Morpheus: An Adaptive DRAM Cache with Online Granularity Adjustment for Disaggregated Memory

***Abstract*—Disaggregated memory introduces a cost-effective solution for improving in-memory utilization rate of data centers as a shared distributed memory pool among a number of individual servers. However, latency penalty in current connection between a computing node and the memory pool introduces performance degradation due to frequent far memory accesses. Based on our observation, page caching in the local DRAM, despite its reductions in the number of far memory accesses, still faces severe data over-fetching problem.**

With a detailed analysis of far memory access traces collected via several representative real-world applications, we argue that exploiting the various page-specific preference of caching granularity is the key point of solving the data over-fetching problem in DRAM caches. Consequently, in this paper, we present that it is influential to enable 1) dynamic selection of caching granularity for each page to not only guarantee sufficient spatial localities compared to the conventional fine-grained cache lines but also avoid data over-fetching caused by the coarse-grained pages, as well as 2) adaptive adjustment of cache capacity during execution for each granularity to accommodate the varying proportion of pages with different granularity preference. In specific, we propose Morpheus, an adaptive DRAM cache architecture that determines an optimal page-specific caching granularity at run-time and adjusts capacity occupations of different caching granularities dynamically. Based on our modeling and evaluations within the DRAMSim3 simulator, Morpheus exhibits 1.17-1.34x performance speedup for a wide range of workloads against the state-of-the-art DRAM cache design.

I. Introduction

When running applications with various resource requirements, the traditional data center faces the challenges of fixed proportionality of memory resources at design-time, and the memory utilization is usually as low as 65% [1]. With emerging techniques of high-speed network [2]–[4], memory disaggregation is proposed to separate memory from other resources as a shared huge memory pool, which has no or few computational units. The capacity of the memory pool is free-scaling and independent from other resources. Thus flexible amount of memory can be allocated to applications and be accessed through high-bandwidth network.

Unfortunately, the memory disaggregation system fails to achieve competitive performance, due to the microsecond-level latency of network which is 20-50 times longer than local memory. A common way to alleviate latency overhead is caching. The partial memory disaggregation systems, which store kernel data and cache application data in the small a-few-GB local memory, have been proposed to exploit locality to reduce the frequency of remote memory accessing. Among the existing partial memory disaggregation systems [5]–[11], the software approaches present either severe software-processing overhead or application non-transparency. On the other hand, the hardware DRAM cache designs tackle these drawbacks by managing local memory with hardware logic and relying on the cache coherence interconnects [12], [13] to track memory accesses at cache-line granularity.

The existing hardware DRAM cache designs mainly manage local memory with 4KB page granularity due to full exploitation of the available spatial locality in the applications and reduction of the size of cache metadata. However, when some pages are accessed in 64B line granularity for some time, the page granularity results in severe over-fetching problem, in which unused data is fetched and cached. It wastes network bandwidth and cache capacity, which ends up damaging system performance. For example, when running Memcached with a current state-of-the-art hardware DRAM cache for disaggregated memory, no more than 25% of data within the cached pages have been accessed before eviction.

In this paper, we implement partial memory disaggregation systems using different management granularities of DRAM cache and evaluate their performance running real-world applications (Section III). We demonstrate three key observations for designing the DRAM cache: Firstly, the conventional line (64B) granularity presents no performance advantages at the DRAM cache level in the memory hierarchy. Then, the page (4KB) granularity has the advantage of fully exploited spatial locality, while the block (256B) granularity is beneficial to temporal locality. Last but not least, the preferred management granularities for different time epochs of each page are different.

Based on above observations, we propose Morpheus (Section IV), a hardware DRAM cache attached to cache coherence interconnect, breaking through the conventional page-granularity-only management constraint. 1) To store data in different granularities, Morpheus splits cache into dual regions managed by block and page granularity respectively. Morpheus dynamically select the best-fitting granularity for each pages based on the observed spatial locality within the page in three miss situations. Morpheus monitors several system performance metrics to adjust the system's thresholds of recognizing a high-spatial-locality page at run-time using heuristic search. 2) To enable adaptive region capacity adjustment, Morpheus sets up one-level hash page tables storing address-mapping of both regions, whose capacity growth speed is determined by current data fetching behavior.

For evaluations (Section V), we implement Morpheus on the DRAMSim3 simulator [14]. We use HMTT [15], a hybrid memory trace toolkit, to collect memory accessing traces of

real-world applications on an Intel Xeon CPU platform, and inject these traces into the simulator. The evaluation results (Section VI) show that Morpheus presents 1.17x to 1.34x speedup on applications with both temporal and spatial locality compared to the existing state-of-the-art work [10] for disaggregated memory. Besides, Morpheus achieves competitive performance on applications with either temporal or spatial locality.

II. Background and Related Work

A. Partial Memory Disaggregation System

Apart from the advantages of free-scaling and high memory utilization, the disaggregated memory introduces new system parameters: **P1) Long network latency.** The network latency is 2-5 µs which is 20-40 times longer than local main memory. **P2) Limited network bandwidth.** The network bandwidth is typically 100-200 Gbps, while a single channel DDR5 memory bandwidth is around 300 Gbps. **P3) Unlimited capacity of memory pool.** The remote memory is independent of other nodes in the data center, which means its capacity is free-scaling and we can hypothesize it has infinite capacity.

In the new scenario, long data access latency is the main challenge for system performance. To alleviate such overhead, we propose three new optimization goals of partial memory disaggregation systems: **G1)** The number of remote data fetching should be minimized, to reduce impacts on performance due to long latency. **G2)** Each compute components only fetch the useful data to avoid wasting network bandwidth. **G3)** Compared to infinite but slow memory pool, fast but limited local memory capacity should be fully exploited.

There exists three main approaches to implement the partial memory disaggregation system. Figure 1.a shows the software runtime approaches. The paging-based approaches [5]–[7] depend on the virtual memory mechanism for 1) fetching remote data by detecting pages not present in the main memory using load page faults. 2) tracking dirty data by setting cached pages as read-only and invoking a store page fault. They need to extend Linux's swap subsystem to use remote memory as swap space, and to initiate conventional NICs to swap remote pages into local main memory. The APIs-based approach [8], [9] offers a library interface for applications to proxy any data movement from/to remote memory. As they can exploit application-level semantics, they are capable of fetching in object granularity (**G2**) and performing prefetching, with the penalty of lacking compatibility. The software approaches have drawbacks either on programming complexity or severe page faults overhead which takes up around 40% of overall latency [5]. Therefore, we focus on the hardware approach in this paper.

To achieve both compatibility and high performance, Figure 1.b shows a typical hardware approaches, which has a coherent interconnect attached card (FPGA or ASIC) connecting to CPU. The CPU has local main memory to store kernel data, and the card uses off-chip DRAM to cache data from remote memory. The card either has the network cable to directly fetch data from remote memory, or initiate NICs through the coherent interconnect to fetch data. The card exports a fake large physical address space to CPU, backed by remote memory (it may have private memory management [9]). Once the application accesses remote memory, the CPU sends a cache-line load request that will be routed to the card by the coherent interconnect. The card checks if the request hits in DRAM cache before fetching it from the remote momory. In addition, the card can observe the cache-line writebacks, and mark data as dirty for future eviction.

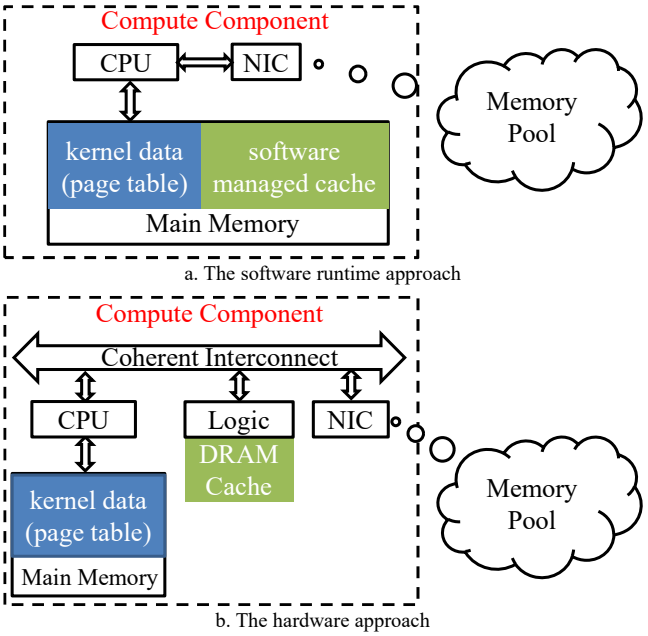

Fig. 1. Three main approaches to implement partial memory disaggregation systems.

B. Granularity Selection in the Hardware Approaches

Kona [10] organizes off-chip DRAM as a 4-way set associative cache, with per-way size equal to the page size. Kona writes back dirty data by aggregating the dirty cache lines into the log to save network bandwidth, and fetches data in page granularity. Although it reduces remote memory fetching (**G1**) using spatial locality within pages, it will over-fetch unused data which does not fulfill the **G2**.

To quantify how severe the over-fetching problem is, we evaluate the cache utilization (the average # of lines touched in a page before it is evicted [16]) of Kona. We study a comprehensive collection of real-world applications and benchmarks as listed in the Table III. We segment a page into 16 blocks (256B) and track the blocks touched before the page is evicted. We simulate Kona for 10M traces (around 1 second) and count the percentage of pages with the same number of blocks touched when evicted. As shown in the Figure 2, up to all pages in Memcache and down to 30% pages in IS are evicted with only 1-4 blocks touched.

Although managing DRAM cache with line granularity eliminates the over-fetching problem, it is hard to exploit any spatial locality in applications as most of the locality in lines has been exploited in higher levels in the memory

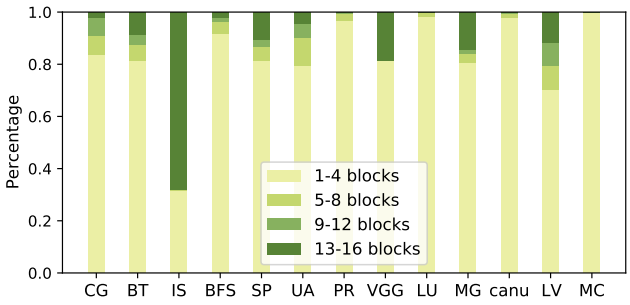

Fig. 2. Percentage of 4 categories, classified by the number of blocks touched when pages evicted (simulating Kona [10]).

hierarchy like CPU cache. As a consequence, frequent cache miss leads to performance degradation. Besides, the line granularity requires more capacity for metadata which increases the hardware overhead.

C. Multi-Granularity DRAM Cache in Hybrid Memory

A couple of researches in the hybrid memory scenario focus on tackling the over-fetching challenge faced by DRAM cache. Unison [17] and TDC [18] fetch only those blocks (64B) that will likely be touched during the lifetime of a page in the DRAM cache. Micro-Sector Cache [19] allocates a physical page (sector) to two logical pages, and the data is loaded and managed as micro-sector (128B) granularity in the physical page. They save the amount of moved data to some extent, but they still manage cache with page granularity, leading to plenty of empty memory bubbles in the cache which does not satisfy the **G3**. Not to mention that they rely on the high correlation between data and the instructions to predict useful data [16], which information is not reachable for DRAM cache attached on the coherence interconnect. To summary, a novel DRAM cache design is demanded to have advantages of both granularities in the memory disaggregation scenario.

III. Motivation

To possess advantages of both granularities, we need to find out which is the favorite granularity for majority of pages of an application in the memory disaggregation scenario. To answer the question, we compare the system performance (the # of clocks running 10M traces) between three kinds of granularity configurations of DRAM cache (cache-line (64), block (256), page (4K)). We choose the 256B granularity because one of commercial coherent interconnect protocols, Compute Express Link (CXL) has pronounced the support of 256B mode in its 3.0 standard. For simplicity, we implement a direct-mapped DRAM cache, as the cache associativity has limited impact on performance when the cache capacity is large enough [20].

As shown in the Figure 3, we can categorize these applications into three types according to their favorite granularity. **1) Temporal locality only.** These applications achieve higher performance with the block granularity, as they have hot spots in pages that are accessed more frequently than adjacent data within the same page. For example, PageRank frequently visits

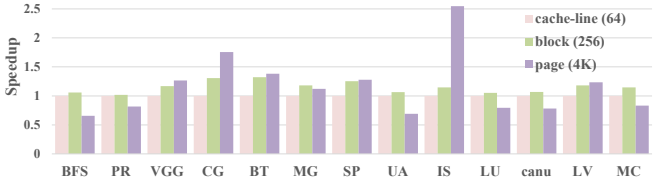

Fig. 3. Effect of the management granularity on system performance. The results are normalized to the *cache-line (64)*.

fat vertices in the graph, and Memcache queries hot key-value in the database. These applications' performance drop dramatically using page granularity, caused by severe over-fetching problem. **2) Spatial locality only.** These applications perform better with the page granularity, because they tend to sequentially access large continuous data. For example, the large matrix in the CG and the integer vector in the IS. **3) Both localities.** There exist both hot spots and continuous data in these applications, resulting in a similar performance with both granularities. We summarize the applications in the Table I.

TABLE I
THE LOCALITY IN APPLICATIONS

Categories	Applications
Temporal locality only	BFS, PageRank, UA, LU, canu, Memcache
Spatial locality only	CG, IS
Both localities	VGG, BT, MG, SP, LeViT

After investigating the behaviors and performance of three types of applications, we come to three key observations to direct our architecture design. **O1) Line granularity has no locality.** We find out that the performance of the block granularity is always better than the line granularity. **O2) All granularity configurations are useful.** The page granularity is beneficial for spatial locality and demands one fetching request per page (**G1**). On the contrary, the block granularity will avoid over-fetching (**G2**) and exploit temporal locality. **O3) Different pages favorite different granularity.** There is no consistent granularity suitable for every page. Moreover, the same page may present different locality at different time epochs especially when the system runs multiple applications simultaneously.

To design a DRAM cache based on the above observations, there are three main challenges. 1) How to manage data of different granularities? If we have data in cache managed in different granularities, it is hard to check cache only by physical address, as cache tags are part of physical addresses and their lengths depend on the cache granularity. 2) How to decide the granularity of one page? Attaching to coherent interconnect leads to the lack of code information which prevents us from predicting the optimal granularity based on high-level application semantics. 3) How to adjust the capacity proportion of each granularity? It requires elaborate designs on the dynamic cache mapping scheme, and decisions on which data to evict for free space.

IV. Architecture Design

A. Overview and Components

We design Morpheus as shown in the Figure 4. For the first challenge, Morpheus separates DRAM into *Block Region* and *Page Region* to support both block and page granularities. To dynamically select the suitable granularity for different pages, Morpheus fetches data in the block granularity by default and predicts the spatial locality of a page based on current memory accessing behavior. In specific, *Miss Handler* decides to fetch a page or block based on different thresholds in three miss situations. Morpheus collects run-time performance metrics for thresholds adjustment merely based on a couple of registers. To adaptively adjust each region's capacity, there is a *Hash Page Table (HPT)* for each region storing the mapping from the physical address to the cache address. We only set up the mapping when the data exists in the cache, and a cache miss happens when both address translations fail. Morpheus organizes two regions as stack and heap respectively, so that they compete for the left space of *Data Section*.

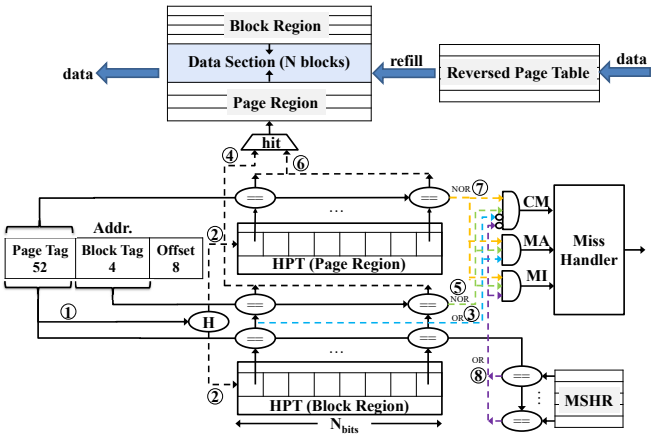

Fig. 4. Cache Layout. Three page tables and Data Section are stored in DRAM. The input signals to Miss Handler are: Cold Miss (CM), cache Miss with Adjacent blocks (MA), cache Miss with In-flight fetching (MI). solid lines are input data, and dashed lines are generated data.

B. Cache Operations

Region Organization. The *Block Region* is organized as stack which allocates entries from top to the bottom of *Data Region*, while the *Page Region* grows in the opposite direction as heap. The head of the *Block/Page Region* returns to the top/bottom of the *Data Section* right after they crash into each other.

Address translation. The structure of an entry in *HPT* is shown in Table II. Morpheus maps *Page Tag* to *HPT*'s index by the Murmur3 hash function ① [21]. We organize the *HPT* as n-way associative ($n = N_{bits}/(56+\log_2 N)$, where N is the size of *Data Section*). Morpheus reads out n-way in a single DRAM read ②, and checks every *Tag* in n entries to generate cache hit or miss signals.

Tag comparison. Morpheus uses two-level tag comparison logic in the block region. In the first level, adjacent blocks hit ③(blue dashed line) is triggered for any *Tag* equal to *Page Tag*. The lower 4 bits of *Tag* are compared to *Block Tag* in the second level. Morpheus generates a block region hit ④ when comparisons succeed in both layers, otherwise triggers a block region miss (green dashed line) ⑤. The page region only requires matching *Tag* with *Page Tag* to generate page region hit ⑥, or page region miss (yellow dashed line) if fails ⑦. Morpheus tests every *Tag* from entries in *MSHR* with *Page Tag* to find out whether any in-flight data fetching belongs to the same page and trigger a MSHR hit (purple dashed line) ⑧.

Cache hit. Morpheus concatenates *Cache Addr*$[\log_2 N - 1, 4]$ with <*Block Tag*, *Offset*> to read data from *Page Region*, while the <*Cache Addr*, *Offset*> is used to read from the *Block Region*. For data writes, the dirty vector which is stored in on-chip buffer is updated.

Three cache miss situations. Morpheus defines three kinds of cache miss using different combinations of miss signals. 1) cache cold miss (CM) refers to the very first fetching to a page, which happens when misses in both regions, adjacent blocks and MSHR are triggered. 2) cache miss with adjacent blocks (MA) happens when misses in both regions and adjacent blocks hit are triggered. 3) cache miss with in-flight fetching (MI) represents the situation with misses in both regions and MSHR hit.

Fetching one block or page? Morpheus chooses the fetching granularity based on the rule that the application inclines to access adjacent data within the high-spatial-locality page in a relatively short time. To control the system's preference of the fetching granularity, Morpheus sets two thresholds on the number of adjacent data observed in three cache miss situations. Morpheus will always fetch data in the block granularity for CM, as we have no hint to predict the page's spatial locality at this moment. For MA, Morpheus will fetch a page only if the number of adjacent blocks is greater than the threshold *PROMOTION_T*. We name this action as *Promotion*, and *PROMOTION_T* is adjustable from 0 to n. In the MI situation, Morpheus will fetch a page if the time interval between two adjacent fetching is less than the threshold *PADDING_T*. We name this action as *Padding*, and *PADDING_T* is adjustable from 0 to maximum network latency.

If Morpheus decides to fetch a page instead of a block, any adjacent blocks existing in *Block Region* will be evicted, and the old MSHRs will be removed. For eviction, Morpheus invalidate the *HPT* entry and temporarily stores the dirty data in a buffer which will be updated to *Page Region* after the page returned.

Data refill. The structure of an entry in *RPT* is shown on the Table II, which stores the physical address of data in the cache and is indexed with *Cache Addr*$[log_2 N - 1, 8]$. Once the data responses return, Morpheus looks up the *HPT* to find an empty entry. If a hash collision happens, Morpheus will refill data to *Cache Addr* of the collided entry. Otherwise, we refill the block/page to the head of the *Block/Page Region*, and look up the *RPT* for information about victim data. Then Morpheus writes information of the new data to *RPT* and *HPT*. If the victim data is dirty, Morpheus reads the victim block or

page and writes it back to remote memory.

Design Overhead. Morpheus trades latency for low miss rate. The latency gap between remote memory and off-chip DRAM is a least five times wider than fast memory and slow memory faced by traditional DRAM cache. It is worthwhile for Morpheus to take the cost of inserting several DRAM R/W into data path, to decrease miss rate to reduce the number of remote data fetching. Morpheus adds two *HPT* outstanding reads for tag comparison. For data refilling, one read and write for *RPT* and *HPT* respectively are required. To save DRAM bandwidth, a TLB for each *HPT* is implemented in Morpheus, and the evaluation (Figure 6) shows that most of the *HPT* accessing are eliminated. Morpheus reads *RPT* sequentially as a new block/page is always allocated from the head of *Block/Page Region*. Morpheus adds a *RPT* buffer of N_{bits} length so that we only need to read DRAM once every $(N_{bits})/(53+log_2 n)$ *RPT* entries.

TABLE II
THE STRUCTURE OF KEY METADATA

HPT entry			
Valid (1)	Tag (56)	Cache Addr ($\log_2 N$)	
RPT entry			
Valid (1)	Physical Addr (52)	HPT way index ($\log_2 n$)	
MSHR entry			
Valid (1)	Is Page (1)	Physical Addr (64)	Time (64)
Thresholds History Stack entry			
Valid (1)	PADDING_T (16)	PROMOTION_T (16)	Roll Back Times (8)

C. Heuristic Searching for Thresholds

To quickly discover the approximately optimal values of *PROMOTION_T* and *PADDING_T* at run-time, Morpheus implements the heuristic searching in *Miss Handler*, which decides how to adjust the thresholds based on several metrics. The system states and promised actions at each state along with their probability are shown in the Figure 5.

Morpheus splits the time into a series of monitoring windows. At the end of each monitoring window, Morpheus tags the current system as *Sub-optimal* or *Local Optimal* state based on sparsity ratio (SR) and average miss penalty (AMP), which are used to measure over-fetching and system performance respectively. *SR* refers to the ratio of evicted pages with less than 8 blocks touched, and *AMP* is calculated by cache miss rate $\times$ average remote memory accessing latency. At each state, Morpheus randomly chooses one promised actions, as it is hard to judge the potential performance gains.

In addition to actions adjusting thresholds, we define three more actions to speed up the convergence of searching. Considering the AMP of old thresholds may be inconsistent under the current monitoring window, Morpheus allows retrying the last thresholds' values by implementing a *Thresholds History Stack* with registers (as shown in the Table II). To stop the searching at any time if the current thresholds are worse than the best solution already found, Morpheus will roll back to the best thresholds in the *Thresholds History Stack*, and stop the searching for *Roll Back Times* windows, which increased

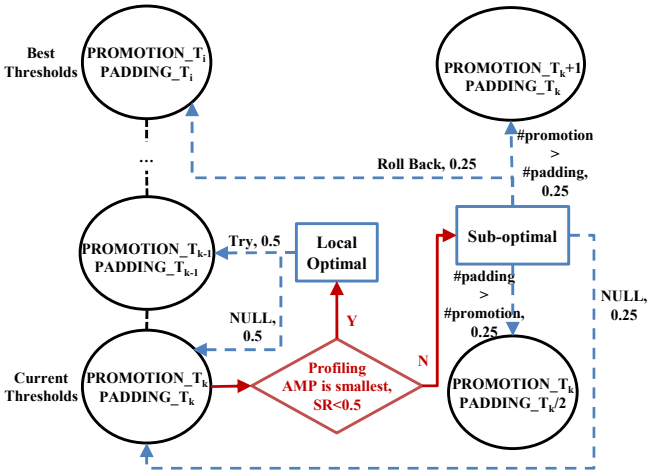

Fig. 5. The system states (blue square) and promised actions along with its chosen chance (blue dashed line). HSU stores limited thresholds' history (dark circle).

by one after rolling back. Morpheus does nothing for the *NULL* action, as some extreme cases may disturb the metrics, and longer monitoring time is needed to smooth out the interference.

The adjustment of two thresholds will affect the number of pages and blocks fetched, which will further have an impact on the growth speed of the two regions. Morpheus allows the fast-growing region to evict the space occupied by the other region. For example, when the system presents more temporal locality for some time, the *Block Region* grows faster so that its head exceeds the last crashing address, and the pages of *Page Region* will be evicted until the heads of the two regions crash again. In this way, Morpheus can adaptively adjust the space allocated to each granularity based on the characteristic of the application.

V. EVALUATION METHODOLOGY

We evaluate Morpheus using memory accessing traces of full real-world applications runs and a cycle-accurate trace-driven simulator. We leverage a hybrid memory trace toolkit [15] to collect memory accessing traces on the Intel Xeon E5-2620 CPU platform with 32G high-speed memory and all optimizations. We simulate the main memory, off-chip DRAM, and memory pool shown in the Figure 1.b using DRAMsim3 [14]. We implement in-house trace scanner to replay the traces collected on ideal memory and interconnect simulators to add latency and bandwidth constrains.

Comparison. In addition to the Morpheus, we implement two basic and one state-of-the-art DRAM cache (Kona [10]) for comparison. The two basic versions deploy direct-mapped DRAM cache with block and page granularity respectively.

Workloads. We use real-world applications and the NAS Parallel Benchmarks (NPB) [22] which will consume a large amount of memory, as shown in the Table III.

To control the simulating time within a reasonable range, we segment traces into a series of 10M-length trace windows. To highlight the effect of DRAM cache on performance, we

TABLE III
INFORMATION OF APPLICATIONS

Workload	Domain	Input Data	Trace Size (GB)
BFS [23]	Graph Processing	32M vertices × 512M edges	13
PageRank (PR) [23]			31
VGG [24]	Machine Learning	Flickr1024 [25]	47
LeViT (LV) [26]		imagenet200 [27]	151
Memcached (MC) [28]	Database	memtier [29]	4
canu [30]	Bioinformatics	Nanopore [31]	87
CG, BT, SP UA, LU	NPB	Class D	59, 51 59, 81 57
MG, IS		Class C	50, 62

choose the memory-intensive trace window, whose running time spans 0.1 to 1 second, to evaluate Morpheus. We warm up the simulator with traces before the chosen trace window.

TABLE IV
PARAMETERS OF SIMULATED SYSTEM

Trace Scanner	time window	256 ns
	max. outstanding	64
DRAM	DDR4,8Gbx4,3200MHz	
	64-entry read queue, 64-entry write queue	
	CL-tRCD-tRP-tRAS: 22-22-22-52	
	N_{bits}	512
Network	propagation delay	800 ns
	Bandwidth	100 Gbps
Coherent Interconnect	propagation delay	100 ns
	Bandwidth	500 Gbps
Morpheus	Block Region	HPT: N entries, 4-way. TLB: 1 MB
	Page Region	HPT: $N/16$ entries, 4-way. TLB: 64 KB
	Reversed Page Table	N entries
	Thresholds History Stack	16 entries
	monitoring window	10 ms
	Thresholds initial value	PROMOTION_T = 0 PADDING_T = 2000

System Configurations. We summarize the configurations of the simulator in the Table IV. We configure the *Coherent Interconnect* latency as CXL-Memory [12] accessing latency which adds around 50-100 nanoseconds of extra latency over normal main memory accessing [32]. The main memory, off-chip DRAM, and remote memory pool are configured with the same parameters, shown as *DRAM* in the Table IV. We reserve the length of the *Block/Page Region HPT* as same with the maximum number of blocks/pages in the *Data Section*. In this case, the hash collision rate may close to 100% and the cache allocation scheme rolls back to direct-mapped if one application only uses one region. However, it has little impact on performance as off-chip DRAM is large enough [20].

VI. Results and Discussion

A. Hardware Cost

In Morpheus design, we require less than 2MB SRAM to store the TLB of each *HPT*, and 4Kb registers to store MSHRs, *Thresholds History Stack*, and metrics in the *Miss Handler*. We store other metadata in the off-chip DRAM. According to the Table II and IV, the *Block/Page Region HPT* occupy 6.3%/0.4% DRAM, and the *Reversed Page Table* occupies 3.1% DRAM. In summary, the metadata will consume 9.8% DRAM. By only storing information related to the data existing in the DRAM cache, Morpheus is independent and capable of scaling to any remote memory implementations.

B. Impact on System Performance

We count the cycles used to simulate the trace window and show the speedup of systems normalized to *block (256)* in the Figure 6. When the capacity ratio equals $1/16$, for temporal or spatial locality only applications, Morpheus presents similar performance with *block (256)* or *page (4K)* respectively. Morpheus achieves 1.17x to 1.34x speedup or a geometric mean 1.28x speedup compared to Kona running applications possessing both temporal and spatial locality.

When we increase DRAM capacity, the speedup of *page (4K)* rises a lot on some applications. It is because the time pages remaining in cache prolongs, which will improve the cache hit rate. On the contrary, due to cache thrashing, the speedup of *page (4K)* drops significantly as decreasing DRAM capacity. Except for running BFS under the $1/4$ capacity ratio, the Morpheus presents similar performance gains under different DRAM capacity. We further investigate the thresholds changing history running BFS, and find out there is 15% reduction in the times of adjusting thresholds due to the condition $SR < 0.5$ remains true. As a consequence, Morpheus has fewer opportunities to adjust thresholds on some applications under large DRAM capacity.

Although the additional *HPT* looking up may worsen latency and compete for DRAM bandwidth, with the help of *TLB*, Morpheus eliminates at least 70% of *HPT* looking up, as shown in the Figure 6.

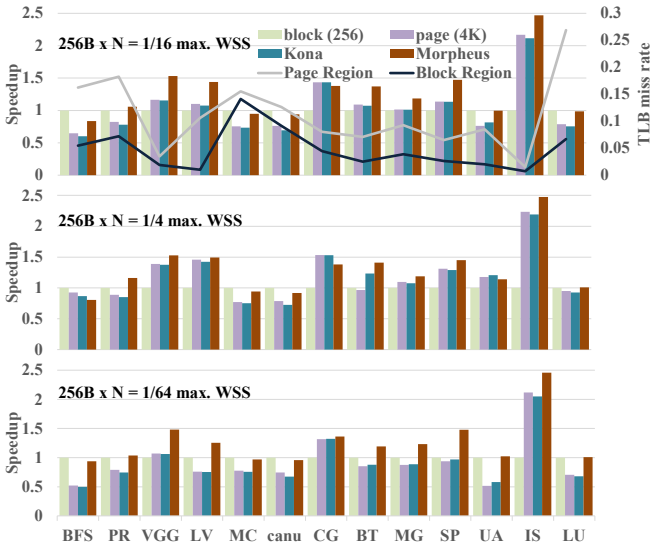

Fig. 6. The speedup of four systems under different off-chip DRAM capacity ratio compared to remote memory.

C. Impact on Cache Utilization

We count the ratio of evicted pages with the same number of blocks touched, as shown in the Figure 7. Compared to the

Figure 2, Morpheus improves the cache utilization remarkably, even for spatial locality only applications. Morpheus saves network bandwidth by 24.5% to 94.7% compared to *Kona*, which means Morpheus achieves better performance on these applications with fewer data fetched.

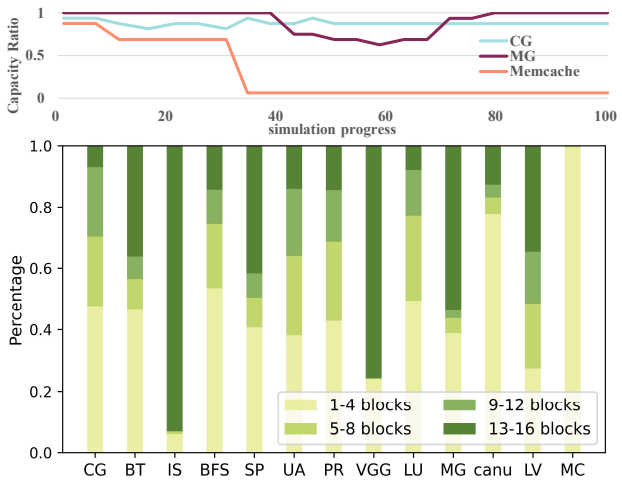

Fig. 7. Percentage of 4 categories, classified by the number of blocks touched when pages evicted. We show the capacity ratio of Page region's changing history on the top, running representative applications of three kinds.

Memcache presents low cache utilization because before searching converging to the situation where Morpheus only fetches blocks, a few pages will be fetched. At last, there is a few cache capacity left for the *Page Region* as shown on the top of Figure 7, so that the performance is close to the *block (256)*. For the spatial locality only applications like CG, the capacity ratio of the *Page Region* remains high. While the capacity ratio fluctuates up and down for the applications presenting both spatial and temporal locality like MG.

D. Performance Breakdown

To discover the impact of *Padding* and *Promotion* operations on performance, we conduct three experiments: fixing thresholds with their initial values, adjusting *PADDING_T* only, and adjusting *PROMOTION_T* only. According to the Table IV, we set initial values of *PADDING_T* and *PROMOTION_T* as 2000 and 0 respectively, which means Morpheus fetches a page when one adjacent block or one in-flight fetching exists. In the initial state, Morpheus intends to fetch more pages which makes it suitable for applications with spacial locality only. Therefore, we conduct experiments on other two application types, to test which threshold that Morpheus needs more urgently to eliminate over-fetching.

As shown in the Figure 8, the thresholds have different impacts on various applications. LeViT and MG are only benefit from the architecture of Morpheus, and the initial values do well for them. Adjusting *PADDING_T* or *PROMOTION_T* alone will improve the speedup on BFS and SP, while adjusting both thresholds will lead to sub-optimal performance. On the contrary, Adjusting both thresholds will further improve performance on Memcache, canu and BT, compared to adjusting single threshold. Other applications only favor adjusting *PROMOTION_T*.

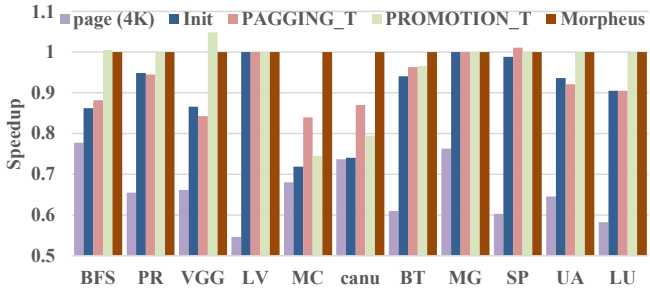

Fig. 8. The speedup of five system configurations normalized to Morpheus. Three new experiments: fixing thresholds with their initial values (*Init*), adjusting *PADDING_T* only (*PADDING_T*), and adjusting *PROMOTION_T* only (*PROMOTION_T*).

E. Sensitivity Study on Monitoring Window Length

The Figure 9 shows the speedup of Morpheus using different length of monitoring window, normalized to *10ms*. The shorter monitoring window, the more opportunities to adjust thresholds, which may has influence on the speed of searching convergence. We conduct experiments on applications whose performance will be affected by adjusting thresholds, according to the Figure 8.

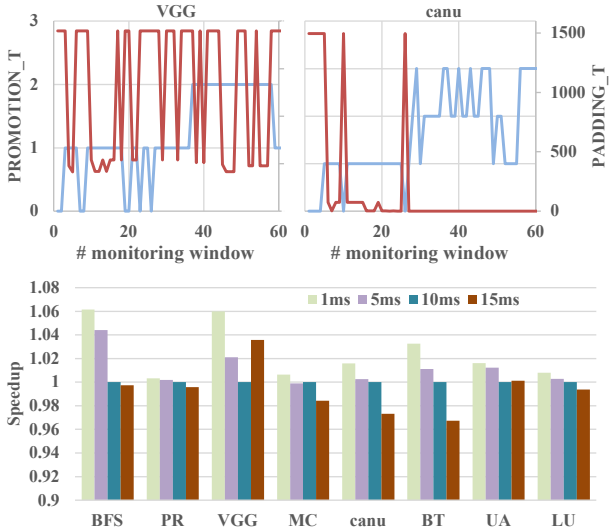

Fig. 9. The speedup of systems with different monitoring window length, normalized to 10ms (bottom). The thresholds' changing history of the first 60 monitoring window, running VGG and canu (top). The deep red line is PADDING_T, and the light blue line is PROMOTION_T.

For most of applications, Morpheus achieves better performance under shorter monitoring window, as their thresholds converge in a linear trend. For example, *PADDING_T* converges to 0 running canu (top right in the Figure 9), and stay steady during the rest of time. However, if the application presents a complicate trend, such as *PADDING_T* when running VGG (top left in the Figure 9), there is no obvious relationship between performance and monitoring window length. It is because Morpheus modifies the thresholds based on the observed history, which may be inconsistent with the applications' behavior in the next monitoring window.

F. Performance on Multi-Application

To evaluate Morpheus when *Compute Components* run multiple applications simultaneously, we set up four scenarios by mixing one spatial-locality-only application with one temporal-locality-only application, and randomly choosing applications from Ligra and NPB. As shown in the Figure 10, the performance is similar with *block (256)* on mix-Ligra, and Morpheus presents 1.02x to 1.34x speedup or a geometric mean 1.19x speedup compared to the *Kona*. Morpheus does not depend on applications, as it dynamically select the optimal granularity merely based on the observed memory accessing behavior.

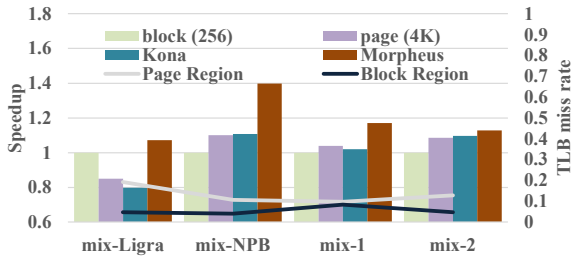

Fig. 10. The speedup of four systems running multiple applications. mix-Ligra: PageRank, Components, and MIS. mix-NPB: SP, CG, LU, and MG. mix-1: Memcache and IS. mix-2: CG and BFS.

VII. Conclusion

In this work, we propose Morpheus, an online-granularity-adjustment DRAM cache for disaggregated memory. Morpheus tackles the data over-fetching challenge with the block granularity while maintains the page granularity to exploit spatial locality. Morpheus dynamically selects optimal granularity for each page and adaptively adjusts cache capacity of each granularity. We use the memory accessing traces collected by running real-world applications and one well-known benchmark, on the Intel Xeon platform to evaluate Morpheus. The result shows that Morpheus presents 1.17x to 1.34x speedup compared to the Kona, and remarkably saves network bandwidth by 24.5% to 94.7%.

References

[1] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proc. of EuroSys*, 2020.

[2] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale," in *Proc. of SIGCOMM*, 2016.

[3] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network Requirements for Resource Disaggregation," in *Proc. of USENIX OSDI*, 2016.

[4] Gen-Z Consortium, "Gen-Z final specifications," https://genzconsortium.org/specifications.

[5] W. Yoon, J. Ok, J. Oh, S. Moon, and Y. Kwon, "DiLOS: Do not trade compatibility for performance in memory disaggregation," in *Proc. of EuroSys*, 2023.

[6] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proc. of EuroSys*, 2020.

[7] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *Proc. of USENIX NSDI*, 2017.

[8] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-Performance, Application-Integrated far memory," in *Proc. of OSDI*, 2020.

[9] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proc. of ASPLOS*, 2022.

[10] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proc. of ASPLOS*, 2021.

[11] C. Giannoula, K. Huang, J. Tang, N. Koziris, G. Goumas, Z. Chishti, and N. Vijaykumar, "Daemon: Architectural support for efficient data movement in fully disaggregated systems," *ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 1, mar 2023.

[12] CXL Consortium, "Compute Express Link specification revision 2.0," https://www.computeexpresslink.org/download-the-specification.

[13] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "Ibm power9 opens up a new era of acceleration enablement: Opencapi," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8–1, 2018.

[14] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[15] Y. Huang, L. Chen, Z. Cui, Y. Ruan, Y. Bao, M. Chen, and N. Sun, "HMTT: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, feb 2014.

[16] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *Proc. of Micro*, 2012.

[17] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *Proc. of Micro*, 2014.

[18] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, "Efficient footprint caching for tagless dram caches," in *Proc. of HPCA*, 2016.

[19] M. Chaudhuri, M. Agrawal, J. Gaur, and S. Subramoney, "Micro-sector cache: Improving space utilization in sectored dram caches," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, mar 2017.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[21] Adam Horvath, "MurMurHash3, an ultra fast hash algorithm for C# / .NET," https://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html.

[22] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[23] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. of PPoPP*, 2013.

[24] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[25] Y. Wang, L. Wang, J. Yang, W. An, and Y. Guo, "Flickr1024: A large-scale dataset for stereo image super-resolution," in *Proc. of ICCV Workshop*, 2019.

[26] B. Graham, A. El-Nouby, H. Touvron, P. Stock, A. Joulin, H. Jegou, and M. Douze, "Levit: A vision transformer in convnet's clothing for faster inference," in *Proc. of ICCV*, 2021.

[27] Y. Le and X. Yang, "Tiny imagenet visual recognition challenge," *CS 231N*, vol. 7, no. 7, p. 3, 2015.

[28] B. Fitzpatrick, "Memcached," https://www.memcached.org/.

[29] Redis, "memtier_benchmark: A high-throughput benchmarking tool for redis & memcached," https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/.

[30] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, N. H. Bergman, and A. M. Phillippy, "Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation," *Genome research*, vol. 27, no. 5, pp. 722–736, 2017.

[31] Kirkegaard, Rasmus, "Ecoli k12 mg1655 r10.3 hac. figshare. dataset." https://doi.org/10.6084/m9.figshare.11823087.v1.

[32] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "TPP: Transparent page placement for cxl-enabled tiered-memory," in *Proc. of ASPLOS*, 2023.