Cache/Memory Coordinated Fair Scheduling for Hybrid Memory Systems

Di Chen, Haikun Liu, Hai Jin, Xiaofei Liao
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
{chendi,hkliu,hjin,xfliao}@hust.edu.cn

ABSTRACT
Hybrid memory systems comprising DRAM and *Non-Volatile Memory* (*NVM*) have gained ever-increasing attention for building large-capacity and energy-efficiency main memory. Nevertheless, there remain challenges to best utilize them because of the asymmetrical access latencies of DRAM and NVM. Traditional memory request scheduling schemes usually lead to notable application performance degradation and unfairness. In this paper, we propose a *cache/memory coordinated* (*CMC*) memory request scheduling strategy to address inter-process memory interference and unfairness scheduling problems in hybrid memory systems. Taking the asymmetrical access latencies into account, CMC leverages *average access time* (*AST*) to quantify inter-process memory interference in hybrid memory systems. CMC preferentially schedules applications with a small number of memory requests, while schedules memory-intensive applications according to their AST. CMC further filters some memory blocks with poor locality to improve the efficiency of *last level cache* (*LLC*). Experimental results show that CMC improves system performance and fairness by up to 16% and 9% than traditional *first-ready-first-come-first-service* (*FRFCFS*) memory scheduling policies, respectively.

CCS Concepts
•Computer systems organization → Heterogeneous (hybrid) systems; • Hardware → Memory and dense storage.

Keywords
Non-Volatile Memory, hybrid memory systems, fairness, dead blocks, cache filter, memory scheduling.

1. INTRODUCTION
With the rapid growth of data volume in the mobile Internet and cloud computing era, computer systems are eager for large-capacity main memory to improve data processing efficiency via in-memory computing paradigm. However, current DRAM scaling is not able to satisfy the ever-increasing memory requirement. Hybrid memory systems composed of DRAM and NVM are the most promising approaches to enlarge the main memory capacity and improve energy efficiency [1-5]. As a typical prototype, Hewlett Packard's *The Machine* project leverages both DRAM and NVM to build a single machine containing 160 TB of main memory [6].

HP3C 2020, June 27–29, 2020, Guangzhou, China

ACM ISBN 978-1-4503-7691-4/20/06…$15.00
DOI: https://doi.org/10.1145/3407947.3407967

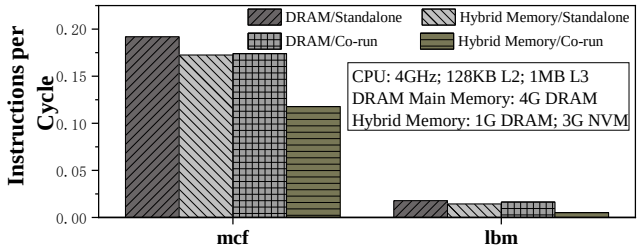

Figure 1. Inter-process memory interference

Although NVM is able to expand the capacity of main memory significantly, hybrid main memory architectures may suffer performance degradation since the high access latency and high write power consumption of NVM. Particularly, this performance degradation is often notable when multiple applications co-run in multi-core servers.

To demonstrate the impact of inter-process memory interference on application performance, we run applications *mcf* and *lbm* together in a DRAM-only system and a hybrid memory system, respectively. For comparison, we also run each application alone in the two memory systems, respectively. When the applications run in the hybrid memory system, data is interleaved in DRAM and NVM according to the DRAM-to-NVM ratio. Figure 1 shows the experimental results. When *mcf* and *lbm* co-run in the DRAM-only system, they only suffer 9.33% and 8.38% performance degradation compared to the standalone executions, respectively. However, co-running *mcf* and *lbm* in the hybrid memory system lead to even 31.67% and 63.89% performance degradation compared with the standalone executions. This implies hybrid memory systems exacerbate memory interference among applications. Traditional memory scheduling schemes become inefficient in a hybrid memory system.

There have been a number of studies [7-13] on mitigating the inter-process memory bandwidth contention in DRAM-based main memory systems. Generally, these studies all aim to mitigate processor starvation by changing the scheduling priorities of memory requests. These scheduling policies group applications according to their memory access behaviors. Applications are divided into groups with different priorities, and the memory

requests from high-priority applications are preferentially scheduled. DRAM-based memory request scheduling strategies can be classified into three categories:1) Prioritizing light threads to improve system throughput [7, 11]; 2) Grouping requests from the same thread into batches to improve the hit rate of row buffer [2, 10]; 3) Mapping data of mutually interfering threads to different channels/ranks to reduce bank conflict [8, 12].

Although these scheduling strategies are effective in DRAM-based main memory, they may be not efficient in hybrid memory systems. In DRAM-based main memory systems, inter-process memory interference may block the execution of processors, and leads to significant performance degradation due to unfair memory request scheduling. Such unfairness is exacerbated in hybrid memory systems due to the asymmetric memory access latency between DRAM and NVM. Since it takes much more CPU clocks to access NVM than DRAM, different data distributions may lead to distinct performance differences in hybrid memory systems. In addition, because the NVM write latency is much higher than the NVM read latency, the proportion of read and write operations in an application can also affect the scheduling fairness and application performance. Previous schemes are not able to evaluate the impact of memory interference on application performance in hybrid memory systems.

In this paper, we propose a *cache/memory coordinated* (CMC) scheduling strategy to address the memory interference and fairness issues in hybrid memory systems. First, we profile applications' memory access behaviors from the memory controller and *last level cache* (LLC) at runtime. Applications are then clustered into two groups: memory-light and memory-intensive. Second, we advocate average service time to evaluate the degree of memory interference for applications in hybrid memory systems. Third, we propose a grouping-based request scheduling strategy in which requests from the memory-light group are prioritized to improve system throughput, while requests from the memory-intensive group are scheduled according to their rank to guarantee system fairness. Finally, we identify blocks with poor access locality and do not fill them into LLC.

The major contributions of this paper are as follows:

- We observe that applications face significant memory interference and unfairness issues in hybrid memory systems, and reveal that the roots of this problem are attributed to data distribution, memory read/write asymmetry, and inter-process memory bandwidth contention.
- We advocate *Average Service Time* (AST) to evaluate the degree of memory interference among applications in hybrid memory systems. If an application's AST is higher than the AST of the host server, it suffers harmful memory bandwidth contention with others and should be prioritized.
- We propose a cache/memory coordinated memory request scheduling mechanism. It profiles applications' memory access patterns in LLC at runtime, and filters memory blocks with poor access locality in the memory controller.
- We evaluate memory scheduling fairness and system throughput by comparing it with two previous memory scheduling policies. Experimental results show that CMC achieves an average 9.43% and 7.43% improvement on system throughput and fairness than FRFCFS in hybrid memory systems.

The rest of this paper is organized as follows. Section II introduces the related work. Section III analyzes the root of unfairness memory scheduling in hybrid memory systems and motivates our proposal. Section IV introduces our fairness scheduling model. Section V and VI describe the implementation and evaluation of our scheduling strategy, respectively. We conclude in Section VII.

2. RELATED WORK

2.1 Memory Scheduling in DRAM-based Memory Systems

There have been a large number of studies on memory scheduling for multiple applications in DRAM-based memory systems.

TCM [7] divides threads into two clusters: latency-sensitive cluster and memory-sensitive cluster. Threads in the latency-sensitive cluster have a small number of memory accesses. Since these memory requests have negligible interference to other threads, prioritizing these requests can impel these light threads to execute quickly in CPU. Therefore, *TCM* preferentially schedules memory requests of threads in the latency-sensitive cluster to improve system throughput. Besides, to guarantee the scheduling fairness of threads in the memory-sensitive cluster, *TCM* reshuffles the priority of each thread periodically.

Channel Partitioning [8] gives the highest scheduling priority to lightweight applications to achieve high system throughput. For other applications with intensive memory accesses, *Channel Partitioning* attempts to classify them into two categories based on threads' *Row Buffer Locality* (RBL). Then, *Channel Partitioning* maps threads with low RBL and high RBL into different channels to reduce memory interference at the end of each time window. In the next period, a new page is allocated to an unassigned channel when a page fault occurs.

FST [9] guarantees memory scheduling fairness by cooperatively scheduling the shared main memory and caches. This coordinated scheduling avoids conflicting strategies on LLC and main memory when scheduling separately. *FST* achieves this goal by limiting the number of memory requests in each core sent to the memory controller.

GPU applications usually have a large number of concurrent threads, and they can issue thousands of memory access requests at the same time. Therefore, in a CPU-GPU integrated system, the memory access requests from the CPU cores are often seriously delayed, resulting in CPU starvation. In *SMS* [10], memory requests from CPU/GPU threads that access the same DRAM row are grouped into a batch, and memory requests in each batch are processed sequentially. Batch processing of memory requests significantly increases row buffer hit rate and system performance.

Overall, to mitigate inter-process memory interference, previous DRAM-oriented request scheduling strategies are designed for homogeneous memory systems. They do not consider the impact of significant memory interference on application performance in hybrid memory systems.

2.2 Memory Scheduling in Hybrid Memory Systems

Hybrid memory systems combining DRAM with NVM have both advantages of them to build large-capacity, low-cost, and high-

performance main memory systems. However, due to the asymmetry of DRAM and NVM in read/write latency, power consumption, and lifetime, hybrid memory systems usually face challenges of fair scheduling and memory interference.

There has been a litter work [1, 2, 4] on the performance improvement of hybrid memory systems through memory scheduling. Those studies have focused on scheduling memory requests from regular applications and persistent applications.

Since NVM is able to guarantee data persistence upon power off or system crash, NVM can be also used as persistent working memory. Applications that leverage NVM as working memory are called regular applications, while applications that use NVM as persistent storage are called persistent applications. Since persistent applications are constrained to data consistency, the order of write requests should be guaranteed. Therefore, barriers are inserted into memory requests to guarantee the write order in persistent applications. NVM-Duet [1] preferentially schedules requests from regular applications to improve memory throughput because these memory requests are not constrained by the barriers.

Furthermore, persistent applications are typical applications such as databases or file systems that generate a large number of consecutive write requests. The write operations are usually scheduled with a low priority. The large number of consecutive write requests from persistent applications can lead to frequent bus turnarounds. To reduce the overhead of bus turnarounds, FIRM [2] calculates and schedules the optimal batches of requests during each bus turnaround.

However, those memory scheduling strategies have not fully explored the root of memory interference and scheduling unfairness issues in hybrid memory systems, and also does not provide a quantitative fairness model to direct the memory scheduling.

3. MOTIVATION

3.1 Inapplicability of Previous Approaches

Traditional memory scheduling strategies face two challenges in hybrid memory systems:

First, the unfairness of memory scheduling is exacerbated in hybrid memory systems. When we use the maximum slowdown model [7] to evaluate the scheduling unfairness, the unfairness of two applications changes from 1.10 in the DRAM memory system to 2.77 in the hybrid memory systems, as shown in Figure 1. The reason is that the high delay of NVM access increases the blocking time of applications.

Second, traditional scheduling strategies are not effective enough to quantify the impact of memory interference on application performance degradation in hybrid memory systems. To mitigate the scheduling unfairness, we should identify applications that are vulnerable to competition and prioritize these applications. Figure 1 shows the IPCs of two workloads running in the DRAM and hybrid memory systems, respectively. In the DRAM-only system, the co-running applications show similar performance degradation. However, in the hybrid memory system, the scheduling strategy is unfair for *lbm* since it suffers higher performance degradation than *mcf* (63.89% vs. 31.67%). This means traditional memory scheduling strategies may lead to a higher degree of unfairness. This is due to the asymmetrical memory access latencies between DRAM and NVM. The inter-process interference is not the only factor that affects system fairness in hybrid memory systems.

Besides, we find most previous work only leverage memory access behaviors to schedule memory requests, rather than to guide the filling strategy of LLC. Some previous works [14-17] try to improve cache efficiency by evicting blocks with poor locality. Blocks are called dead blocks if they would not be accessed again before it is evicted from LLC. Since dead blocks is not accessed, evicting these blocks would not result in a decrease of LLC hit rate and system performance, and more precious LLC space can be used to store frequently re-referenced blocks.

3.2 Roots of Unfair Scheduling in Hybrid Memories Systems

In this section, we analyze the root of unfair memory scheduling issues in hybrid memory systems. As shown in Figure 2, the scheduling unfairness is mainly attributed to three factors in hybrid memory systems: data distribution, memory read/write ratio of applications, and inter-process memory contention.

3.2.1 Data Distribution

Figure 2(a) shows the unfairness caused by data distribution in hybrid memory systems. In DRAM-based main memory, the memory access latency is roughly uniform regardless of row buffer hit or miss. As the NVM access latency is much longer than the access latency of DRAM, an application may suffer higher performance degradation if most of its data are allocated in NVM. In Figure 2(a), application A and application B are the same application with different data distributions on DRAM and NVM. Compared to application A, application B demonstrates a higher average memory access time because most of its data are placed on NVM.

The channel-interleaving address mapping scheme can be exploited to mitigate the unfairness caused by data distribution in hybrid memory systems. To improve memory level parallelism, channels/ranks/banks are usually encoded at the low address bits. In our hybrid memory systems, addresses are decoded in the order "subarray:row:rank:bank:channel:column" so that data can be uniformly distributed across different channels. This channel-interleaving scheme is widely adopted by many industrial hardware architectures, such as Intel Ivy Bridge architecture [18].

3.2.2 Read/Write Asymmetry

As a NVM write operation consumes much more system clocks than a NVM read operation, the read/write ratio of applications can also lead to unfairness of memory scheduling, as shown in Figure 2(b). Suppose application A and application B are two applications with the same number of memory access requests and different read-to-write ratios. As NVM takes more system clocks to serve a write request than a read request, application B with a higher write-to-read ratio suffers greater performance degradation than application A. Therefore, we should consider read-to-write ratio when we evaluate the impact of memory interference on application performance.

3.2.3 Inter-process Memory Bandwidth Contention

Figure 2(c) illustrates an example of inter-process memory contention, which may lead to significant performance degradation and unfairness. Application A running on core 0 is a streaming application with a large number of memory accesses, while application B running on core 1 is a computing-intensive application with a small number of memory accesses. When applications A and B co-run together in an SMP (*Symmetrical*

Multi-Processing) system, the mixed memory requests in the queue are shown in Figure 2(c).

According to the *First Come First Serve* (FCFS) scheduling strategy, the memory request *B1* is not scheduled until the requests from application A are served. As requests from application A occupies the most space of the transaction queue, application B is hung up for a long time. Therefore, the performance of the memory-light application (Application B) is significantly degraded if the order of the memory requests is not changed. Since application B is a CPU-intensive application, if the small number of requests from application B can be prioritized, application B can run much fasterwithout being blocked by the memory reference. Because the number of requests from application B is small, the performance degradation of application A is also not significant, and the average response time of the system is greatly improved.

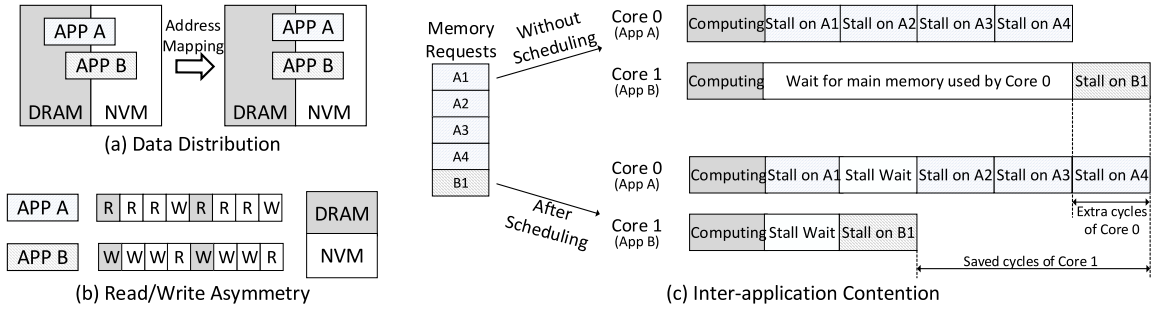

Figure 2. Unfairness in hybrid memory systems

4. DESIGN

In this section, we first analyze the performance metrics to characterize memory access behaviors in hybrid memory systems, and then present our fair memory scheduling model and LLC filtering strategy.

4.1 Memory Access Behaviors in Hybrid Memory Systems

Intuitively, applications with a small number of memory requests should be prioritized to improve system throughput. Besides, applications that are highly susceptible to memory interference in hybrid memory systems are prioritized to guarantee system fairness. To characterize applications' memory access behaviors, we profile applications at runtime and classify them into different groups. Overall, there are five performance metrics to characterize memory access behaviors in hybrid memory systems.

***Misses Per Kilo Cycles* (MPKC).** Most previous studies [7, 8, 10] use MPKI (*misses per kilo instructions*) to quantify cache miss rate, which reflects the data locality of applications. However, MPKI cannot quantify whether an application is memory-intensive. In contrast, we exploit a more accurate metric—MPKC to quantify applications' requests intensity. It represents the number of memory requests received by the memory controller in a short period.

***Row Buffer Locality* (RBL).** RBL is used to characterize the spatial locality of an application on banks. Row buffer is a high-speed cache within each bank. When LLC requires a 64-byte cache block, the row containing the data block is entirely read to the row buffer. If subsequent memory requests refer to the same row, the data can be read directly from the row buffer without accessing the slow data array. Specifically, RBL reflects the hit rate of the application's memory access requests on the row buffer. We set a shadow register in each bank for each application to record the row accessed currently. The row hits if a new request locates on the same row accessed in the last time. To calculate RBL, we use a counter for each application to count the number of row buffer hits.

***Bank Level Parallelism* (BLP).** Since banks within a rank can be accessed in parallel, multiple requests of applications can be handled simultaneously if the data are distributed in different banks. An application with a high BLP means that its average memory access time is shorter. High bank-level parallelism is able to mitigate inter-process memory competition. We use a shadow register for each application to track its BLP. This register records the maximum bank parallelism of an application.

Read/Write Ratio. Since a NVM write operation consumes more system clocks than a NVM read operation, an application with a larger proportion of write requests often illustrates higher average memory access time (see more in Section IV.B). To calculate the read-to-write ratio, we record the numbers of read requests and write requests for each application in the memory controller, respectively.

LLC Hit Rate. LLC hit rate reflects data access locality of an application. For an application with a low hit rate, data blocks usually have a low probability to be accessed again in the near future. In other words, a block in an application with a low hit rate is more likely a dead one. Therefore, LLC hit rate can be used to recognize dead blocks in the cache. LLC hit rate can be obtained by programming Intel *Performance Counter Monitor* (PMU).

4.2 Fair Memory Scheduling Model

Memory-intensive applications typically issue a large number of memory requests. Prioritizing these requests usually have little impact on the total system throughput. For these memory-intensive applications, the main goal of scheduling is to avoid starving other applications' memory requests, guaranteeing the fairness of memory scheduling. The memory-intensive applications usually have different features of memory requests, read/write ratios, row buffer hit ratios, and bank-level parallelism. To fairly schedule co-running applications, we need to find out which applications are susceptible to memory interference and which applications' performance is more sensitive to hybrid memory systems.

We propose *Average Service Time* (AST) to evaluate the impact of hybrid main memory systems on application performance. AST represents the average response time to finish a memory request of an application. In a period, if an application has a higher AST, this means that the hybrid memory system has a higher impact on the application performance degradation compared with other

applications. Hence, it is necessary to preferentially schedule applications with high AST to ensure the fairness of memory scheduling. Our goal is to guarantee that all co-running applications experience similar AST.

Let P_D, P_{NR}, and P_{NW} represent the proportions of DRAM requests, NVM read requests, and NVM write requests, respectively. Let T_D, T_{NR}, and T_{NW} represent the latency of DRAM access, NVM read, and NVM write, respectively. AST can be presented as Equation (1).

$$\mathrm{AST} = P_D * T_D + P_{NR} * T_{NR} + P_{NW} * T_{NW} \quad (1)$$

A memory request is either a DRAM request or a NVM request, and a NVM request is either a NVM read request or a NVM write request. Therefore, we have Equation (2).

$$P_{NR} + P_{NW} = P_N,\ P_D + P_N = 1 \quad (2)$$

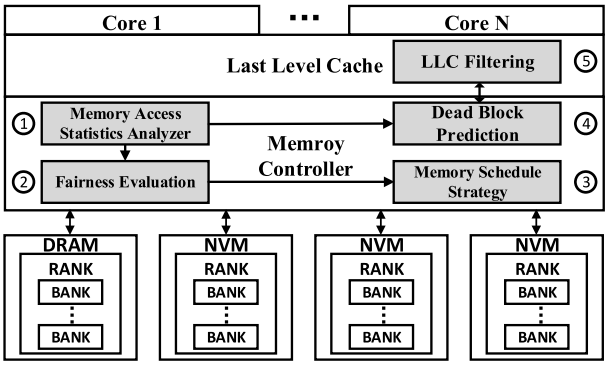

Figure 3. Architecture of CMC

Let α, β, and γ denote the ratio of DRAM access latency to NVM read latency, the ratio of NVM read probability to NVM write probability, and the ratio of NVM write latency to NVM read latency, respectively.

$$\alpha = \frac{T_D}{T_{NR}},\ \beta = \frac{P_{NR}}{P_{NW}},\ \gamma = \frac{T_{NW}}{T_{NR}} \quad (3)$$

Let T_{NRH} and T_{NRM} present the latency of row buffer hit and row buffer miss of a NVM read request, respectively. Let *BLP* present the degree of bank level parallelism, and *RBL* represent the probability of row buffer hit of an application. Since the memory references to different banks are isolated, requests to different banks can be served in parallel. Consequently, the average access time of an application can be amortized to all concurrent banks. Finally, the latency of NVM read can be presented as Equation (4).

$$T_{NR} = \frac{T_{NRH} * RBL + T_{NRM} * (1 - RBL)}{BLP} \quad (4)$$

Combining the above equations, AST can be calculated in Equation (5).

$$\mathrm{AST} = \left[\alpha + \left(\frac{\beta+\gamma}{1+\beta} - \alpha\right) * P_N\right] * \frac{T_{RBH} * RBL + T_{RBM}(1 - RBL)}{BLP} \quad (5)$$

At the end of each time window of scheduling, we calculate the AST of each application. In the next time window, applications with high AST will be scheduled preferentially to improve the system fairness.

4.3 LLC Filtering

Due to the limited space of LLC, it is essential to use the LLC efficiently. Cache filtering schemes are widely exploited to avoid filling LLC with cold data blocks, which are directly fetched to L2 cache in an on-demand manner. Thus, cache filtering is able to best utilize the limited LLC space, and improve LLC hit rate. For many applications, there are usually a large number of dead blocks in the LLC [14, 15], and these dead blocks will not be accessed again before they are evicted. Dead blocks waste the precious LLC space and can decrease the LLC hit rate and system performance. However, it remains challenges to identify dead blocks due to complicated memory access patterns of applications.

Usually, memory access patterns can be classified into four kinds: recency-friendly, streaming, thrashing, and random [14, 17]. For the recency-friendly access pattern, blocks are with good locality and would be accessed frequently. These blocks should not be filtered by the memory controller since they can improve the LLC hit rate and system performance. For the streaming access pattern, blocks of continuous address spaces are accessed sequentially. The hit rate of LLC is low and blocks in streaming applications should be bypassed. For a thrashing access pattern, the working size of an application is often larger than the cache size, resulting in massive cache misses. Usually, applications in recency-friendly and thrashing access patterns do not represent low RBL and low BLP at the same time. For random access patterns, applications often represent a mixture of three other access behaviors, a low LLC hit rate and low RBL. Current LLC replacement policies [4, 17] have been able to reduce the footprint of the streaming access pattern on the LLC by changing re-reference distance, but there are still challenges in reducing the dead blocks footprint in the random access pattern.

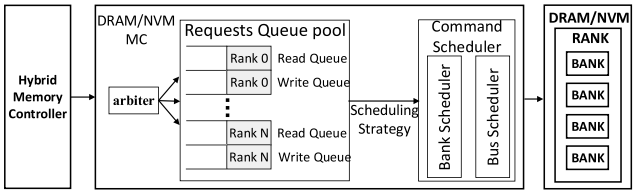

Figure 4. Architecture of hybrid memory controller

Our *cache/memory coordinated* (CMC) memory scheduling strategy only tries to filter some dead blocks that lead to both low LLC hit rate and low RBL. These dead blocks have poor inter and intra-block data locality. Filtering these dead blocks does not compromise LLC hit rate and row buffer hit rate.

To achieve this goal, we design a memory controller and LLC coordinated cache filtering mechanism. At the end of each epoch of memory scheduling, our CMC memory scheduling strategy recalculates the RBL of applications on each bank. In the next period, we check the RBL of applications when the memory controller receives a memory response returned by a bank. If the RBL of the application is lower than a given threshold, this request is marked as a potential candidate which would be directly fetched to L2 cache (bypassing LLC). CMC further checks the LLC hit rate of the application. If it is lower than a given threshold, this requested data block is marked as "bypass" finally.

5. IMPLEMENTATION

5.1 System Layout

Figure 3 shows the architecture of CMC, a cache and memory controller coordinated memory request scheduling mechanism. This mechanism mainly consists of five modules in the following.

1) The *Memory Access Statistic Analyzer* module tracks and collects the memory access statistics for each application at

runtime. These memory access characteristics include MPKI, RBL, BLP, read-to-write ratio, and LLC hit rate.

2) The *Fairness Evaluation* module uses the data acquired by the *Memory Access Statistic Analyzer* module to calculate the AST of each application through Equation (5). Applications are clustered into two groups: a memory-light group and a memory-intensive group. Requests from the memory-light group are prioritized without ranking, while applications in the memory-intensive group are ranked with their AST.

3) The *Memory Scheduling* module schedules memory requests received by the memory controller in multiple request queues. Requests from the memory-light group are prioritized to improve system throughput. Applications from the memory-intensive group are scheduled according to the rank of AST, and applications with a high AST are prioritized. Figure 4 illustrates the scheduling in the hybrid memory controller.

4) The *Dead Block Prediction* module is responsible for marking dead blocks for bypassing in the future. This process can be divided into two stages. At the first stage, the module identifies all dead blocks as candidates for bypassing. When the memory controller receives the requested data, this module checks the RBL of the application. If the RBL of the application is lower than the given threshold, this data block is marked as "potential bypassing" for further checking. In the second stage, the module determines whether a block should bypass the LLC by checking the cache hit rate of the application. The module only selects the "potential bypassing" blocks with a low cache hit rate as "bypass" finally.

5) The *LLC Filtering* module filters blocks that marked as "bypass" when filling the LLC. These dead blocks are not filled into LLC, and thus more space is saved for filling blocks with good locality.

5.2 Priority-based Memory Scheduling

Algorithm 1 shows our priority-based memory request scheduling in hybrid memory systems. Requests from memory-light applications are prioritized over other requests. As these applications issue a small number of memory requests, there is no need to sort the scheduling order among them. Requests that are hit in the row buffer are prioritized over other requests. Among these requests, requests with a high AST are prioritized to improve system fairness and system performance. If an application's memory requests reach a starvation threshold, these requests are prioritized over other applications to avoid memory request starvation. At last, the oldest requests are prioritized over the newly-arrived requests.

Algorithm 1 CMC: Memory requests scheduling prioritization

Input: *queue*, the request queues (read/write) in memory controller
Output: *nextRequest*, the request to be scheduled in the next epoch

```
/*Requests from memory-light applications are prioritized*/
find_Light_Request(&queue, &nextRequest)
/*Requests leads to row buffer hit are prioritized*/
find_Row_Buffer_Hit_Request(&queue, &nextRequest)
    Applications with high AST are prioritized
/*Requests that exceeds the starvation threshold are prioritized*/
find_Starved_Request(&queue, &nextRequest)
    Applications with high AST are prioritized
/*Oldest requests are prioritized*/
find_Oldest_Ready_Request(&queue, &nextRequest)
```

6. EVALUATION

6.1 System Configuration

We adopt a full-system architectural simulator combining NVMain [19] and GEM5 [20] to evaluate CMC. The CPU and caches are modeled in GEM5, while NVMain simulates the hybrid memory system includes one DRAM channel and three NVM channels. Table 1 shows the configuration of octa-core processors and hybrid memories.

Table 1. Configuration of Hybrid Memory System

Processor	8 cores, 2.0GHz, 32KB i-cache 64KB d-cache, 128KB L2
LLC	64 KB cache-line, 16-way associative, 2MB
Hybrid Memory	4 channel: 1 DRAM channel; 3 NVM channel
DRAM	1 rank, 8 banks, 128 queue size row buffer hit: 18 cycle; read/write: 36 cycle
NVM	1 rank, 8 banks, 128 queue size row buffer hit: 18 cycle read: 144 cycle; write: 288 cycle

6.2 Workloads

We evaluate CMC with multi-programmed workloads which are generated from the SPEC CPU2006 benchmark suite. The memory intensity of those applications is quantified by MPKC. We classify these benchmarks into three categories based on the average MPKC when each application runs solely: light-workload (MPKC $\leqslant$ 3), medium-workload (3 < MPKC $\leqslant$ 15) and heavy-workload (MPKC $\geqslant$ 15), as shown in Table 2. It should be noted that this classification is only used for selecting benchmarks to generate different workloads. We build three kinds of multi-programmed workloads using applications with different memory intensity, as shown in Table 2.

Table 2. Workload Configuration

Application/Workload		**Characteristics**
namd; libquantum; soplex		MPKC $\leqslant$ 3 (Light-workload)
bzip2; cactusADM; gromacs; xalancbmk; gcc; hmmer; milc; omnetpp		3 < MPKC $\leqslant$ 15 (Medium-workload)
sjeng; gobmk; mcf; lbm		MPKC > 15 (Heavy-workload)
Multi-programmed Workload	**A**	Over 50% light-workload
	B	Over 50% medium-workload
	C	Over 50% heavy-workload

6.3 Evaluation Metrics

We evaluate CMC with both system throughput and memory scheduling fairness. The system throughput represents the number of instructions executed during a period and usually is measured by *instructions per cycle* (IPC). We evaluate the system throughput using a weighted speedup model [7, 9], as shown in Equation (6), where IPC_i^{shared} represents the IPC of application *i* when it co-runs with other seven applications in an octa-core server together, and IPC_i^{alone} represents the IPC of each application when it runs alone on one core in the same server (other seven cores are not used), and N is the total number of cores in the server. We evaluate the scheduling fairness of hybrid

memory using a maximum slowdown model [7, 9], as shown in Equation (7).

$$\text{Weighted\_Speedup} = \sum_{i=0}^{N-1} \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad (6)$$

$$\text{Maximum\_Slowdown} = \max_i \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad (7)$$

6.4 Experimental Results

We evaluate four memory scheduling strategies in hybrid memory systems: FRFCFS, TCM [10], CMC-noFilter, and CMC. CMC-noFilter is a variant of CMC that excludes the cache filtering scheme. It is particularly used to evaluate the effect of the dead block bypassing strategy. Figure 5 shows the geometric average throughput, IPC (instructions per cycle), and fairness of the four scheduling policies for the three multi-programmed workloads. Overall, CMC achieves the highest system performance and fairness for all policies.

6.4.1 Performance

Figure 5(a) shows the system throughput for the four scheduling strategies. Compared with FRFCFS, TCM and CMC achieves a significant improvement on system throughput. For workloads A, B, and C, TCM achieves 3.78%, 10.09%, and 6.68% system throughput improvement, respectively, while CMC improves th system throughput by up to 6.27%, 10.91%, and 16.02%, correspondingly.

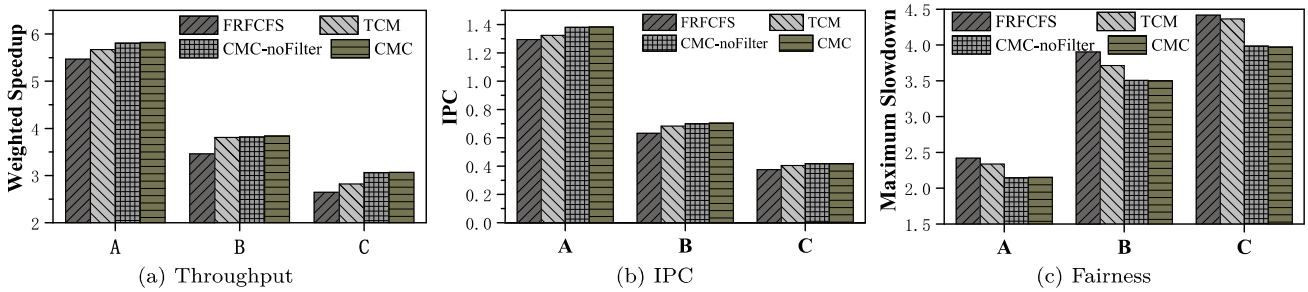

Figure 5. Average system throughput, IPC, and fairness of four scheduling strategies

Figure 5(b) shows the shared system IPC of workloads for the four scheduling policies. The system IPC is more intuitive to represent the performance improvement of workloads using the memory scheduling strategies. Compared with FRFCFS, TCM achieves 2.25%, 7.76%, and 7.97% system IPC improvement on workloads A, B, and C, respectively. CMC increases the system IPC by up to 6.99%, 10.66%, and 11.25%, correspondingly.

Our memory scheduling strategy is able to balance the fairness and system throughput by prioritizing memory-light applications and memory-intensive applications with a high AST. By prioritizing memory-light applications, both TCM and CMC reduce the average request servicing time in the memory controllers. Once these memory-light applications are serviced, the corresponding cores can continue to execute and avoid CPU starvation. However, since TCM uses MPKI to quantify memory intensity, this metric often leads to an inaccuracy of identification, such as the thrashing memory access pattern. In contrast, CMC uses the number of requests received by the memory controller in a period to identify memory intensity, it is more accurate than TCM, and thus achieves higher system performance improvement. For example, the average MPKI and MPKC of *omnetpp* is 0.37 and 5.3 for the same workload. Therefore, *omnetpp* is treated as a memory-light application according to TCM (MPKI < 1), while it is classified as a memory-intensive application according to CMC (MPKC > 3).

For memory-intensive applications, the improvement of system throughput and IPC is mainly attributed to preferentially scheduling the high-AST applications. These applications represent high AST due to low RBL (such as namd), low read/write ratio (such as bzip2), and low BLP (such as sjeng). CMC achieves higher system performance because it can identify competition-susceptible applications more accurately in hybrid memory systems.

6.4.2 Fairness

Figure 5(c) shows the maximum slowdown (lower is better) of the four scheduling strategies. CMC improves the system fairness by 7.97%, 5.74%, and 8.99% on average compared with the state-of-the-art TCM, respectively.

For most workloads, the four scheduling strategies can identify the same application that leads to maximum slowdown. Moreover, 77.8% of these identified applications have the highest AST in each workload. This implies that memory-intensive applications are apt to affect the system unfairness in hybrid memory systems. AST can accurately evaluate the memory interference in hybrid memory systems. By prioritize applications with high AST, CMC achieves higher system fairness.

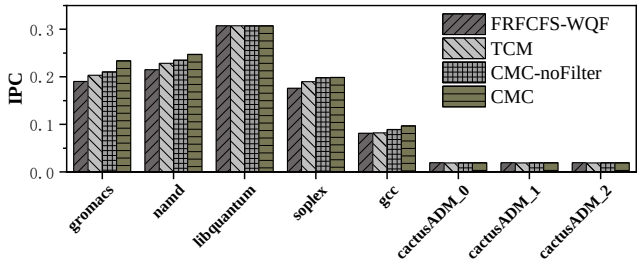

Figure 6. The effect of dead block bypassing

6.4.3 Effectiveness of Cache Filtering

It seems CMC and CMC-noFilter have little difference on the three metrics in Figure 5. We find that only cactusADM has an average RBL below the filtering threshold (RBL<0.1). The number of dead blocks in other applications is small. Due to only a small part of workloads contain cactusADM, there are only a few dead blocks in most evaluated workloads. To make more insight into how cache filtering contributes to LLC hit rate and system throughput, we analyze a workload that contains *cactusADM* in detail.

Figure 6 shows the IPC of each application in a dead-block-rich workload under four scheduling policies. To increase the number of dead blocks, this workload contains three *cactusADM*. Compared with FRFCFS, TCM, CMC-noFilter, and CMC improve system IPC by 3.85%, 6.86%, and 11.13%, respectively. When run together with other applications, *cactusADM* only gets 6.44% performance depredation and the IPC of *cactusADM* does not change much under the four strategies. It means bypassing dead blocks with a low RBL does not harm much to their performance. Compared with CMC-noFilter, CMC improves system IPC and hit rate by 3.99% and 8.03%, respectively. It indicates that if abundant dead blocks are bypassed, cache filtering is a good way to improve LLC hit rate and system performance.

6.5 Hardware Overhead

CMC needs some counters and shadow registers to track applications' runtime memory access behaviors. Table 3 shows the details of SRAM storage overhead for an 8-core system. Overall, the total storage overhead is less than 8KB.

Although the priority encoders [7] can be exploited to sort applications' memory requests according to AST, we note that it is unnecessary to schedule memory requests in strictly descending order of AST. We divide the memory-intensive applications into two groups according to the AST threshold. Applications with a high AST (over the threshold) are scheduled preferentially, and applications in each group do not need to be sorted. This simplification significantly reduces the storage overhead in memory scheduling.

Table 3. Storage Overhead of CMC

Counter/Register	Size (bits)
MPKC Counter	$N_{app} * \log_2(MPKC_{max}) = 40$
Read/Write Counter	$2 * N_{app} * \log_2(Request_{\max}) = 208$
BLP Bitmap	$N_{app} * N_{rank} * N_{bank} = 256$
BLP Counter	$N_{app} * N_{rank} * \log_2 N_{bank} = 96$
Shadow Register	$N_{app} * N_{rank} * N_{bank} * \log_2 N_{row} = 4096$
Row Buffer Hit Counter	$N_{app} * N_{rank} * N_{bank} * \log_2(Counter_{max}) = 3328$

7. CONCLUSION

Due to asymmetrical memory access latencies between DRAM and NVM, hybrid memory systems may significant exacerbate the memory scheduling fairness and application performance. In this paper, we propose a *cache/memory coordinated* (CMC) memory request scheduling strategy to address the fairness problem. We advocate *average service time* (AST) to evaluate memory interference among applications and prioritize memory-light requests and high-AST memory requests to improve the system throughput and fairness. Compared with state-of-art policy TCM, CMC improves system throughput and fairness by up to 8.76% and 8.99%, respectively.

8. ACKNOWLEDGMENTS

This work is supported jointly by National Key Research and Development Program of China under grant No.2017YFB1001603, and National Natural Science Foundation of China (NSFC) under grants No. 61672251, 61732010, 61825202, 61929103.

9. REFERENCES

[1] Renshuo Liu, Deyu Shen, Chialin Yang, Shunchih Yu, and Chengyuan Michael Wang. 2014. NVM duet: Unified working memory and persistent store architecture. *SIGPLAN Not.* 49, 4 (Feb. 2014), 455–470.

[2] Jishen Zhao, Onur Mutlu, and Yuan Xie. 2014. FIRM: Fair and high-performance memory control for persistent memory systems. In *Proceedings of MICRO*. 153–165.

[3] Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Ji Chen, Hai Jin, Yu Zhang, Long Zheng, Bingsheng He, and Song Jiang. 2019. Supporting superpages and lightweight page migration in hybrid memory systems. A*CM Trans. Architecture and Code Optimization* 16, 2 (April 2019), 11:1–11:26.

[4] Di Chen, Hai Jin, Xiaofei Liao, Haikun Liu, Rentong Guo, and Dong Liu. 2017. MALRU: Miss penalty aware LRU-based cache replacement for hybrid memory systems. In *Proceedings of DATE*. 1086–1091.

[5] Runze Han, Peng Huang, Yudi Zhao, Xiaole Cui, Xiaoyan Liu, and Jinfeng Kang. 2019. Efficient evaluation model including interconnect resistance effect for large scale RRAM crossbar array matrix computing. *Science China (Information Sciences)* 62, 2 (2019), 169–179.

[6] Hewlet Packard. 2017. HPE unveils computer built for the era of big data. [online] https://www.hpe.com/us/en/newsroom/press-release/2017/05/a-newcomputer-built-for-the-big-data-era.html.

[7] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of MICRO*. 65–76.

[8] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Tomas Moscibroda. 2011. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of MICRO*. 374–385.

[9] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of ASPLOS*. 335–346.

[10] Rachata Ausavarungnirun, Kevin KaiWei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 416–427.

[11] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2016. BLISS: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Trans. Parallel and Distributed Systems* 27, 10 (2016), 3071–3087.

[12] Xuchao Xie, Liquan Xiao, Dengping Wei, Qiong Li, Zhenlong Song, and Xiongzi Ge. 2019. Pinpointing and scheduling access conflicts to improve internal resource utilization in solid-state drives. *Front. Comput. Sci.* 13, 1 (2019), 35–50.

[13] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of MICRO*. 62–75.

[14] Junwhan Ahn, Sungjoo Yoo, and Kiyoung Choi. 2014. DASCA: Dead write prediction assisted STT-RAM cache architecture. In *Proceedings of HPCA*. 25–36.

[15] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. 2011. Bypass and insertion algorithms for exclusive last-level caches. In *Proceedings of ISCA*. 81–92.

[16] Priyank Faldu and Boris Grot. 2017. Leeway: Addressing variability in dead-block prediction for last-level caches. In *Proceedings of PACT*. 180–193.

[17] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of ISCA*. 60–71.

[18] Jahagirdar Sanjeev, George Varghese, Sodhi Inder, and Wells Ryan. 2012. Power management of the third generation Intel core micro architecture formerly codenamed Ivy Bridge. [online] http://www.hotchips.org/wpcontent/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf.

[19] Matt Poremba and Yuan Xie. 2012. NVMain: An architectural-level main memory simulator for emerging non-volatile memories. In *Proceedings of ISVLSI*. 392–397.

[20] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The GEM5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.