and if they exhibit a different CF, we remove and insert them as if they are a newly fetched range (see case 3 below).

Case 2: block in cache/flat area, and sub-block hit. We directly access the data in the fast memory. Note that this means data in the fast memory (originally, or after committed) are *not* further staged. One key difference from case 1 is that, the data layout cannot change now (Rule 4). If a write makes the sub-block data overflow its original CF and not fit in the block, we have to evict them to the slow memory. Because the remap entry format is sorted and dense (Rule 4), the whole block layout becomes invalid, unless the evicted sub-block is the last one. So we must evict the whole block. Fortunately, such evictions only happen in less than 1% of all memory accesses (Fig. 3), incurring minor performance loss.

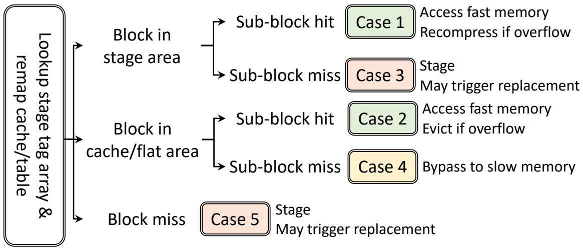


Fig. 6. Access flow in Baryon. Cases are elaborated in Section III-D.

Case 3: block in stage area, and sub-block miss. In this case, we fetch the sub-blocks and immediately return the demanded 64 B cacheline to the processor LLC. In the background, we compress the sub-blocks and try to append them into the stage area block space with other sub-blocks from the same data block, i.e., a stage operation. Fig. 5(c) shows this case: an access to B4 finds block B in the stage area but the sub-block misses, so we fetch B4 together with its neighbors. We always fetch the maximum contiguous and aligned range (Rule 2; B4-B7 here) that can be compressed into one physical sub-block space. In other words, compression enables a form of *slow-to-stage prefetching*. More specifically, at the first time when data from the slow memory are prefetched in the above way, we try all possible CFs and choose the maximum one. Section III-F introduces an optimization that keeps these fetched data in the compressed form when they are evicted back to the slow memory. As a result, the next time when the same data are fetched to the stage area, they are already grouped as the desired range for slow-to-stage prefetching, and do not need repetitive trials. If the physical block does not have an empty place, the fetch may trigger a replacement, which we discuss in detail shortly.

Case 4: block in cache/flat area, and sub-block miss. Because the committed block layout is finalized (Rule 4), we cannot add the sub-block to this block. Because some other sub-blocks in this target data block have already been remapped to this physical block in the fast memory, we cannot stage this sub-block to the stage area, otherwise violating Rule 3. Therefore, we directly access the data from the slow memory. This case is also rare (4.9% in Fig. 3), because the stage phase is able to capture the most frequently accessed footprint. Such bypassing may sometimes be beneficial, leading to more balanced fast/slow memory bandwidth usage [1], [15], [55].

Case 5: block miss in the fast memory. We fetch the data from the slow memory similar to case 3, and also try to stage it and its neighbors. The fetched data go to the physical block

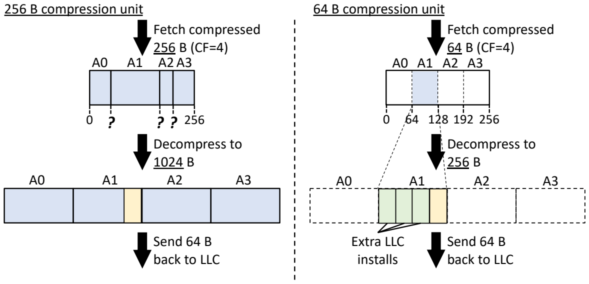


Fig. 7. Original vs. cacheline-aligned compression. Left: Without knowing the exact boundaries in the compressed data, all 256 B must be fetched. Right: By enforcing to compress into individual 64 B, fewer data are fetched.

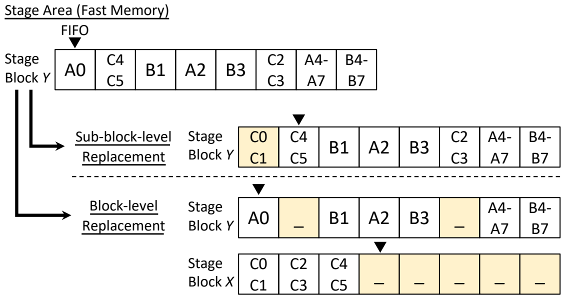


Fig. 8. Two-level (sub-block- and block-level) replacements in the stage area.

in the stage area if there is already one for its super-block, or require allocating a new physical block. This may also trigger replacements. In rare cases when there are multiple physical blocks staging this super-block, we randomly select one to append. This heuristic attempts to balance the block staging lifetime, and has no metadata cost.

E. Stage Area Fetch/Replacement/Commit Policies

Cacheline-aligned compression. Baryon uses a sub-block size of 256 B, larger than the 64 B cacheline and DDRx access granularity. As a result, at an LLC miss, we must transfer at least a compressed sub-block back to the memory controller, decompress it, and retrieve the demanded cacheline. The rest cachelines, which can be up to 1024 B with CF = 4, result in bandwidth waste and may pollute the LLC (Fig. 7 left).

Baryon uses *cacheline-aligned* compression to alleviate this issue [50]. In particular, we require that each individual 64-byte chunk in the compressed sub-block can be independently decompressed. This means that, while originally 256n-byte data are compressed into one sub-block to get a CF of n ($n = 1,2,4$), now each chunk of 64n-byte data (4 such chunks in n sub-blocks) must be individually compressed with a CF of n, as shown in Fig. 7. This is a stronger compression restriction, as typically smaller chunks are harder to compress than larger chunks. However, we find the actual CF loss (from 1.78 to 1.63) and the hit rate degradation (3%) are marginal, matching the observation in previous work [50]. There are enough compression opportunities even within each cacheline. Most hardware compression algorithms for caches/memories, including those we adopt, are designed to work sufficiently well at the cacheline granularity [4], [13], [47], [49], [78].

With cacheline-aligned compression, each data access only transfers a 64-byte compressed data chunk, compatible with DDRx. This piece of data may be decompressed into multiple cachelines (up to 4), which are all installed into the LLC to benefit from spatial locality, same as in previous work [50], [74]. Such *memory-to-LLC prefetching* is bandwidth-free and could increase the LLC hit rate by up to 5% for workloads with high spatial locality [74].

Two-level replacements. Staging newly fetched sub-blocks into the stage area may need to first evict other sub-blocks or blocks (cases 3 and 5 in Fig. 6). If there are already other sub-blocks from the same data block existing in the stage area, the incoming new sub-block *must* go to that physical block (Rule 3), and may trigger a sub-block-level replacement. Otherwise, the new sub-block, as the first fetched one from a data block, may choose either a sub-block-level replacement, or to allocate a new physical block space (a block-level replacement).

Baryon relies on a *two-level replacement* policy to efficiently handle these scenarios. The block-level replacements follow a standard LRU policy, using the `LRU` bits in the stage tag entry (Fig. 5(a)). For sub-block-level replacements, we use a 3-bit `FIFO` pointer to the next sub-block victim, so new data sub-blocks are appended sequentially, and wrap back if overflowed. A simple FIFO policy reduces the metadata size, and works sufficiently well because compression and sub-blocking make sub-block-level replacements much more infrequent than ordinary cache evictions.

For those scenarios that can use either level of replacement (i.e., case 5), we use the following heuristic. If the physical block suffering from a sub-block-level replacement is the LRU one, we prefer a sub-block-level replacement that only evicts a sub-block to the slow memory. As the top in Fig. 8 shows, C0-C1 replaces A0 in block Y. However, if the block is not LRU, we prefer a block-level eviction to the LRU block, which represents a less frequently/recently accessed super-block. The incoming sub-blocks are inserted into the new block space, e.g., C0-C1 in Fig. 8 bottom; additionally, the other sub-blocks in the same data block as the incoming ones, e.g., C2-C3 and C4-C5, are also moved to the new block to satisfy Rule 3. Now two physical blocks are used for this super-block. While this rearrangement seems costly, it is actually beneficial in the perspective of decreasing fragmentation, where the sub-blocks belonging to the same data block have a higher chance to be re-grouped together and even recompressed to a smaller size. In either case, the replacements are off the critical path.

Selective commits. A block-level replacement in the stage area could choose to commit the victim block, i.e., putting it into the fast memory, or directly evict the victim block to the slow memory. In the first case, to make room for the committed block, a block in the cache/flat area is selected for evicting to the slow memory in the same way as the baseline fast-to-slow eviction, which can use policies such as LRU, LFU, CLOCK, and even random [31], [43], [55], [62]. This is orthogonal to Baryon. We choose LRU for low-associative cases and FIFO for high-associative.

To decide whether to evict the stage area victim block or the cache/flat area LRU block to the slow memory, Baryon uses a *selective commit* policy based on a novel *stability-aware* heuristic cost model. Traditional policies like Hybrid2 [67] mainly consider the writeback traffic, where the two candidates are compared based on their numbers of dirty sub-blocks. In Baryon, we further incorporate the block layout stability, because committing an unstable block would inefficiently cause many sub-block misses and write overflows later in the fast memory. Following the idea in Fig. 4, we approximate the benefit of committing a block using the reduction of misses during the stage phase. Essentially, the miss rate at the beginning indicates the situation where blocks are not committed but accessed from the slow memory, and the miss rate at the end of the stage phase represents the expected miss rate after we commit the compressed block to the fast memory. Their difference reflects the saving from committing.

We add a 2 B `MissCnt` in each stage tag array entry to count the sub-block misses to this block (case 3 in Section III-D). We also add a same-size `MRUMissCnt` per each *set*, which counts the misses to the MRU position in this set, including block-level misses (case 5), and sub-block misses (case 3) to the current MRU block. All these counters age themselves by right shifting one bit every 10000 accesses to this set. When we need to decide whether to commit a block, its own `MissCnt` represents the recent miss count close to the end of its stage phase, while the `MRUMissCnt` divided by the associativity estimates the miss count of a just staged block. These are the two statistics needed by our policy. Therefore the commit benefit is given as

$$B = k \times \left(\frac{\texttt{MRUMissCnt}}{\texttt{assoc}} - \texttt{MissCnt}\right) + (\#\text{Dirty}_{\text{stage}} - \#\text{Dirty}_{\text{cache/flat}}) \quad (1)$$

The first term considers the layout stability as discussed above. The second term adds the write traffic cost [67], where #Dirty is the number of dirty sub-blocks that need writeback in the candidate block. For the flat area, all sub-blocks need to be swapped and so all are treated as dirty. Our policy uses a parameter k to measure the importance of the two factors. If $B \geq 0$, committing is more beneficial; otherwise the block should be directly put back to the slow memory.

Obviously the value of k is critical to the effectiveness of this policy. When $k = 0$, the policy only cares about the write traffic similar to Hybrid2 [67]; when $k = \infty$, it only considers the layout stability. In Section IV-D, we empirically find that a k value slightly larger than 1 (e.g., 4 as the default in Baryon) works well. This is because the writes are mostly off the critical path so we should put a heavier weight on the misses.

F. Additional Discussion and Optimizations

Supporting high associativities. Hybrid memories with higher associativities address conflict misses and exhibit higher fast memory hit rates [51], [66], [67]. However, cache-style inverted (device-to-physical) tag arrays cannot be used anymore because of impractical associative search, and forward (physical-to-device) remap tables become the only choice. Baryon uses such a remap table for the fast memory areas, thus easily supporting high associativities. The stage area, on the other hand, is independent and uses its own low-associative structure. Another issue of highly associative fast memory is that, during replacement, it is hard to know the original location of the victim only from the remap table, and an inverted table is needed with extra overheads [51], [67]. Baryon makes this orthogonal issue neither better nor worse.

Supporting the flat scheme. Baryon manages its fast memory similarly to previous work [38], [51], [62], [67]. However, for the flat scheme, if a fast memory block space is selected to hold a migrated data block, its original content must be *swapped* to the slow memory. This could raise issues in Baryon when we allow for sub-block-level fetches that may not leave a full empty block in the slow memory. For example, in Fig. 5, now assume Block Z is in the fast memory *flat* area into which we would commit a stage area Block Y. The original block Z data must be swapped to the slow memory. However, as in Fig. 5(c), none of the eight block spaces in the super-block is completely available; each has some sub-blocks that are not migrated, e.g., A1, B0, C0, etc. We cannot swap the original Block Z into any *single* one of them.

Baryon instead spreads the swapped block into the many *free individual sub-block spaces* in the super-block. Note that to have a full compressed Block Y to be committed, we must have migrated a sufficient number of sub-blocks from that super-block, which is typically larger than a block size due to further compression (we never stage fast memory data, so all sub-blocks should come from a slow memory super-block according to Rule 1). These available sub-block spaces are already accurately recorded in the remap entry after committed. For example, in Fig. 5(e), Block A has `Remap` bits `10101111`, meaning the sub-block spaces with `1` bits are available for swapped content. We thus choose the first eight sub-block spaces to hold the swapped block. This also works even if multiple compressed blocks are committed (e.g., Blocks X and Y in Fig. 5(d)); the victim of each can go to their disjoint sets of slow memory sub-block spaces.

When the chosen victim in the flat area is not an original fast block, but another committed compressed block, Baryon follows the widely used *slow swap* mechanism [55], [62] that requires data originally from the slow memory to only be evicted to their original locations, not any other slow memory space. This avoids chains of swapping that may go arbitrarily long and complicate metadata tracking. The cost is a three-way swap that moves more data. Following our example in Fig. 5(e), say now we want to commit Block X to the space of Block Z, but Z contains the committed data as shown in the figure. The original content of Z is now spread over the super-block Φ. (1) We first move the original content of Z to the sub-block spaces corresponding to the data in the to-be-committed new Block X. (2) Then we evict the previously committed data currently in Block Z to their original slow memory locations, which are just freed up in the last step. (3) Now Block X can use Block Z space. The final state is the same as if we only had committed Block X. So the Baryon metadata scheme can directly support it.

TABLE I
SYSTEM CONFIGURATIONS.

Cores L1I L1D L2 LLC	x86-64, 3.2 GHz, 16 cores 4-way, 32 kB per core, 64 B cachelines, LRU 8-way, 64 kB per core, 64 B cachelines, LRU 8-way, 1 MB per core, 9-cycle latency, LRU 16-way, 16 MB shared, 38-cycle latency, LRU
Stage tag array	8192 sets, 4-way, 5-cycle latency
Remap cache	256 sets, 8-way, 8 entries per line, 3-cycle latency
Compressor	FPC/BDI, 2 B/4 B/8 B segments, 5-cycle decompr.

Fast memory	DDR4-3200, 4 channels, 2 ranks, 16 banks;
	RCD-CAS-RP: 22-22-22;
	RD/WR: 5.0 pJ/bit, ACT/PRE: 535.8 pJ

Slow memory	NVM, 1333 MHz, 4 channels, 1 rank, 8 banks; read 76.92 ns, 14 pJ/bit; write 230.77 ns, 21 pJ/bit

Fast-to-slow writeback with compressed data. To further reduce slow memory bandwidth consumption, we can keep data compressed when they are written back from the fast memory to the slow memory. When doing such fast-to-slow writeback, we clear the `Remap` bits, but keep the `CF2`/`CF4` bits which indicate what data are compressed. Another benefit is that these `CF2`/`CF4` bits also act as slow-to-stage prefetching and compression hints, when the sub-blocks are fetched again in the future. We find this small optimization reduces bandwidth by 7.2% and improves performance by 3.1%.

IV. EVALUATION

A. Experimental Setup

Simulated configurations. We use zsim [56], a Pin-based simulator, to evaluate Baryon on a 16-core system summarized in Table I. We use DDR4 as the fast memory and an NVM as the slow memory, with a capacity ratio of 1:8, i.e., 4 GB and 32 GB, respectively. They are modeled from commercial datasheets as well as previous literature and open-source implementations [11], [35], [67], [68], [77]. We extend zsim to support both cache and flat modes. In the flat mode, the blocks are initially placed in the fast memory until the space is used up. We use CACTI [8] to model the additional SRAM structures, including the stage tag array and the remap cache. We conservatively use a 5-cycle decompression latency on the critical path, though more aggressive designs showed decompression down to a single cycle [4], [13], [49].

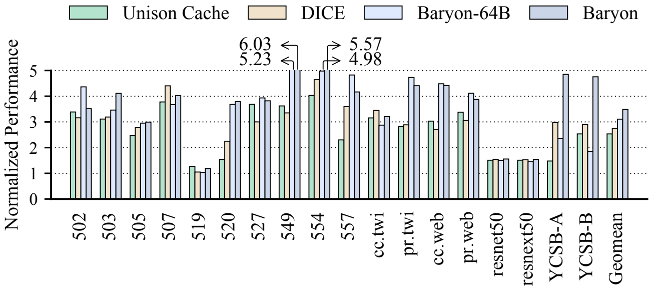


Fig. 9. Performance comparison between Baryon and the baselines in the low-associative cache mode. Normalized to Simple (not shown).

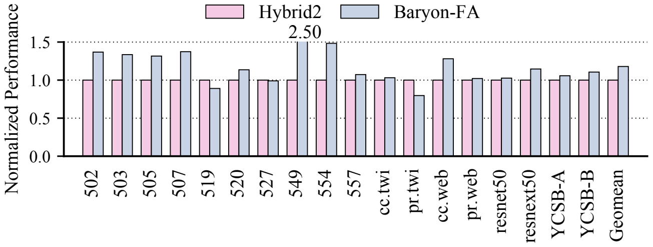


Fig. 10. Performance comparison between Baryon and the baseline in the fully-associative flat mode. Normalized to Hybrid2.

Workloads. We use benchmarks from a subset of SPEC CPU2017 [48] whose data footprints are larger than the fast memory capacity, graph algorithms (`pagerank` and `cc`) from GAP [9] on real-world graphs (`twitter` and `web-sk-2005`), neural network inference (`resnet50` and `resnext50`) using Intel OneDNN [29], and memcached key-value store [21] with YCSB [18]. SPEC workloads are executed in the rate mode with 16 copies; other workloads use 16 threads each. For SPEC workloads, we fast forward the first 5 to 50 billion instructions until they are fully initialized, and simulate the next 5 billion instructions. They use 5.8 GB (`557.xz_r`) to 13.4 GB (`549.fotonik3d_r`) memory space. We simulate iterations 2 to 5 of each GAP workload, with up to 34.6 GB footprints. The neural network inference tasks run four batches of size 64 each, with 14.6 GB to 18.6 GB memory usage. For memcached with YCSB, we generate 30 million records of 1 kB size (in total 30 GB) and run 30 million queries of type A (50%/50% read/write) and B (95%/5% read/write). Both the loading and transactional phases are simulated.

Baselines. For the cache mode, we mainly compare with **Unison Cache** [31] and **DICE** [74]. Unison Cache uses 2 kB blocks with 64 B sub-blocking, but does not support compression. We enlarge its on-chip SRAM (way predictor and footprint history table) proportionally to match our fast memory size. DICE is a state-of-the-art compressed DRAM cache using 64 B blocks without sub-blocking. We assume DICE has the same 5-cycle decompression latency as Baryon. We set Unison Cache and Baryon to 4-way set-associative, while DICE is limited to direct-mapped but we optimistically use a perfect way predictor. We also include a **Simple** DRAM cache with neither compression nor sub-blocking. It uses 2 kB blocks and an associativity of 4 fast memory ways.

For the flat mode, we choose **Hybrid2** [67] with 2 kB block size as the state-of-the-art baseline. Same as Hybrid2, we make Baryon fully-associative (named **Baryon-FA**) for fair comparisons. Both designs use 256 B sub-blocking.

B. Overall Comparison

Fig. 9 shows the performance improvements of Baryon in the low-associative cache mode over the DRAM cache baselines. All performance numbers are normalized to Simple. Baryon outperforms Unison Cache and DICE by 1.38× and 1.27× on average, and up to 2.46× and 1.68×. To isolate the impact of different sub-block granularities (256 B in Baryon

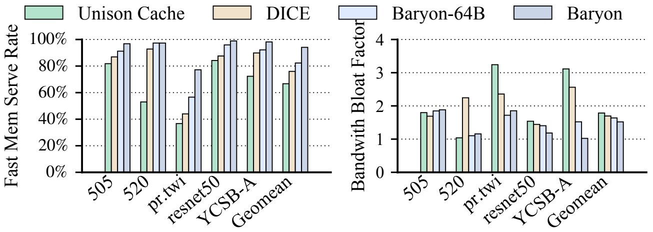


Fig. 11. Performance analysis. Left: the percentage of memory accesses served by the fast memory (higher is better). Right: the ratio between total fast memory traffic (including migration/swapping with the slow memory) and useful data traffic to the LLC (bandwidth bloat factor [15], lower is better).

vs. 64 B in the baselines), we also include Baryon-64B which uses 64 B sub-blocking. Most of the speedup of Baryon is from its compression and sub-blocking features (1.27× and 1.13× from Unison Cache and DICE to Baryon-64B), while a more optimized granularity provides another 12.2%. Note that the optimal sub-block size for each workload depends on its locality behavior. Some workloads, e.g., 557.xz_r, exhibit low spatial locality and prefer a smaller sub-block size like Baryon-64B. On the other hand, using a larger sub-block size always helps reduce the overall metadata storage cost.

In general, Baryon delivers higher benefits on 1) workloads with large datasets, e.g., pr.twitter and YCSB-A, because there is more stress on fast memory capacity and slow memory bandwidth; 2) workloads with highly compressible data, e.g., 549.fotonik3d_r, which has a high average CF of 2.42. Baryon is only slower than Unison Cache on 519.lbm_r. This is because this workload is highly write-intensive and there is very limited compression opportunity with a nearly 1.0 CF. Therefore, compression only adds overheads in DICE and Baryon, making them slower than Unison Cache.

Fig. 10 shows the speedup of Baryon-FA over Hybrid2, on average 1.18× and up to 2.50×, for the fully-associative flat mode. The trend is similar to the cache mode.

We also compare the memory system energy of Baryon and baselines. The energy savings correlate well with the performance improvements, where on average Baryon reduces energy by 31.9% over Unison Cache and 13.0% over DICE, and Baryon-FA reduces energy by 14.5% over Hybrid2. Most energy savings are due to lower traffic to the slow memory.

C. Performance Analysis

We now make an in-depth analysis of the performance gains enabled by Baryon, using representative workloads from each domain under the low-associative cache mode as examples. In each figure we also include the geometric mean results of *all* workloads. From Fig. 11 left we can see that, most of the speedups in Baryon are from the increased fast memory serve rates (the percentages of memory accesses served by the fast memory). For example, in pr.twi, Baryon fits most of its working set in the fast memory and achieves a serve rate of 77%. This is in sharp contrast to 37% in Unison Cache and 44% in DICE. With more efficient sub-blocking and compression, we have placed more data in the fast memory to exhibit lower latency, and also have reduced the amount of slow memory traffic due to misses. This explains the bandwidth bloat factor decreasing of pr.twi from 3.2 in Unison Cache and 2.4 in DICE to 1.8 in Baryon. In other words, Baryon improves the utilization of both memory capacity and bandwidth. For 520.omnetpp_r, on the other hand, compared to DICE, the fast memory serve rate is already high and does not increase much, but Baryon saves 49% fast memory bandwidth usage as in Fig. 11 right, due to reduced evictions and writebacks. This leads to a 1.64× speedup.

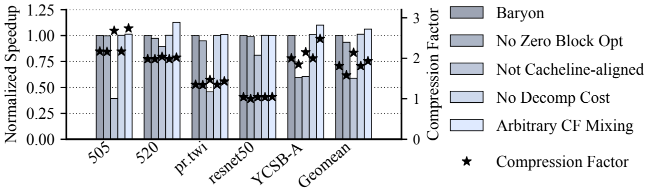


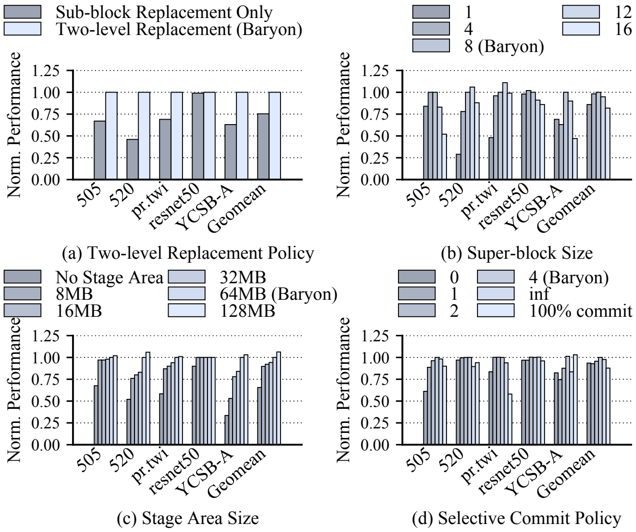
Fig. 12. Impacts of different compression schemes on performance and compression factors in Baryon.

Fig. 12 justifies our specific choices of compression schemes in Baryon. First, the zero block support with the Z bit (Fig. 5) is a simple optimization without any metadata size overhead. It improves the CF from 1.85 to 2.00 and leads to 8% better performance in YCSB-A. Second, enforcing cacheline-aligned compression (Section III-E) decreases the CF noticeably in some workloads like 505.mcf_r but only slightly in others like pr.twi and resnet50. However, without cacheline-aligned compression, the performance always drops significantly (11% to 61%), due to wasted data fetch and LLC pollution. Third, the 5-cycle decompression latency in Baryon, though it is already a conservative assumption compared to other work [4], [13], [49], has a negligible impact of less than 1% on the overall performance. Fourth, the metadata format in Baryon (Fig. 5) only allows adjacent blocks being compressed with the same CF. This restriction greatly saves metadata size, but decreases the achievable CF. We show that empirically the lost performance is small, only up to 12% in 520.omnetpp_r.

D. Design Parameter Exploration

Our two-level replacement policy for the stage area, and particularly the choice between sub-block-level and block-level replacements, is important, as it not only reduces data fragmentation, but also allows multiple fast block spaces to keep critical data from one hot super-block. Fig. 13(a) shows, if we disable block-level replacements and only evict sub-blocks within a block (so that only one block space can be used for data from a super-block), the performance would significantly degrade by about 25%.

Fig. 13(b) shows the impact of super-block sizes. Baryon already allows multiple physical block spaces to buffer data from one super-block, so larger super-block sizes do not help much. Furthermore, a large super-block size sometimes decreases performance, e.g., 505.mcf_r loses 50% performance with a very large super-block size, due to indexing many



(a) Two-level Replacement Policy

(b) Super-block Size

(c) Stage Area Size

(d) Selective Commit Policy

Fig. 13. Impacts of (a) two-level replacement policy, (b) super-block size (in blocks), (c) stage area size, and (d) selective commit policy parameter. All performance numbers are normalized to the default Baryon configuration.

adjacent blocks within one set which increases conflict misses. Grouping 8 blocks as a super-block is sufficient in Baryon.

The stage area size also affects performance as Fig. 13(c) shows. A larger stage area can more effectively stabilize data layouts and result in better performance, but also requires more metadata (on-chip stage tag array). We see that even a small 8 MB stage area performs well enough for some workloads like `505.mcf_r`, but a moderately larger 64 MB size offers 24% speedup in `520.omnetpp_r`. These results again validate our key observation in Section III-B, i.e., a relatively small stage area is quite effective to stabilize compressed and sub-blocked data layouts. Note that if we do not use any stage area, then nearly every data insertion/replacement needs to re-sort the data block layout, as required by the compact and sorted remap entry format. This incurs significant performance degradation of 34.5% on average.

Fig. 13(d) evaluates the commit policy. The selective commit policy parameter k (Section III-E) affects the tradeoff between layout stability and dirty data write cost. Choosing a proper k outperforms either policy that only considers one of the two factors ($k = 0$ or $k = \text{inf}$). Nevertheless, the policy is insensitive to the exact k value, as 1, 2, and 4 perform similarly. Finally, committing selectively is better than committing all blocks regardless of the stability and the write overhead.

V. Related Work

To our best knowledge, Baryon is the first architecture to support both *memory compression* and *sub-blocking* on *hybrid memory systems*. There are large bodies of previous proposals regarding each of the three domains.

Hybrid memory combines two or more heterogeneous memory technologies to exploit their latency, bandwidth, and capacity advantages. Early designs [33], [46], [53] used OS page tables for data migration and thus suffered from long migration intervals and coarse granularities. Later hardware-based architectures [14], [36], [51], [55], [62], [66], [67] were able to realize high associativity, fine granularity, and/or high fast memory utilization. These designs either used the fast memory as an additional level of cache, or as part of the main memory visible to software. Some recent proposals supported both schemes. Hybrid2 provisioned a fixed cache capacity [67]. Chameleon sought help from the OS for memory allocation [38]. Stealth-Persist also mixed the two schemes, but focused on persistency, not fast memory utilization [3].

Sub-blocking is a well-known optimization for caches and hybrid memories. It was initially used in sector caches [54]. Footprint Cache introduced it to DRAM caches [32]. SILC-FM used it for flat schemes in hybrid memories [55]. Micro-sector cache further allowed sub-blocks from multiple blocks to pack together, with another associativity dimension, in order to save capacity as well as bandwidth [12]. But it had significant metadata tag overheads. We use specific optimizations to reduce tag space, and also support more complex data compression.

Memory compression brings both capacity and bandwidth gains. There exist a wide range of compression algorithms, offering wide tradeoffs for compression ratio, latency, and hardware design cost [4], [6], [13], [34], [47], [49], [50], [78]. These algorithms are orthogonal to our design. Regarding metadata management, super-blocks have been used in compressed caches to group several neighboring blocks for flexible compaction [57]–[59]. In contrast, other designs used shortened tags to compact blocks from arbitrary addresses not necessarily nearby [26], [27], [73]. We follow the first category as it is simpler and corresponds well with current hybrid memory address remapping. Some other papers also aimed at memory compression under hybrid memory systems [7], [19], but mainly to reduce traffic to the slow memory, especially writes to NVMs to avoid wear-out issues. We instead compress data in the fast memory, to optimize for higher fast memory utilization and hence performance.

VI. Conclusions

We propose Baryon, a novel hybrid memory system architecture that supports fine-grained data compression and sub-blocking with moderate metadata overheads and management complexity. Baryon is OS and software transparent, and can work with either cache or flat scheme of hybrid memory systems. The benefits of Baryon are mostly enabled by a dual-format metadata scheme and the stage area which effectively manages the complex, irregular, and fast-changing data layouts after compressed and sub-blocked. Baryon significantly improves performance on memory-intensive multi-program and multi-threaded workloads.

Acknowledgment

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262) and Huawei. Mingyu Gao is the corresponding author.

References

[1] N. Agarwal, D. W. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2015, pp. 354–365.

[2] N. Agarwal and T. F. Wenisch, "Thermostat: Application-Transparent Page Management for Two-tiered Main Memory," in *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017, pp. 631–644.

[3] M. Al-Wadi, V. R. Kommareddy, C. Hughes, S. D. Hammond, and A. Awad, "Stealth-Persist: Architectural Support for Persistent Applications in Hybrid Memory Systems," in *27th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 139–152.

[4] A. Alameldeen and D. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2004.

[5] A. R. Alameldeen and D. A. Wood, "Adaptive Cache Compression for High-Performance Processors," in *31st International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2004, pp. 212–223.

[6] A. Arelakis, F. Dahlgren, and P. Stenström, "HyComp: A Hybrid Cache Compression Method for Selection of Data-Type-Specific Compression Methods," in *48th International Symposium on Microarchitecture (MICRO)*. ACM, 2015, pp. 38–49.

[7] S. Baek, H. G. Lee, C. Nicopoulos, and J. Kim, "A Dual-Phase Compression Mechanism for Hybrid DRAM/PCM Main Memory Architectures," in *Great Lakes Symposium on VLSI 2012 (GLSVLSI)*. ACM, 2012, pp. 345–350.

[8] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[9] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, Aug 2015.

[10] C. Cagli, "Characterization and Modelling of Electrode Impact in HfO2-Based RRAM," in *Workshop on Innovative Memory Technologies*, 2012.

[11] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 73–84.

[12] M. Chaudhuri, M. Agrawal, J. Gaur, and S. Subramoney, "Micro-Sector Cache: Improving Space Utilization in Sectored DRAM Caches," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 1, pp. 7:1–7:29, 2017.

[13] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[14] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 1–12.

[15] C. Chou, A. Jaleel, and M. K. Qureshi, "BEAR: Techniques for Mitigating Bandwidth Bloat in Gigascale DRAM Caches," in *42nd International Symposium on Computer Architecture (ISCA)*. ACM, 2015, pp. 198–210.

[16] C. Chou, A. Jaleel, and M. K. Qureshi, "BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM," in *3rd Internatinoal Symposium on Memory Systems (MEMSYS)*. ACM, 2017, pp. 268–280.

[17] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic Main Memory Compression," in *51st International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2018, pp. 546–558.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *1st ACM Symposium on Cloud Computing (SoCC)*. ACM, 2010, pp. 143–154.

[19] Y. Du, M. Zhou, B. R. Childers, R. G. Melhem, and D. Mossé, "Delta-Compressed Caching for Overcoming the Write Bandwidth Limitation of Hybrid Main Memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 55:1–55:20, 2013.

[20] M. Ekman and P. Stenström, "A Robust Main-Memory Compression Scheme," in *32st International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2005, pp. 74–85.

[21] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[22] A. Ghasemazar, P. J. Nair, and M. Lis, "Thesaurus: Efficient Cache Compression via Dynamic Clustering," in *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020, pp. 527–540.

[23] N. D. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 38–50.

[24] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quéma, R. Rodrigues, P. Romano, and V. Vlassov, "Bandwidth-Aware Page Placement in NUMA," in *34th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 546–556.

[25] F. T. Hady, A. P. Foong, B. Veal, and D. Williams, "Platform Storage Performance With 3D XPoint Technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[26] S. Hong, B. Abali, A. Buyuktosunoglu, M. B. Healy, and P. J. Nair, "Touché: Towards Ideal and Efficient Cache Compression By Mitigating Tag Area Overheads," in *52nd International Symposium on Microarchitecture (MICRO)*. ACM, 2019, pp. 453–465.

[27] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K. Kim, and M. B. Healy, "Attaché: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads," in *51st International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2018, pp. 326–338.

[28] C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2014, pp. 51–60.

[29] Intel, "oneDNN Documentation," https://oneapi-src.github.io/oneDNN/.

[30] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient Footprint Caching for Tagless DRAM Caches," in *22nd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2016, pp. 237–248.

[31] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 25–37.

[32] 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 *40th International Symposium on Computer Architecture (ISCA)*. ACM, 2013, pp. 404–415.

[33] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter," in *44th International Symposium on Computer Architecture (ISCA)*. ACM, 2017, pp. 521–534.

[34] J. Kim, M. B. Sullivan, E. Choukse, and M. Erez, "Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures," in *43rd International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2016, pp. 329–340.

[35] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letter*, vol. 15, no. 1, pp. 45–49, 2016.

[36] D. Knyaginin, V. Papaefstathiou, and P. Stenström, "ProFess: A Probabilistic Hybrid Main Memory Management Framework for High Performance and Fairness," in *24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2018, pp. 143–155.

[37] A. Kokolis, D. Skarlatos, and J. Torrellas, "PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems," in *25th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 596–608.

[38] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir, "CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System," in *49th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2018, pp. 533–545.

[39] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alterna-

tive," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2013, pp. 256–267.

[40] A. Labrinidis and H. V. Jagadish, "Challenges and Opportunities with Big Data," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 2032–2033, 2012.

[41] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[42] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *36th International Symposium on Computer Architecture (ISCA)*. ACM, 2009, pp. 2–13.

[43] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2187–2200, 2014.

[44] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A Fully Associative, Tagless DRAM Cache," in *42nd International Symposium on Computer Architecture (ISCA)*. ACM, 2015, pp. 211–222.

[45] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *44th International Symposium on Microarchitecture (MICRO)*. ACM, 2011, pp. 454–464.

[46] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories," in *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2015, pp. 126–136.

[47] B. Panda and A. Seznec, "Dictionary Sharing: An Efficient Cache Compression Scheme for Compressed Caches," in *49th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2016, pp. 1:1–1:12.

[48] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon?" in *24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2018, pp. 271–282.

[49] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," in *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2012, pp. 377–388.

[50] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *46th International Symposium on Microarchitecture (MICRO)*. ACM, 2013, pp. 172–184.

[51] A. Prodromou, M. R. Meswani, N. Jayasena, G. H. Loh, and D. M. Tullsen, "MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 433–444.

[52] 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 *45th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2012, pp. 235–246.

[53] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *25th International Conference on Supercomputing (ICS)*. ACM, 2011, pp. 85–95.

[54] J. B. Rothman and A. J. Smith, "Sector Cache Design and Performance," in *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society, 2000, pp. 124–133.

[55] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization," in *23rd International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2017, pp. 349–360.

[56] D. Sánchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *40th International Symposium on Computer Architecture (ISCA)*. ACM, 2013, pp. 475–486.

[57] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed Compressed Caches," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 331–342.

[58] S. Sardashti, A. Seznec, and D. A. Wood, "Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 3, pp. 27:1–27:25, 2016.

[59] S. Sardashti and D. A. Wood, "Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization," in *46th International Symposium on Microarchitecture (MICRO)*. ACM, 2013, pp. 62–73.

[60] S. Sardashti and D. A. Wood, "Could Compression Be of General Use? Evaluating Memory Compression across Domains," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 44:1–44:24, 2017.

[61] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring Unconventional Benefits from Memory Compression," in *20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2014, pp. 638–649.

[62] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *47th International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 13–24.

[63] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-Level Cache Deduplication," in *28th International Conference on Supercomputing (ICS)*. ACM, 2014, pp. 53–62.

[64] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–286, 2001.

[65] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, pp. 65:1–65:23, 2019.

[66] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "LLC-Guided Data Migration in Hybrid Memory Systems," in *33rd IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 932–942.

[67] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining Caching and Migration in Hybrid Memory Systems," in *26th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 649–662.

[68] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and Modeling Non-Volatile Memory Systems," in *53rd International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 496–508.

[69] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting Program Semantics to Place Data in Hybrid Memory," in *24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2015, pp. 163–173.

[70] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John, "Invited Paper for the Hot Workloads Special Session Hot Regions in SPEC CPU2017," in *International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 2018, pp. 71–77.

[71] Z. Yan, D. Lustig, D. W. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 331–345.

[72] V. Young, C. Chou, A. Jaleel, and M. K. Qureshi, "ACCORD: Enabling Associativity for Gigascale DRAM Caches by Coordinating Way-Install and Way-Prediction," in *45th International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2018, pp. 328–339.

[73] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling Transparent Memory-Compression for Commodity Memory Systems," in *25th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 570–581.

[74] V. Young, P. J. Nair, and M. K. Qureshi, "DICE: Compressing DRAM Caches for Bandwidth and Capacity," in *44th International Symposium on Computer Architecture (ISCA)*. ACM, 2017, pp. 627–638.

[75] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation," in *50th International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 1–14.

[76] W. Zhao, E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle, "Macro-Model of Spin-Transfer Torque Based Magnetic Tunnel Junction Device for Hybrid Magnetic-CMOS Design," in *2006 IEEE International Behavioral Modeling and Simulation Workshop*. IEEE, 2006, pp. 40–43.

[77] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology,” in *36th International Symposium on Computer Architecture (ISCA)*. ACM, 2009, pp. 14–23.

[78] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.