Puzzle Memory: A Multifractional Partitioned Heterogeneous Memory Scheme

Jee Ho Ryoo*
Architecture and Technology Group
ARM Inc.
San Jose, CA, USA
jeeho.ryoo@arm.com

Shuang Song
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, USA
songshuang1990@utexas.edu

Lizy K. John
Electrical and Computer Engineering)
The University of Texas at Austin
Austin, TX, USA
ljohn@ece.utexas.edu

***Abstract*— As current main memory technology scaling is coming close to an end due to its physical limitations, many emerging memory technologies are coming to the market to fill the scaling gap. Future memory systems will require a heterogeneous memory architecture where one technology acts as a low latency memory whereas the other acts as a high capacity memory. This will allow the future main memory system to continue to scale in terms of capacity, yet have similar or slightly better latency than today's DRAM technology. Prior work on data management in heterogeneous memory has optimized one or a maximum of two components in the computing stack. However, different components are good at different tasks in data management, so in the era of heterogeneous memory, it is inevitable that cooperative multi-component data management will be adopted in future systems. We propose a heterogeneous memory layout where two memories are laid out asymmetrically. The operating system is aware of this layout and places pages with different locality characteristics in different regions of memory. Finally, a custom hardware performs the data remapping to optimize the data placement at finer granularity than what is visible to the operating system. In the end, we show that our multi-component cooperative data management scheme can improve the overall system performance by up to 40%.**

I. Introduction

The DRAM scaling rate has recently slowed down and is expected to halt as the DRAM technology node gets very close to the physical limits [1]. Fortunately, memory/processor vendors and researchers have proactively looked for many emerging memory technologies. Therefore, future systems will need multiple types of memories where one will act as high capacity memory whereas the other will act as low latency memory. However, since this memory is rather small, the application memory footprint will not to fit in this memory. Thus, it is inevitable that a large chunk of data will be present in high capacity memory and migrated as needed.

Our work leverages memory technology, processor hardware, and Operating System (OS). Prior work [2]–[6] has optimized only one of the aforementioned components at a time, and thus, was able to achieve suboptimal performance improvement. For example, prior hardware optimization [3] migrated data between two memories at the cache line granularity (e.g., 64B in x86-64). While this technique incurs low overheads, the mapping of data is fixed at design time and the OS doesn't optimize the data placement as it is oblivious to memory heterogeneity. Furthermore, OS data management approaches [2] are good at optimally placing larger chunks of data. Yet, since the minimum OS manageable memory size is quite large (e.g., 4KB in x86-64 and 8KB in SPARC), a lot of precious low latency memory space is left untouched as many of these cache lines are not accessed during runtime. Finally, prior work has assumed symmetric memory layout where data is laid out identically within one memory technology. Our asymmetric layout was not needed in today's homogeneous DRAM-only memory system as there is effectively one memory with the same characteristics. Laying out data symmetrically when different memory rows have different access characteristics leads to suboptimal performance and wastage of precious low latency memory capacity.

Our technique employs a bottom up approach where we redesign the memory layout so that different memories take different capacity ratios for different regions of memory. For example, the low latency to high capacity memory capacity ratio is 1:3 in some regions, so 1/4 of the data in this memory region can be placed in low latency memory. What data to place in what a region is not the decision that the low-level memory controller has to make, so we leave this work to the next component in the computing stack. Although our unconventional layout is not common, it is essential to have such layout for the next component. Now, the OS comes in and places data accordingly. The OS is modified so that it is aware of the asymmetric memory layout as well as different memory ratios in different regions of the physical address space. Although the OS places the data well, there is still room for improvement by hardware. The memory visibility of the OS is limited to the minimum OS page size, which is considerably larger than that of hardware. Memory subsystems in most processors work at the cache line granularity, which is commonly at 64B. Therefore, the fine-tuning of 64B block placement within the OS page is done by hardware in our scheme. Our work outperforms prior schemes where they only optimized data placement at one component in the computing stack, which has limited visibility as well as abilities.

*The author was a graduate student at the University of Texas at Austin when this work was done.

II. Background

In this section, we briefly discuss challenges associated with prior data management schemes in each component of the computing stack. Throughout the paper, we refer to the fast, capacity constrained memory as Near Memory (NM) and to the slow, large capacity memory as Far Memory (FM) since NM is physically located closer to the cores.

A. Challenges in Memory Layout

Heterogeneous memory systems are composed of two or more discrete memories, yet emerging memory technologies do not have specialized controllers that are optimized for the heterogeneous system. They are managed by discrete controllers where the data is laid out as in today's memory. These memories rely on higher levels in the computing stack to place data. For example, NM can be used as a cache where the data placement is calculated using the part of the address. In addition, some software schemes [2] have the OS to place data in NM. In either scheme, the data layout and the controller design is the same as in the conventional memory of today's computers.

B. Challenges in Hardware Data Management

A hardware component in the computing stack manages data at either small or large size blocks between two memories upon a request. The management granularity is smaller than OS page sizes as hardware is superior at managing data at small granularity. In hardware management schemes [3], each data block in NM has a remap table entry, which identifies the location of a requested block either in NM or FM. Different schemes [5] use different data management granularities from a small block (64B) to a large block (2KB) in efforts to carefully balance the tradeoff between bandwidth usage and metadata overheads.

POM [5] and CAMEO [3] are two state-of-the-art schemes that use a large and small block size respectively. POM uses large blocks (2KB) to minimize the metadata overheads while limiting the data migration to NM by calculating the benefits and cost analysis. POM requires a counter for a page to reach a threshold until the migration occurs. Also, although only a subset of a 2KB page is desired, it has to fetch the entire 2KB, which wastes significant bandwidth in low spatial locality workloads. Unlike POM, CAMEO adopts a small block size. Since the migration bandwidth consumption is low, it allows small blocks to swap from FM upon a request. CAMEO manages data at a small block granularity, so each block must have an accompanying remap table entry. The high metadata storage overheads due to having a remap table with each block is a drawback. Therefore, the metadata is stored within the same row as data in NM that has a large capacity, not in dedicated SRAM.

C. Challenges in Software Data Management

An optimization done in the software component is focused on using the OS to place pages in NM intelligently. Two state-of-the-art schemes are epoch based [2] and on-demand [7] approaches. In an epoch based scheme, the OS explicitly manages the NM capacity as a special region of memory. In this subsection, pages refer to OS pages, which typically are 4KB in x86-64. The software scheme relies on hot page detection to achieve performance improvement as there is no additional hardware to perform sophisticated operations such as dynamic remapping. The page migration occurs at a large interval, and the interval is referred to an epoch. At each epoch boundary, the OS sweeps through the PTEs to select those pages that are marked, and the bulk page migration occurs between NM and FM. The OS schemes move pages into NM and the PTE is updated accordingly. Upon an access, the physical address translated by TLBs is directly used to access the data in NM. Unlike hardware data management schemes, it does not have additional hardware structures like the remap table as the changes are purely done in the OS domain. The epoch based scheme is slow to changes in the hot working set. Furthermore, the working set coverage by NM is fixed during an epoch as no data migration occurs between NM and FM except at epoch boundaries. The on-demand scheme frequently incurs high software related overheads because any access to a page in large capacity FM involves the OS. These overheads include a context switch, TLB shootdown, and PTE update in addition to physical data transfer costs [8].

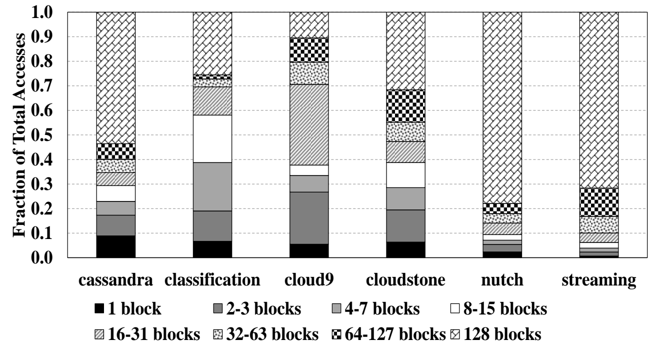

Fig. 1: Spatial Locality of Applications (Access Density)

III. Motivation

In this section, we show how applications access memory and what the requirements of heterogeneous memory systems are to optimize performance based on our observations.

A. Spatial Locality

Spatial locality of an application is a good indicator on how densely accesses are made in a physical address range. If the locality is high, this means nearby addresses are accessed, and in this case, a large data management granularity is preferred. If extra contiguous data is brought into NM, then it is likely to be used in the future in high locality cases. If the spatial locality is low, then the extra data brought in is wasteful since it merely occupies the space in NM and wastes migration bandwidth. Therefore, it is important to understand spatial locality characteristics of an application in order to design an effective memory system.

Figure 1 shows the memory access density of various popular cloud applications selected from CloudSuite 2.0 [9].

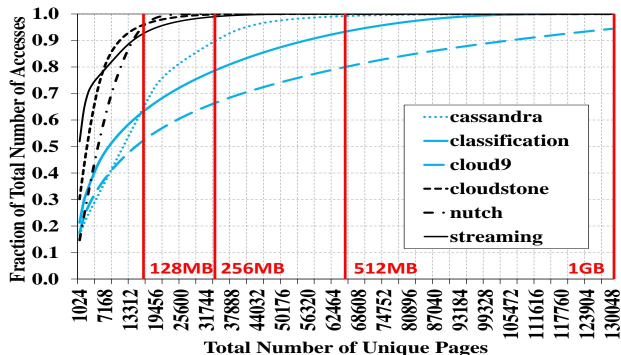

Fig. 2: Fraction of Accesses Made Up to X Pages (Cumulative)

A block in the figure represents a 64B data chunk (cache line size), and the number of unique blocks accessed within the 8KB data segment is counted. For example, 1 block in the figure means only one 64B data chunk is touched within an 8KB region. This indicates a low locality page as only 1 block out of 128 blocks is demanded. An access density is defined as the degree of 64B blocks touched within the 8K page. The access density is the highest if all unique 64B blocks within an 8KB data segment are demanded. On the other hand, if only one 64B block is demanded within an 8KB page, then the access density is lowest. The Y-axis is calculated using the total unique 8K pages accessed during the application runtime, and the fraction of those unique pages whose access density falls under each category.

As expected, different applications show different memory access characteristics. Therefore, fixing the migration granularity to one size statically as in the case of hardware managed schemes is not ideal as different applications access memory differently. For example, *classification* shows that a majority of pages have only 1 block usage within 8KB. In this workload, it is wasteful to use any migration granularity larger than 1 block. On the other hand, *nutch* shows an opposite behavior where a majority of pages have all of its blocks demanded. Other measured benchmarks fall between these two benchmarks.

Furthermore, when a few blocks are accessed within a page, they are not necessarily contiguous. This means that there are many blocks between demanded blocks that are not accessed. Such issues are called fragmentation where there are unaccessed blocks between demanded blocks. The pages with fragmentation cannot achieve optimal performance in previously proposed schemes. Simply varying the data management granularity cannot solve issues with fragmentation because the undemanded blocks cannot be filtered out. Although there are prior hardware works that can filter such undemanded blocks, the filtering only saves bandwidth. The precious NM capacity is still reserved for those undemanded blocks and cannot be used for any other blocks or pages.

SUMMARY: The spatial locality characteristics differ across applications, and thus, different data management granularities are preferred. In addition, fragmentation issues must be addressed to achieve the optimal bandwidth and performance.

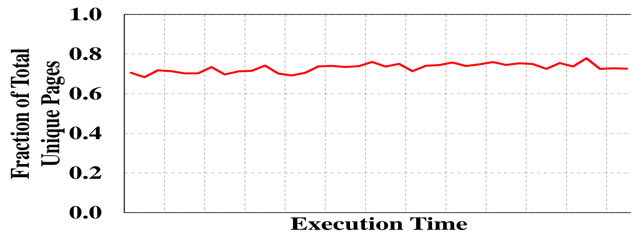

Fig. 3: Fraction of Total Pages Whose Patterns Repeat

B. Temporal Locality

Temporal locality of an application represents how frequently pages are re-accessed. Previously described spatial locality does not consider the hotness of pages. In other words, whether a page is accessed once or thousand times is not accounted in Figure 1. However, heterogeneous memory systems are composed of memory technologies whose access latencies are an order of magnitude higher than SRAM latency. Therefore, naively migrating any page will result in degraded performance as the data migration costs are high. In such a scenario, only pages with high temporal locality have to be involved in data migration between NM and FM. Pages with low locality are less likely to be accessed when migrated to NM, so even though it benefits from NM's lower access latency, the benefits are not enough to offset the high migration costs.

Figure 2 shows the temporal locality results of *nutch* (other workloads are excluded not to clutter the plot). The x-axis shows the total number of unique pages accessed during the application runtime. The y-axis shows the total cumulative fraction of accesses made. The y-axis is calculated as ① pages are sorted based on the number of accesses made to each page ② a cumulative sum of pages are computed ③ the cumulative sum of each sorted page is divided by the total number of accesses. From Figure 2, it is apparent that a small subset of pages is responsible for a large fraction of total accesses. Although different applications show different slopes on the left side of the graph, indisputably all applications have a very steep slope since small pages are heavily accessed while most other pages have a very small access counts. Consequently, we call those pages whose access counts are high as hot pages. Since not all pages can fit in capacity constrained NM, those hot pages will provide the most system performance benefits when placed in NM. Thus, a heterogeneous management scheme must be able to detect these hot pages.

SUMMARY: The temporal locality characteristics show that only a subset of pages within the entire memory footprint needs to be involved in migration as these pages account for a large fraction of total memory accesses.

IV. Implementation

With insights gained from motivational data in Section III, we propose a heterogeneous memory management scheme that spans across multiple components in the computing stack. We begin from the memory layout design and build up to the OS component.

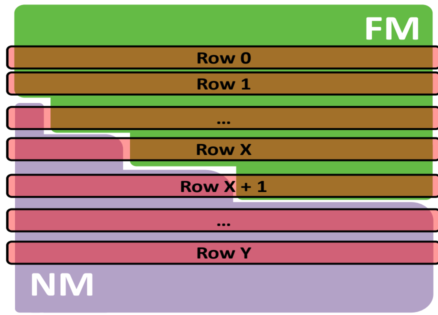

Fig. 4: Logical FM and NM Layout

A. Memory Layout

Pages in an application have different access densities as a different number of 64B blocks within an 8KB are accessed. The memory layout can take this into account and be customized where different logical memory rows have a different portion of NM. Only demanded blocks need to occupy the precious capacity in NM. Figure 4 shows the logical layout of our proposed heterogeneous memory system. In such a system, each row has a different fraction of NM and FM. For example, the memory row marked as "Row 0" has the entire row composed of FM, while "Row X" has a small fraction of the row given the faster NM capacity. Such rows may be useful for pages that only a small fraction of them is demanded by the application. On the other hand, "Row Y" is entirely made of fast NM, so it is an ideal location for pages most of whose 64B blocks are demanded. Such unconventional memory layout allows us to maximize the space usage in the capacity constrained NM. As we will show in the next section, we will only allow demanded blocks to occupy the NM space.

Figure 4 is a logical view of the data layout, yet the actual implementation is not overly complicated. In our work, we assume that we have four different proportions of NM and FM as shown in Figure 4. The logical address 0 starts from the top and the largest logical address ends in the bottom region where a page is composed of 100% NM. We assume that we have three boundary registers that mark the transition in the regions. Also, each region has a preset comparator that specifies whether blocks are in NM or FM. Using these auxiliary hardware, the access to different memory is initiated as follows: ① the logical address is compared against the boundary registers ② the request is forwarded to appropriate regions (4 regions available in our example) ③ the offset of the logical address is compared against auxiliary comparator ④ the request goes to NM if smaller than comparator or to FM if larger than comparator. Using the above step, only blocks with smallest offsets will be placed in NM. However, additional optimization steps will be added in the subsequent sections. In this paper, we use only four regions to demonstrate our design as four regions are enough to capture a large number of spatial access patterns and deliver good performance based on our experiments.

The physical data memory layout is the same as modern memories as shown in Figure 5. The top figure shows the addressing when accessing the configuration where 1 block out of 8KB resides in NM. Note that which region of configuration to access is determined by the boundary register described in the previous paragraph. For addresses within the same region, a comparator is used to check whether the request is to the block offset 0 or not. If 0, then it accesses NM (only block 0 resides in NM since region 1 only allows 1 block to reside in NM). When accessing NM, the column offset is 13 bits from the least significant bit as every 8KB has only one block in NM row. On the other hand, if the comparator check fails, then the request accesses FM. The FM access uses column offset that is 6 bits away from the least significant bit (64B block offset). However, block 0 must be excluded, so a address arithmetic is performed in the memory controller, which is widely used in die-stacked DRAM caching schemes [3], [4], [10]. The bottom scheme in Figure 5 shows the access scheme for the configuration where 2 blocks out of 8KB reside in NM. The accessing method is similar, but the offset bit width is different. Although the asymmetric layout of our memory layout looks sophisticated in the beginning, it can be implemented by using a set of registers and comparators. In addition, a slight modification in the addressing scheme as similarly done in prior work [3], [4], [10], [11] enables such novel, yet conventional memory layouts feasible without changing the actual memory layout design.

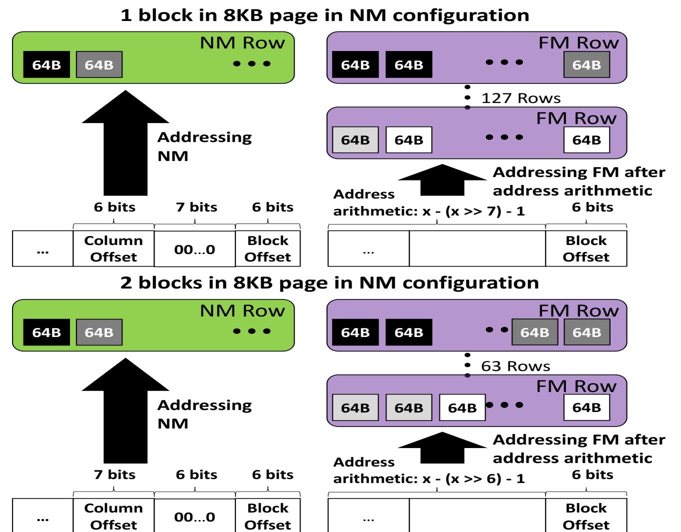

Fig. 5: Physical Access to FM and NM

B. OS Page Placement

The OS detects whether pages are hot or not using a predefined dynamic threshold as done in prior work [2]. Some modern processors have counters to detect hot pages. In these processors, we adopt the approach where a page access count is monitored. When the count crosses the threshold, one special bit in the Page Table Entry (PTE) is set. This bit indicates that a page is hot. The predefined threshold is dynamically adjusted at each epoch boundary so that the number of hot pages selected is close to the number of pages that can be accommodated in memory layout regions which

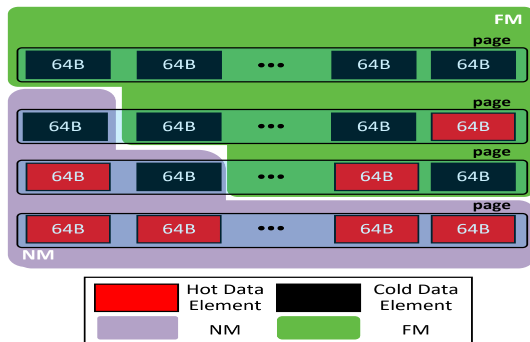

Fig. 6: Data Layout After OS Placement

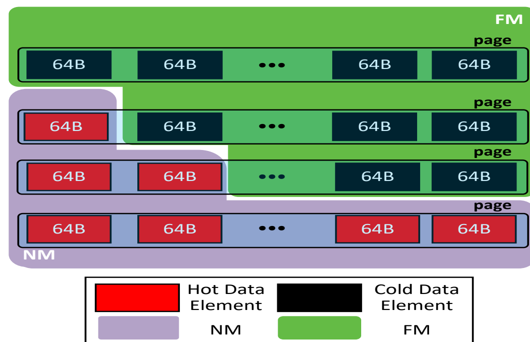

Fig. 7: Data Layout After Hardware Remap

have a small fraction or more of NM. After the hot page selection, the OS must be able to detect the access density of each page. Since the minimum memory visibility of the OS is the page size, we need a hardware support to detect which blocks within a page is accessed. We extend the TLB structure to include the bit vector where each bit indicates whether a block in the bit location is accessed or not. Each block within a page has its own bit. When the page is evicted from TLB, the number of set bits in the bit vector is summed up and used to determine the access density. We assume that we store the access density in PTE.

At the epoch boundary, the OS interrupts the system and walks down the pages which have the hot page bit set. These pages are migrated to appropriate regions according to the access density. For example, if the access density shows that this page has all of its blocks demanded, then it is placed in the last region where the entire region is composed of pure NM. The OS guarantees the page placement where the memory layout matches the access density. At this point, the data layout appears as in Figure 6. In this figure, the access density is matched with the region. The OS can place data only at the granularity of page size, so only page level placement is guaranteed. For example, a page with one hot 64B block is placed correctly in the appropriate region. However, the desired block is not placed in NM but FM due to its offset, and the control of the OS placement ends at the page level. Such placement is not optimal. This leads to the next step in our design where custom hardware is employed to place hot 64B blocks into desired NM space.

C. Hardware Block Remapping

After the OS page placement, blocks within a page are still in their original offset location. Since our memory layout is static with NM space located near the smallest offset (e.g., offset 0), if hot blocks are located in high offset locations, the memory requests will still be serviced from undesirable FM space. In order to place hot blocks into NM space, block granularity remap is implemented. Our remap structure is similar to the one presented in prior work [3], [5] where each block has its own remap entry. We are assuming a 64B block size and 8KB page size, so each page would have 128 blocks. 7 bits per block will be required as a tag and they are stored in the same NM row as the data. This would eliminate the need for expensive SRAM overheads as well as performance overheads of accessing a separate NM row to read each remap entry. Upon an access, the block is swapped into NM space of a page if not already placed there. If the NM space is enough to hold more than 1 block (e.g., 64 out of 128 blocks can fit in NM space), then the block is placed in a empty slot if there is one or the block replaces the oldest block in NM space. This is done in a round robin fashion and an additional 6-bit counter (in the case of 64 blocks in NM per page) is used per page to keep track of the oldest block.

With the remapping scheme, when a request comes, it first needs to know whether the block is in NM or in FM. In order to reduce one extra access to NM requesting the remapping entry to tell the location of the block, a block location predictor is used. The predictor has 256 entries that are indexed using the least 8 bits of *XOR*(PC, Mem Addr Offset). This method exploits the PC history and memory access patterns to increase the predictor accuracy. The predictor outputs a 7-bit location (a block location within an 8KB page), which is a speculated block location within the page. If correctly speculated, then the remap table lookup overhead is completely eliminated. Otherwise, the remap table lookup and the block fetching are serialized as in the case without a predictor. Therefore, misspeculation does not add any latency penalty. The predictor entry is updated once the remap entry is resolved. This predictor accuracy is over 80% on average across workloads used in this paper. The predictor is implemented per core and our 16-core architecture requires total 3KB storage overhead, which is negligible on a modern server class processor.

When all the hot blocks are heavily accessed, they are likely to be remapped into the NM space of a page. The results are shown in Figure 7 where hot blocks are demanded heavily, and thus, get migrated to the NM space of a page. As shown in the figure, all hot blocks are located in the NM space, and the NM capacity is not wasted by fetching undesirable blocks or being held empty due to filtering as in prior work. Note that there can be cases where the number of hot blocks is slightly larger than the number of blocks that NM space can hold in a page. For example, 3 blocks are hot and only 2 blocks can reside in the NM space. In this case, there might be slight interference due to blocks competing for the NM space. This problem can be solved by offering more configurations where more combinations of NM to FM ratios in a page can be offered.

Processor	Values
Number of Cores (Freq)	16 (3.0GHz)
Width	4 wide out-of-order

Caches	Values
L1 I-Cache (private)	32 KB, 2 way, 4 cycles
L1 D-Cache (private)	32 KB, 2 way, 4 cycles
L2 Cache (shared)	8 MB, 16 way, 11 cycles

HBM	Values
Bus Frequency	500 MHz (DDR 1 GHz)
Bus Width	128 bits
Channels	8
Ranks	4 Rank per Channel
Banks	2 Banks per Rank
Row Buffer Size	2KB (open-page policy)
tCAS-tRCD-tRP-tRAS	7-7-7-28 (memory cycles)

PCM	Values
Bus Frequency	400 MHz (DDR 0.8 GHz)
Bus Width	64 bits
Channels	4
Ranks	1 Rank per Channel
Banks	4 Banks per Rank
Read Latency	36 ns
Write Latency	90 ns

TABLE I: Experimental Parameters

In this paper, we tolerate such interferences as our work is focused on showing the cross-component scheme that can optimize the data placement in the context of heterogeneous memories. We leave further fine tuning and optimization to future work.

V. Experimental Setup

We used a Simics-based Flexus simulator [12] to simulate our scheme. Our configuration modeled a 16-core processor similar to SPARC T5 processor. For memory timing model, we used a modified version NVMain simulator [13]. Timing and configuration parameters of heterogeneous memories are listed in Table I. We assumed that NM to FM capacity ratio is 1:3 to emulate future systems where NM is a fraction of the FM capacity. As an example of fast and capacity-constrained memory, we use HBM generation 2 technology for NM in our evaluation and derived timing parameters from publicly available sources [14]. For slow memory that has large capacity, PCM technology is used as an example of FM memory in our evaluation with latency parameters derived from prior work and public sources [15]. We used the workload images from the Cloudsuite authors that are tuned, in steady state, and ready to be run on Flexus. Each workload was run for 1 billion instructions to warm up caches and memory systems in the beginning of the steady state phase. Then, additional 4 billion instructions were run to measure performance. The pages are randomly placed across NM and FM address space, and once they are placed, they are placed in the same location until the end of execution unless a page fault occurs. NM First is a slightly improved scheme which always brings new pages into NM. When NM is full, new pages are brought into FM instead. The NM First scheme never moves a page to FM unless a page fault evicts the page from NM. We compared our scheme against other state-of-the-art designs: Alloy Cache, CAMEO, Part of Memory (POM). We ran a representative collection of 6 benchmarks from CloudSuite 2.0 [9] on Flexus. All the workloads are run on Sparc architecture and tuned to steady state before any observation or measurement is performed.

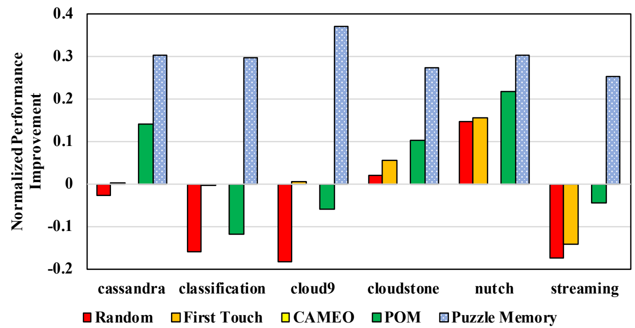

Fig. 8: Execution time normalized to CAMEO

VI. Results

A. Performance Evaluation

In this subsection, we present the evaluation results in Figure 8. The performance is presented as relative performance to prior state-of-the-art scheme, CAMEO. The Random scheme performs the worst (longest in execution time) in general because it does not leverage the presence of NM which has higher bandwidth and lower latency instead treating NM and FM with no difference. NM First is slightly better since it takes characteristics of NM into account. Hot pages change over time, however, sticking pages which are initially hot in NM loses performance as over a few epochs pages in NM are no longer hot. Meanwhile, most recent hot pages are placed into FM because NM is full. CAMEO is better than the two naive schemes because it keeps the most recent hot blocks in NM. Nonetheless, CAMEO does not necessarily obtain good performance as it uses a direct-mapped organization to reduce the access latency. Therefore, CAMEO is inherently susceptible to conflict misses, and thus, achieves approximately 30% lower overall performance than our scheme in case of cassandra as the hit rate is significantly lower (more discussions in the next subsection). POM performs better in cases as it does not immediately swap a block between NM and FM upon access. Instead, POM keeps a counter for each slot in NM and swaps the block only when a threshold is met. As a result, POM eliminates thrashing issues in CAMEO, and thus achieves a higher hit rate in some workloads such as cassandra. Yet, since POM's migration granularity is 2KB, it is vulnerable to fragmentation issues. For example, the access characteristics of classification workload shows 23% of memory requests access one 64B block within 2KB memory page. However, due to POM's design constraint, the whole 2KB page has to be moved into NM. This wastes both bandwidth and capacity of NM. This results in suboptimal performance, and this is clear in classification workload. CAMEO performs better than POM since it does not waste any BW and capacity due to such design constraint. Finally, Puzzle Memory outperforms all the other schemes evaluated since it captures hot pages

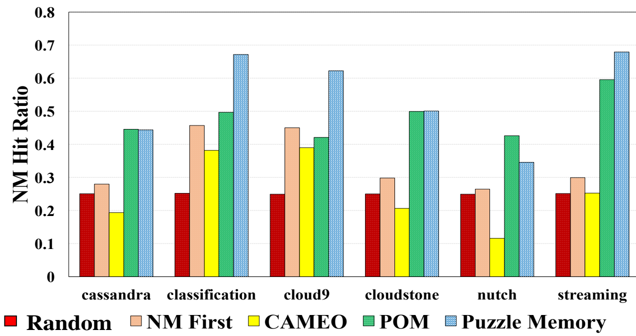

Fig. 9: NM Hit ratio (NM accesses/Total accesses)

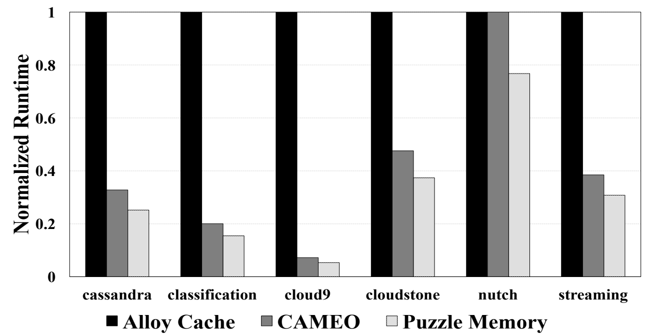

Fig. 10: Performance of using NM as cache (Alloy Cache) vs as added capacity (CAMEO and Puzzle Memory)

and reduces migration overhead at the same time. All newly allocated pages are first placed in NM region as newly allocated pages are most likely to be hot pages in the next several epochs. When a page is no longer hot, the associating counter will detect that the page is cold at the end of the epoch. The page will be moved to pure FM region such that NM always has room for new pages. This phase gets the benefits of both First Touch and POM schemes. Furthermore, pages will be migrated to regions whose the ratio of NM and FM capacity approximately matches that of hot and cold blocks within a page. This effectively eliminates the drawbacks of POM and CAMEO where their design constraint fixed the migration size although different workloads show different spatial locality characteristics. Puzzle Memory has NM regions that are designed to work like CAMEO for low spatial locality workloads and those regions that are for high spatial locality workloads just like POM. Cloud9 is an excellent example where the workload has a good mix of accesses that show low and high spatial locality. The performance results of CAMEO and POM both perform poorly since either scheme only benefits one set of memory regions. In other words, POM largely benefits high spatial locality memory regions, and CAMEO does benefit low spatial locality regions due to their strict migration granularity constraint. This is where Puzzle Memory shines where it is ready to benefit both regions as the memory organization inherently assumes that no single migration granularity fits all workloads.

B. NM Hit Ratio

Conceptually, an ideal scheme will leverage temporal and spatial locality of data and direct most accesses to NM as well as reduce the migration costs to the least. NM hit ratio is a good indicator on whether a scheme captures and leverages locality of workloads. As our NM to FM ratio is 1:3, randomly placing pages in the whole memory gets roughly one fourth of accesses to hit in NM. This matches the hit ratio of the Random scheme as can be seen in Figure 9. The hit ratio of NM First is higher because any available space in NM is always guaranteed to be used before FM is filled. CAMEO does not always have higher hit ratio than the Random scheme, as thrashing would cause hit ratio in NM to be very low for workloads like *nutch*, and this reported low hit ratio is also confirmed in literature [4], [10]. POM has very high hit ratio because it leverages spatial locality by swapping larger blocks into NM. Our scheme, in general, has the highest hit ratio of all due to the flexibility of our scheme which makes it better able to adapt to multiple patterns of accesses of workloads. Besides comparison between schemes, it is also interesting to look at hit ratio across workloads. We can see workloads that have good locality such as *classification* and *cloud9* are easy to capture in NM resulting in high hit ratios. The trend is consistent across different schemes. The thrashing effect is especially high in *nutch*. The footprint of *nutch* is also the smallest. A lot of data reuse occurs in this workload and this irregular access pattern causes CAMEO to perform worse. This is the only workload for which POM has a higher hit ratio than our scheme because of better spatial locality in 2KB blocks.

C. Using NM as Cache VS as Added Capacity

We performed a study on performance improvement of using NM as additional capacity over using NM as a cache. The most naive scheme to utilize NM is to treat it as a level of cache between LLC and FM, e.g. Alloy Cache. The typical capacity ratio of NM to FM is 1:3 or even higher. Therefore, not adding capacity of NM into the memory space is a significant lost. In such case, the total capacity of memory space is much smaller and the system is subject to a critical number more page faults. Figure 10 shows a comparison between using NM as cache or as part of whole memory. All execution time is normalized to the performance of Alloy Cache. We see a significant amount of reduction in runtime when NM is used as added capacity and performance is improved by 3.2X (CAMEO), and 4.2X (our scheme). In *nutch*, since the footprint of *nutch* is smaller than 2GB, the working set is completely memory resident. CAMEO essentially regresses to Alloy Cache so no difference is observed. The 4.2X non-linear improvement with 1:3 NM-to-FM capacity ratio reemphasizes the importance of not wasting NM capacity by using it as a part of memory rather than a cache.

VII. Related Work

A plethora of work has been proposed to efficiently use heterogeneous memory technologies. A large number of proposals [4], [11], [16]–[18] have focused on using these memories as another level of cache. Yet, managing large metadata overhead of such multi-gigabyte caches is difficult, thus such work focused on reducing the metadata or storing the metadata in die-stacked DRAM for scalability. These proposals use NM as hardware caches, and thus, do not take advantage of the NM capacity as visible memory space. Yet, the NM capacity is scaling and is expected to become a non-negligible amount of the main memory, and thus, more recent work [3], [5] has focused on using this capacity as a part of OS visible space. However, many of these schemes derived caching techniques from existing SRAM caching techniques and they are susceptible to problems that exist in SRAM caches such as thrashing. Purely software solutions without additional hardware support [2] have been proposed, yet the lack of fine granularity (e.g., 64B data migration) limits these schemes from achieving full potential since the minimum visible OS granularity is the OS page size, which is 4KB in case of x86 architecture. Multifractional memory architectures can also be realized using memory architectures such as VaWiRAM [19]

VIII. Conclusion

In this paper, we have presented a novel heterogeneous data management scheme, Puzzle Memory, that allows multifractional splitting of memory regions into fast and slow regions. Our innovative memory data layout is customized to allow multiple subblock swapping within the same page between FM and NM. At the same time, the OS periodically reorganizes pages to a location that best meets the spatial locality of each page. The cross-component efforts have resulted in an average performance improvement of 40% over the prior scheme by placing as much useful data as possible into the NM and minimizing the waste of capacity constrained NM.

IX. Acknowledgement

This research was supported in part by National Science Foundation under grants 1745813 and 1725743 and by Oracle Corporation. Authors would also like to acknowledge computational resources from TACC. Any opinions, findings, conclusions or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

References

[1] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 72–83, ACM, 2013.

[2] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[3] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[4] D. Jevdjic, G. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, 2014.

[5] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[6] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

[7] M. Oskin and G. H. Loh, "A software-managed approach to die-stacked dram," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 188–200, IEEE, 2015.

[8] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 340–349, IEEE, 2011.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 37–48, ACM, 2012.

[10] 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*, ISCA '13, 2013.

[11] 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*, MICRO-45, 2012.

[12] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–34, 2004.

[13] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2015.

[14] JEDEC, "High Bandwidth Memory (HBM) DRAM Gen 2 (JESD235A)." https://www.jedec.org, 2016.

[15] Micron Technology Inc., "TN-46-03 Calculating Memory System Power for DDR," 2001.

[16] F. Hameed, L. Bauer, and J. Henkel, "Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, 2013.

[17] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Integrating DRAM Caches for CMP Server Platforms," *IEEE Micro*, vol. 31, no. 1, 2011.

[18] G. Loh and M. D. Hill, "Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap," *IEEE Micro*, vol. 32, May 2012.

[19] L. K. John, "Vawiram: a variable width random access memory module," in *Proceedings of 9th International Conference on VLSI Design*, pp. 219–224, Jan 1996.