HyFarM: Task Orchestration on Hybrid Far Memory for High Performance Per Bit

Jing Wang*, Chao Li*†✉, Junyi Mei*, Hao He*, Taolei Wang*, Pengyu Wang*, Lu Zhang*,
Minyi Guo*†, Hanqing Wu‡, Dongbai Chen‡, Xiangwe Liu‡
* Shanghai Jiao Tong University, †Shanghai Qi Zhi Institute, ‡Alibaba Cloud
Email: {jing618, meijunyi, he-hao, sjtuwtl, wpybtw, luzhang}@sjtu.edu.cn,
{lichao, guo-my}@cs.sjtu.edu.cn, {hanqin.wuhq, dongbai.cdb, vicki.liuxw}@alibaba-inc.com

***Abstract*—Tapping into secondary memory resources, i.e., far memory (FM), has shown huge potential to improve the cost-efficiency of data centers. Recent advances in both storage-based vertical FM and network-based horizontal FM have raised new questions about leveraging hybrid FM tiers to achieve the best performance per bit of memory. It is still unclear how to efficiently place tasks when far memory access is enabled.**

In this work, we propose HyFarM, a novel task management strategy for hybrid FM clusters. We analyze FM sensitivity and cooperatively co-locate tasks to enable high utilization and scalability. Further, by tapping into dynamic memory adaption within and across servers, our strategy allows one to consistently deliver high performance on memory-intensive tasks. We evaluate our design with a heavily instrumented testbench. Compared with the state-of-the-art designs, HyFarM respectively improves memory utilization and the overall performance per bit (PPB) by up to 17.6% and 20.5%, with minor overhead.

***Index Terms*—memory disaggregation, hybrid, scheduling**

I. Introduction

Recent years have witnessed an important trend in data centers to leverage various secondary memory (i.e., far memory that are not placed in local DIMM slots) for data-intensive tasks [12], [14], [15], [31], [40], [49]. With disaggregated memory, servers can upgrade, expand, and scale memory in a more convenient way. Smartly enabling memory sharing also possesses huge potential to improve memory usage imbalance and save costs for data centers [3], [32], [33].

We classify the existing FM systems into two groups. The first is *vertical far memory* (vFM), as Figure 1-a shows. It taps into lower-layer storage-like persistent memory [40] and solid state drive (SSD) [31], [49] etc. The vFM generally offers larger capacity with lower cost. The other group expands memory capacity by using *horizontal far memory* (hFM), as Figure 1-b shows. In this case, applications can subscribe memory resources on a remote node through high-speed networks or specific fabrics like RDMA [12], CXL [2], OpenCAPI [7], smart-NICs [27], etc. So far, hFM is deemed faster than vFM [12], [15], [26]. For example, RDMA-based hFM has up to 20GB/s bandwidth [5] while commercial SSD-based vFM has up to 5GB/s [9]. Combining horizontal and vertical FM would provide a better design trade-off, leading to much more cost- and resource- efficient facility.

In this work, we are very interested in this question: *how to place and manage data-intensive tasks on hybrid far memory so that the full performance potential can be attained.* As

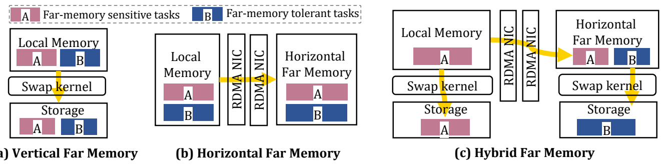
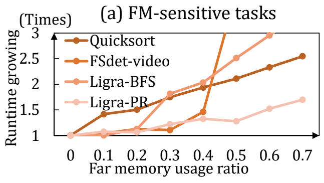


Fig. 1: Task placement on vertical, horizontal, and hybrid FM.

Figure 1 shows. In case that hFM has limited capacity, a task may access vFM and hFM simultaneously. FM-sensitive tasks are more preferable to use hFM than FM-tolerant tasks. By smartly distribute cold memory pages to both vertical and hFM, more critical data can be retained in the local main memory. With far memory, data centers now have the capacity and capability to shift data loads across nodes. What they lack is the visibility into far memory runtimes to understand the impact of memory access patterns on tasks and adapt quickly to the environment.

Developing a task management scheme for a hybrid FM is still an open problem. It is important to effectively harvest fragmented FM memory resources in the server cluster. Further, a more difficult challenge is that applications differ in their sensitivity to far memory access (detailed in Section II). Each task holds different proportions of hot pages to be accessed. It is non-trivial for the scheduler to determine the proper resource for tasks on hybrid far memory. We do not want to over-commit resources to meet the performance goals. Blindly throttling memory usage for sensitive tasks may lead to unpredictable performance degradation.

To date, disaggregated memory systems are still application-oblivious. While recent proposals allow one to seamlessly access remote memory pool [12], [19], [26], [31], [33], [40], they fail to capture the heterogeneity and dynamicity as application's memory behavior varies in a subtle way (detailed in Section II). When the local memory is inadequate and data must be off-loaded onto the far memory, current designs still use very basic memory management strategy: 1) *last-in-first-cap*, i.e., capping the memory footprint of the most recent scheduled tasks [34], [50] and 2) *first-in-first-cap*, i.e, capping the memory footprint of the earliest tasks [12], [31]. Both approaches only blindly squeeze the memory usage of the existing tasks to accommodate the workload.



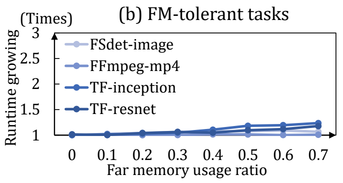
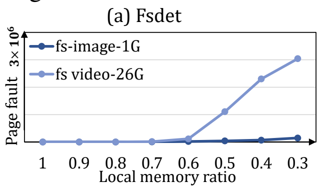
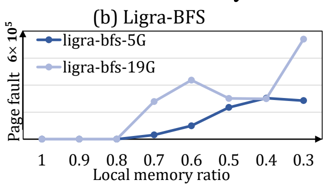
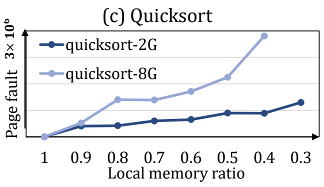


Fig. 2: Performance trend under different far memory ratios.







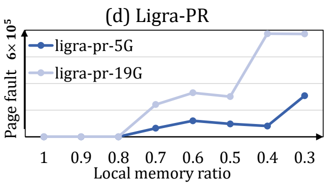
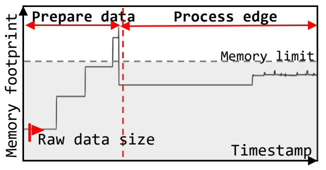
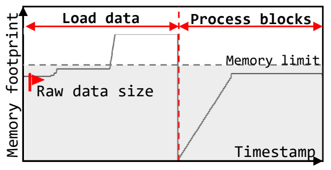


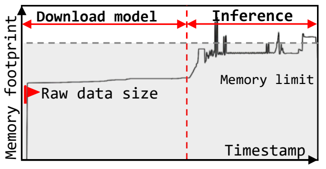
Fig. 3: Impact of different data size on the same application.



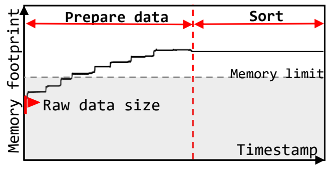
(a) Pagerank on Ligra



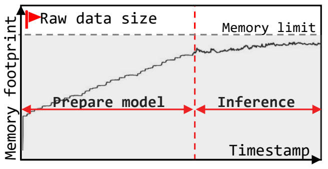
(b) BFS on Gridgraph



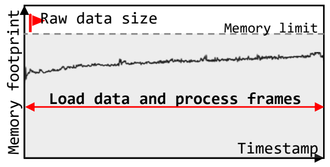
(c) Few-short detection on FsDet



(d) Quicksort on c++ std library



(e) CNN inference on Tensorflow



(f) Video processing on FFmpeg

Fig. 4: Different task behaviors in different execution phases.

There are two important recent works on designing a far memory management strategy. One is XMemPod [15], which is a hierarchical memory expansion framework for virtual machines. The other is TMO [49], a system mechanism that directly offloads cold data to heterogeneous SSD backends. Both schemes do not distinguish hFM from vFM. They lack the ability to manage far memory in a high-performance, application-specific way as well.

In this paper we answer two key questions: 1) how to place tasks on hybrid FM enabled clusters and 2) how to adjust far memory usage as task behavior varies. We present HyFarM, the first task orchestration scheme for hybrid far memory. The novelty of HyFarM is two-fold. First, it offers a resource-centric task placement approach that can serve diversified task demands with hybrid far memory. Second, it features an intra-/inter-node joint memory adaptation approach for balancing the memory usage at both the task and server level. Combing the two techniques allows one to achieve high performance per bit (PPB) in FM-enabled data centers.

This paper makes four important contributions:

- We take the first step to analyze the application performance on hybrid far memory.
- We propose HyFarM, a novel application-aware hybrid far memory management strategy that nicely fits into commercial shared-state cloud schedulers.
- We enhance HyFarM performance by devising an inter-/intra- node joint memory adaptation strategy.
- We validate our design with heavily instrumented evaluation environment. We show that HyFarM can greatly improve resource efficiency and performance.

The remainder of this paper is organized as follows. Section II further motivates our design. Section III proposes our task placement strategy. Section IV describes our fine-tuning mechanism. Section V details the evaluation methodology. Section VI presents the experimental results. Section VII discusses related works and Section VIII concludes this paper.

II. Challenges of Hybrid FM Management

In this section we present two challenges that makes hybrid memory management non-trivial.

Different Sensitivity: Depending on the application type and dataset size, some tasks are more sensitive to FM access. As Figure 2 shows, we analyze a group of popular tasks (detailed in Section V) on real system under growing far memory ratio (defined as FM usage divided by the total memory footprint). If we look at `FSdet-video`, `Ligra-BFS`, `Quicksort`, `Ligra-PR` and `FSdet-video`, their latency grows significantly when limiting local memory size. Differently, workloads in 2-b are tolerant to FM access. Most of their data can be offloaded to FM with small performance change. We investigate the page faults of FM-sensitive tasks (Figures 2-a,b,c,d) on different dataset sizes. As Figure 3 shows, `Quicksort`, `FSdet`, `BFS` and `PR` demonstrate very different page fault behavior when changing data size.

Changeable Sensitivity: Many tasks show coarse-grained phases on memory occupation during execution. This infers that a task may have different sensitivity as its phase changes. We profile six example tasks and present their memory usage trends over time, as shown in Figure 4. Note that the actual memory usage may be much larger or smaller than the dataset size (marked as red flags). We separate each task into two phases: the pre-processing phase and the execution phase. Figures 4-a,b,c show applications with obvious spikes. We can see that `Ligra-bfs` and `Gridgraph-bfs` workloads only show sensitiveness in the pre-processing phase while `FsDet` shows sensitiveness in the execution phase. Tasks in Figures 4-d,e,f show relatively smooth trends of memory occupation. They do not fluctuate heavily throughout their lifetime.

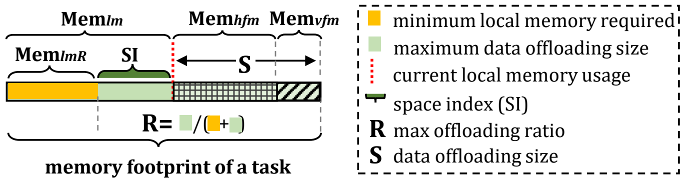

Mem*lm*
Mem*hfm*
Mem*vfm*
Mem*lmR*
SI
S
R= 🟩/(🟨+🟩)
memory footprint of a task
🟨 minimum local memory required
🟩 maximum data offloading size
current local memory usage
space index (SI)
R max offloading ratio
S data offloading size

Fig. 5: The memory usage of a far memory enabled task.

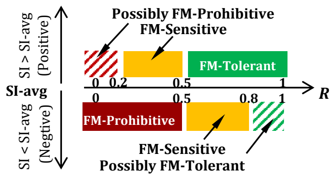


Fig. 6: Task classification.

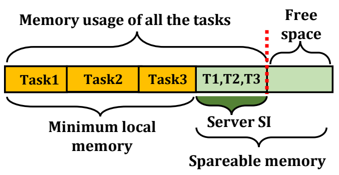


Fig. 7: Task co-location issue.

In summary, it is important to take into account the task's sensitivity to FM offloading ratio. Meanwhile, it can be problematic to make static FM scheduling decisions considering the dynamicity of applications.

Our design keeps the above observations in mind, combining light-weight offline profiling and adaptive online management. HyFarM performs FM-oriented task placement for meeting application requirement at minimum cost (Section V). It also features intra-/inter- node joint adaptation to boost the performance per bit of memory on hybrid FM (Section VI).

III. FM-Oriented Task Placement

The objective of task placement is to give a preliminary decision on a group of balanced tasks mapped to proper servers. We capture the task property and *group* tasks into three groups based on profiling results. We collocate tasks from different groups together and *map* them to proper servers.

A. FM-oriented Task Grouping

1) Task Profiling: For unfamiliar applications, we perform one-time profiling for understanding their sensitivity to FM data offloading. Our profiling generates far memory statistics from task/cluster logs and it does not require special tools or expensive learning procedure. We mainly consider two factors: *max offloading ratio* (R) and *data offloading size* (S).

Max Offloading Ratio. Different applications have varying degrees of latency tolerance. Aggressively offloading data from the local main memory to far memory causes undesirable latency. As shown in Figure 5, we define *max offloading ratio (R)* as the largest allowable far memory ratio that does not cause deadline violation. Recall that the far memory ratio is calculated as the amount of far memory usage divided by the total memory footprint requested by the task. The ratio is largely a property of the application program.

Data Offloading Size. Figure 5 illustrates the memory usage of a task. The *data offloading size* is the actual amount of data that are offloaded to FM (S = $Mem_{hfm} + Mem_{vfm}$). It is not only affected by the program, but also the data size. The range of actual data offloading size should be less than the product of memory footprint and max offloading ratio.

2) Task Assorting: When managing tasks on a FM system, we need to understand their space requirements and status of memory consumption, as shown in Figure 5.

We use *space index* (SI) to refer to the capability of an entity (e.g. a task or a server) to spare its memory. It is defined as the memory space that the entity can be offloaded at most without violating its performance objective. For each task, we can set a default SI quickly by examining the max offloading ratio and the data offloading size, as shown in Eq.1:

$$SI = R * Mem_{total} - S \tag{1}$$

where Mem_{total} is the total memory usage. The task SI value decreases if we de-allocate local memory (Mem_{lm}) from it. Figure 6 shows how tasks share the server memory. We measure a server-level SI (server SI) as the aggregated SI values of tasks co-located on it. A small server SI indicates that the server has limited potential to spare memory.

Finally, we categorize tasks based on their behaviors on the FM system. As shown in Figure 7, we first sort tasks in ascending order of SI and classify a task i according to an offset value $SI_{offset} = SI_i - SI_{avg}$. We then divide tasks into three groups based on their R values.

FM-prohibitive tasks. They are selected from those who have negative SI offset. We label candidate tasks with low R value (e.g., < 0.2) as FM-prohibitive. In general, short-lived, latency-sensitive tasks are often FM-prohibitive. It is not wise to offload FM-prohibitive tasks to FM and they also have little local memory to spare. Theses tasks need all their working sets to be stored locally when scheduling in HyFarM.

FM-tolerant tasks. They are selected from those who have positive SI offset. Tasks with large max offloading ratio and data offloading size are idea candidates for offloading. We label tasks with large R value (e.g., > 0.5) as FM-tolerant. Offloading data to different secondary memory (e.g., hFM or vFM) does not affect their performance significantly. It could result in fairly large data offloading size.

FM-sensitive tasks. These tasks can be tricky. We can treat all the remaining tasks as FM-sensitive tasks, but it may increase the burden of scheduling. On the other hand, we can classify tasks of medium R values (e.g., the yellow part in the figure) as FM-sensitive. Tasks with extremely large (low) R value will be treated as FM-tolerant (FM-prohibitive).

B. FM-oriented Task Mapping

Given the aforementioned analysis, our goal is to smartly assign a group of tasks to a FM-enabled server cluster.

To improve server utilization, we schedule tasks based on their core memory usage, i.e., the minimum required local memory Mem^i_{lmR} specified by the max offloading ratio. We first guarantee the minimum required local memory of the assigned tasks; we then provide extra local memory in a best-effort way. We formulate the task placement problem as a bin packing problem. We want to place tasks (objects) of different core memory sizes in servers (bins) to ensure minimum hosts

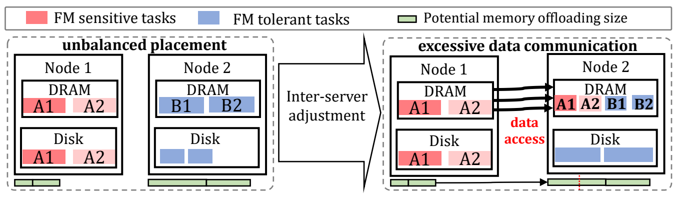


Fig. 8: Issues of unbalanced task placement.

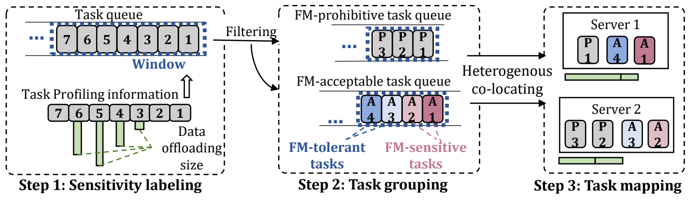


Fig. 9: The overview of task placement steps.

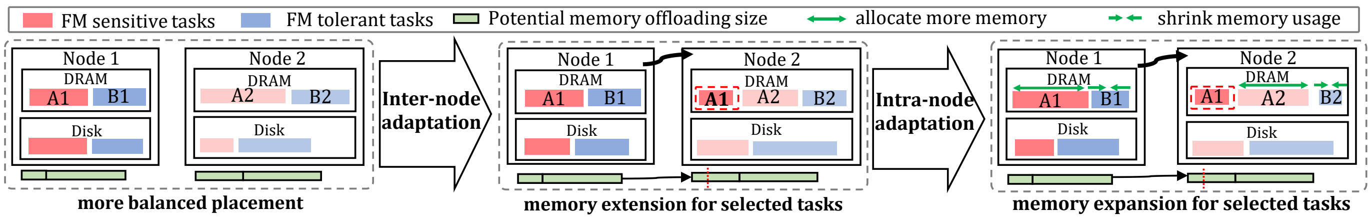


Fig. 10: A case of intra-node and inter-node adaption under balanced task placement.

are employed. Note that the mapping strategy is affected by resource pressure. If the resource is adequate, HyFarM will assign each task with its requested full memory size. If no server can fully meet the memory requirement of the task, we will scale down memory allocation in a task-specific manner (detailed in Section V). Under extreme memory pressure, we only assign the minimum required local memory to each task.

To improve scalability, we also carefully co-locate tasks from two complementary groups (FM-sensitive and FM-tolerant) at the stage of task mapping. Our key observation is that smart collocation of FM-sensitive tasks and FM-tolerant tasks is critical. Figure 8 gives an example of unbalanced placement. Suppose that tasks A1 and A2 are FM-sensitive tasks while tasks B1 and B2 are FM-tolerant tasks. All the FM-sensitive tasks are put on the node 1. When FM access is enabled, tasks on the node 1 will be allowed to access the memory of node 2, causing significant data communication overhead. In today's data centers, network bandwidth is a scarce resource that needs to be saved for serving user-facing latency-critical services. If we initially place different types of tasks on the same machine, we may save the bandwidth. In other words, FM-tolerant tasks can conveniently spare memory space for local high-priority tasks, without generating excessive network I/Os. One can better accommodate new tasks or handle increased memory demand.

Figure 9 shows the overall task placement process. We label tasks based on the profiling information and *group* tasks. We then *map* tasks of diversified performance sensitivity on a set of proper servers. Rather than allocating jobs one by one from the front of the job queue, we adopt the window-based scheduling [25] which allocates multiple jobs from a window at the front of the waiting queue.

IV. Enhancing Hybrid FM Performance

Given a group of balanced tasks mapped to a server, the next is to manage their appropriate usage of main memory (on local server), vFM (on local server), and hFM (on a remote server). As mentioned earlier, applications may switch between FM-sensitive and FM-tolerant types. It is important to fine-tune the memory usage effectiveness.

As shown in Figure 10, we leverage local intra-/inter- node adaption to tune hybrid FM usage. The process is driven by task-level events such as the detected phase shift of existing tasks or the start of a new task. These events can be collected by each node periodically with minor overhead.

A. *Inter-node Adaptation: Memory Extension*

Inter-node adaptation deals with *memory extension*, which allows high-priority tasks to extend their memory usage to hFM of another node. It aims at balancing memory usage pressure among nodes.

From the viewpoint of the controller, we need to decide which server should be shrunk (i.e., far memory providers) and which server should expand (i.e., far memory borrowers) its memory size. Afterwards, the far memory providers need to identify some to-be-shrunk tasks while the far memory borrower need to choose some to-be-extended tasks.

We use *server SI* to maintain a balance between servers. First, we calculate the overall SI of the cluster. We treat servers whose SI values are above the average SI as potential far memory providers. Correspondingly, the servers below the average SI are labeled as far memory borrowers. The provided (or borrowed) memory size is $|Server_{SI} - Avg_{Server_{SI}}|$.

In general, inter-node adaptation happens right after the tasks have been assigned to servers. For far memory providers, they just shrink their local memory usage to spare memory space. For far memory borrowers, only FM-sensitive tasks are selected to use hFM to accelerate their tasks.

B. *Intra-node Adaptation: Memory Expansion*

As shown in Figure 10, each server also allows for *memory expansion*. It can assign more main memory to FM-sensitive tasks and allowing the FM-tolerant one to dump part of its data to the associated vFM. HyFarM determines which tasks

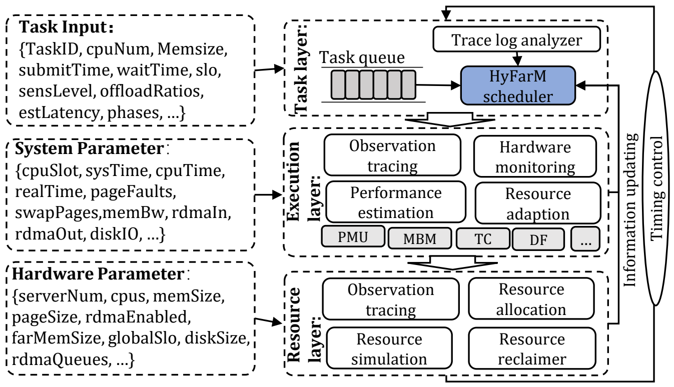


Fig. 11: An overview of our simulation framework.

need to balloon local memory usage for improving the overall performance on a particular server. The process can also be triggered if we observe drastic workload behavior change.

Under a stringent memory budget, it is important to make memory scaling plan for a group of co-located tasks. For example, if a server's local memory can not accommodate all the mapped tasks, an *uniform scaling* strategy will assign a uniform far memory ratio to these tasks.

Our intra-node adaptation gives priority to FM-sensitive tasks. Based on the core memory usage, we calculate how much additional local memory can each task earn. Considering that tasks with larger SI are less FM-sensitive, we tend to assign less local memory to them. It is reasonable to compress the memory footprint of the least sensitive tasks. In this case, one sacrifices a FM-tolerant task to its max offloading ratio and allows a FM-sensitive one to use local memory as much as possible. This minimizes the overall performance penalty. For a given task i, the actual local memory(lm) usage is the sum of its minimum required lm and an offset Δ_{lm} in Eq.2:

$$\Delta_{lm} \propto \frac{B_{lm}}{SI_i} \tag{2}$$

where B_{lm} is the remaining local memory budget calculated based on a given performance scaling target, such as 10%. The motivation behind Eq.2 is that more sensitive tasks will gain more extra local memory space for better overall performance.

V. Evaluation Methodology

Currently there is not such a framework that supports evaluation of task scheduling on FM-enabled data center. At this stage, we mainly focus on exploring the opportunities and benefits of task scheduling issue on such a hybrid FM system, while leaving detailed hardware/architecture support for future work. Therefore, in this paper we simulate a hybrid FM-enabled cluster and build a heavily instrumented testbench. Figure 11 shows the overall simulation methodology.

We implement the core part of HyFarM in 1300LOC in Python. we adopt hierarchical resource scheduling architecture like commercial off-the-shelf schedulers including Google's Omega [37] and Alibaba's Sigma [18]. We design our simulation procedure by referring to today's open-source HPC

Algorithm 1 Pseudocode of HyFarM.

```
 1: HyFarM_manager(tasks, cluster) {
 2:     tasks.group(tasks, cluster)
 3:     For tasks in cluster.task_pending_list  do
 4:     task.server_id = map(task, cluster)
 5:     task.server_id.inter_adapt(cluster)
 6:     while (is_task_finished or is_task_phase_changed) do
 7:         task.local_server.update_SI()
 8:         if (changed_SI $>$ Threshold) then:
 9:             inter_adapt(cluster)
10:         else: intra_adapt(task.local_server)
11:     server_list = sort_server_by_SI(cluster)
12:     for (each server_i in server_list) do
13:         if (server_i.SI $>=$ task.least_local_memory) then:
14:             server_i.add_task(task,task.local_mem) }
15: intra_adapt(server){ // adjust memory inside servers
16:     victim_tasks = estimate_best_ratio(tasks)
17:     task.local_server.reset_task(victim_tasks)
18: inter_adapt(cluster) { // adjust memory across servers
19:     provider = find_largest_SI_server(cluster.server_list)
20:     borrower = find_least_SI_server(cluster.server_list)
21:     provider .add_task(borrower_task)
22:     intra_adapt(provider) }
```

schedulers Slurm [8] and Gridscheduler [6], and and Torque [11]. We impose strict timing control and the minimum simulated time unit is 0.1 second, enough for a server cluster. We use realistic workload traces as input. Our input data traces not only contain job-level cluster usage data like prior work [35], but also take into account detailed FM system parameters. Our simulator can identify task phases by analyzing system events such as page faults and application checkpoints in the trace.

A. Simulation Workflow

We show our key scheduling workflow in Algorithm 1. We implement four stages in detail. i) Function `group` collects and calculates the attribute values (labels) of each task, such as least local memory size, SI value, sensitiveness, and so on. ii) Function `map` adopts bin-packing-based algorithms for solving the task placement problem. It takes task and cluster information as input and returns a proper server_id of each task. iii) Function *intra_adapt* mainly adjusts the to-be-allocated memory size of each task inside the server. iv) Function *inter_adapt* mark servers as either FM borrower or provider. It also determines the memory adjustment of each server. We use `intra_adapt` and `inter_adapt` to re-balance memory allocation if necessary.

B. Evaluation Setup

1) Hybrid FM Testbed: To obtain workload traces, we build both vFM and hFM systems and run various memory-intensive applications. Table I shows the key configurations of our far memory testbed. For vFM we use swap space on local disk. For hFM we use the Fastswap kernel [12]. Each node is provisioned with two 10-core Xeon CPUs, 128 GB of memory, 22TB of AVAGO MR9361-8i RAID with SATA I/O, and a Dual-Port Mellanox ConnectX-5 RDMA NIC supporting 40 to 100 Gb/s Ethernet. The RDMA driver we used is version 4.3.0

TABLE I: Testbed Configurations

Memory Type	Hardware	Bandwidth	Size	Environment
Local main memory	DRAM	200 Gpbs	128G	Linux OS
Horizontal far memory	RDMA	40 Gbps	<32G	Fastswap
Vertical far memory	Disk	2.5 Gbps	1T	Linux swap

TABLE II: Evaluated Applications

Abbr	Algorithm Description	Framework	Mem. Footprint
qs	quicksort	C++ std [1]	2G∼10G
bfs	breadth first search	Ligra [39]	5G∼20G
pr	pagerank	Ligra [39]	5G∼20G
mp4	mp4 format transcode	FFmpeg [4]	6G∼30G
mkv	mkv format transcode	FFmpeg [4]	8G∼30G
tf-i	tensorflow inception	Tensorflow [10]	4G∼20G
tf-r	tensorflow resnet inference	Tensorflow [10]	4G∼20G
fs-i	few-shot detection on images	FsDet [48]	1G∼10G
fs-v	few-shot detection on videos	FsDet [48]	10G∼30G

of the OFED kernel, and it uses the RoCE protocol. We collect the overall runtime using Linux Time, real-time memory usage using Intel VTune, memory access latency using PMU and record the page fault number using Linux perf.

2) Workload Traces: As shown in Table II. We profile a group of representative memory-intensive applications from open-source frameworks. We run each application with two raw source data (small and large). In total, we collect 18 traces from real machines. These traces show different memory footprints, page fault behaviors, and FM sensitivities. Based on the collected software/hardware events, we finally create three workload traces (task set) by randomly invoking them. We consider small (S-Trace), medium (M-Trace) and large (L-Trace) traces. Table III gives the configuration of the evaluated workload traces, each has a different number of tasks.

3) Baseline Schemes: We compare our design with several representative baselines including state-of-the-art works. We realize the key memory allocation strategies of these works in our simulator. In Table IV, `non-FM` implements a common HPC memory allocation method of CQsim [35]. In this scheme, far memory is not available or disabled. `LIFC` represents the Zswap mechanism [31], which is a VFM-based memory offloading method using the last-in-first-cap method when allocating memory resource. `FIFC` represents CFM [12], an HFM-based far memory scheduler which uses composed disaggregated memory nodes to offload data. It adopts the first-in-first-cap method when allocating local memory.

VI. Experiment Result

A. Overall Benefits of HyFarM

We start by assessing the overall performance of our task orchestration strategy. In general, our method shows significant improvement in the overall memory utilization, execution duration, and usage effectiveness.

Memory Utilization. Figure 12-a presents the memory utilization results of different schemes. Compared with `non-FM`, FM-oriented task placement can yield much higher memory utilization as a result of resource sharing. Our design shows high memory utilization than both `LIFC` and `FIFC` due to

TABLE III: Workload Traces and Cluster Configurations

Workload Traces	S-Trace	M-Trace	L-Trace
Task number	200	500	2000
Task characteristics	latency: 10∼6000s; dataset: 1G∼30G		
FM-sensntive task ratio	10%, 30%, 50%, 70%, 90%		
FM-prohibitive task ratio	10%, 20%, 30%, 40%, 50%		
Window size	10	20	50
Server memory	DRAM memory with 256G		
hFM size	32G at maximum		

TABLE IV: Baseline Methods

Evaluation	Reference	Memory Allocation Method
non-FM	CQsim [35], without FM	Kill tasks upon resource shortage
LIFC	Zswap [31], VFM only	Last-in-first-cap in proportional
FIFC	CFM [12], HFM only	First-in-first-cap in proportional
HyFarM	ours, HFM+VFM	SI-based HyFarM strategy

its resource-centric, sensitivity-aware task placement, which provides a flexible way to fit in more tasks.

Execution Duration. In Figure 12-b, we normalize the average task latency of baseline scheme `non-FM` to 1 and present the average execution duration of different schemes. We set a hard limit of 1.5× execution duration. Compared to the very conservative baseline `non-FM`, we observe a moderate increase in average task execution duration of about 19%-30%. Our scheme reduces the average latency by up to 16.5% compared to `LIFC` and 20.7% compared to `FIFC`. In fact, existing far memory scheduling schemes `LIFC` and `FIFC` can almost reach the 1.5× worst-case latency.

Performance per Bit. We study our design's memory usage effectiveness, namely, the system throughput divided by memory resource employed (i.e., performance per bit, PPB). In Figure 12-c, we normalized the result of `non-FM` to 1 and compare the PPB of different task management schemes. The result shows that our design improves the overall PPB by up to 52% compared with the `non-FM` baseline, up to 17.6% compared with `LIFC` and up to 20.5% compared with `FIFC`. This means that our intra-/inter- node adaptation provide better performance for FM-sensitive tasks. Our method delivers much higher throughput on the given resources. Note that HyFarM performs better under larger task set. It can efficiently harvest fragmented memory resources with various types of tasks as the system scales out.

B. Impact of Workload Composition

The optimization effectiveness of our design can be affected by the proportion (ρ) of FM-sensitive tasks in the cluster (in this case the share of FM-tolerant tasks is $1-\rho$). We change ρ in each trace and show the efficiency results in Figure 13. Overall, our method performs better than other baselines over a wide range of FM-sensitive task shares. Again, our system has better results when the task set becomes large.

We also observe that the proportion of FM-sensitive tasks influences the effectiveness of the evaluated schemes in different ways. As Figure 13 shows, `LIFC` and `FIFC` have the best performance when ρ is 0.7, while our method performs best when ρ is 0.3. This indicates that our method is more friendly to task sets with a relatively smaller number of FM-sensitive

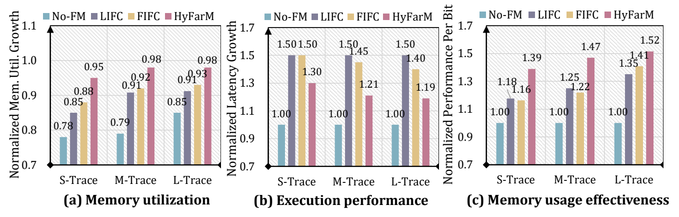


Fig. 12: Comparison of HyFarM to SOTA baseline schemes.

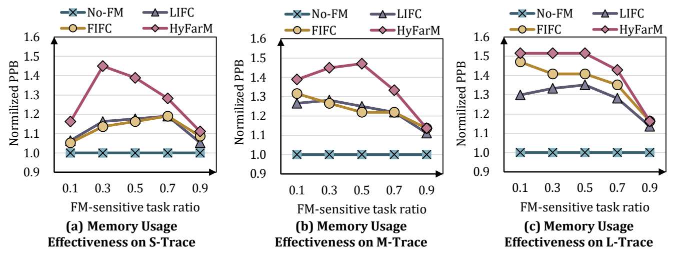


Fig. 13: Results under different FM-sensitive task ratios.

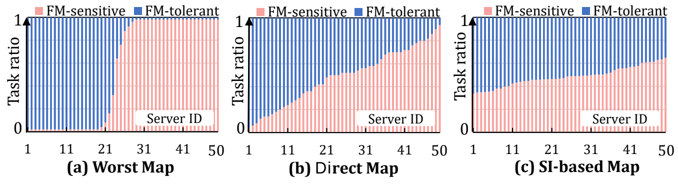


Fig. 14: The memory allocation behavior of task mapping.

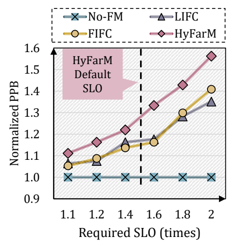
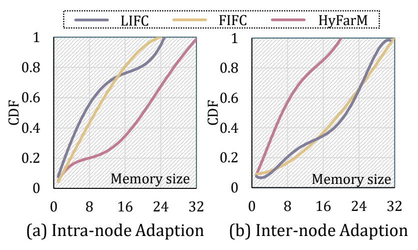



Fig. 15: PPB results Fig. 16: Memory usage distribution.

tasks for small-scale systems. In Figures 13-b and 13-c, our method offers the best results when FM-sensitive tasks account for half of the overall tasks. At this point, it can best balance tasks based on their FM-sensitivity. When ρ reaches 0.9, all the evaluated schemes show poor performance. This is because we do not have enough FM-tolerant tasks to spare memory; there are too many tasks competing for memory resources.

C. A Deeper Look at HyFarM

1) Impact of Sensitivity-aware Mapping: Figure 14 plots the distribution of different tasks. We evaluate various mapping schemes at scale. The x-axis is server IDs sorted according to the proportion of FM-sensitive tasks. i) *Naive*. It is a straightforward task mapping in which tasks with the same sensitivity are assigned together (Figure 14-a). Some servers are full of FM-sensitive tasks while others are full of FM-tolerant tasks. ii) *Random*. It is a method that directly mapping task to servers (`LIFC` and `FIFC`), as shown in Figure 14-b. The distribution of FM-sensitive tasks increases proportionally. iii) *SI-based*. We use a heterogeneous SI-based mapping to balance tasks (Figure 14-c). It is clear that our SI-based mapping can batter balance FM-sensitive tasks for servers.

2) Impact of Latency-tolerance Tasks: We examine the performance of our design by varying task's overall tolerance of latency increase, i.e. different Service Level Objectives (SLOs). Figure 15 shows the normalized memory usage effectiveness of the baselines and our works under various SLO limitations (maximum allowable execution duration, our default value is 1.5×). In general, our scheduler has much better performance. All the evaluated schemes show an increasing trend of effectiveness when the SLO value increases. The reason is that larger SLO value gives more space for far memory tasks to shrink its local memory usage. In general, FM-sensitive tasks favors a larger SLO value.

3) Impact of Joint Node Adaptation: HyFarM balances the memory allocation inside and across servers to further improve performance. We monitor the memory size adjustment after the task mapping step and present its distributions under different schemes. Figure 16-a plots intra-node adaptation while Figure 16-b plots inter-node adaption. The results show that HyFarM triggers more memory space adjustments due to intra-node adaptation. It gives more local memory space to FM-sensitive tasks. In contrast, the inter-node adaption of HyFarM adjusts much less local memory space than that of `LIFC` and `FIFC`. The results show that our HyFarM has lower communication cost brought by far memory usage.

D. Influence of FM-prohibitive Tasks

We analyze the effects of different proportion of FM-prohibitive tasks in the job queue. As shown in Figure 17, we compare two cases: 1) none of the tasks are FM-prohibitive and 2) half of the tasks are FM-prohibitive. The RDMA switches in our Infiniband network adopt static routes and they can maintain full bandwidth between every two server nodes. We can see increased bandwidth consumption if the FM-prohibitive tasks are few. We test the average task throughput of each server on different proportion of FM-prohibitive tasks and present the normalized results, as shown in Figure 18. Overall, HyFarM shows better task throughput than all the baselines. It is not surprising that the benefit of HyFarM falls as we add more FM-prohibitive tasks. However, even if over 50% tasks are FM-prohibitive, we can still yield 10% throughput improvement than the best baseline method.

E. Overheads Analysis

We analyze both scheduling overhead and configuration overhead to show the scalability and practicality of our system. The scheduling overhead refers to the time to make scheduling decisions. It depends on the cluster size since the searching space becomes larger. We test the scheduling time given different sever scales, as Figure 19 shows. Although HyFarM requires more time than the baseline methods, it can still

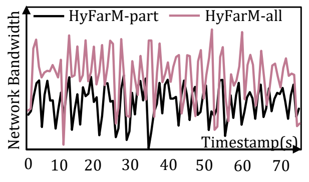


Fig. 17: Network usage

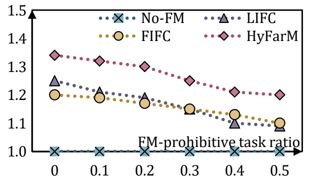


Fig. 18: Task throughput

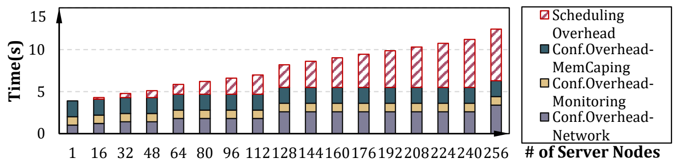


Fig. 21: The estimated overall overhead of HyFarM.

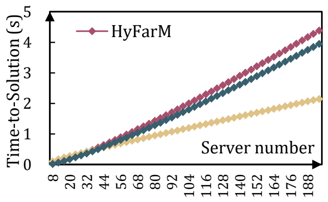


Fig. 19: Scheduling overhead.

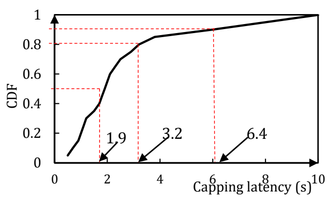


Fig. 20: Capping overhead.

satisfy the general latency requirement of HPC scheduling (15-30 seconds) [25]. The average scheduling time of HyFarM is less than 5 seconds on 192 servers.

The Configuration overhead refers to the time to take actions. As shown in Figure 20, half of the tasks spend 1.9 seconds in memory capping and 90% of the tasks spend less than 6.4 seconds. Phase monitoring and network communication can also cause delay. In Figure 21, we present the total latency of task allocation as the sum of the scheduling overhead and the configuration overhead, We estimate the network overhead by considering the switch latency of the tree-based network architecture. Overall, configuration overheads are positively correlated with system scale. The total control time is less than 14 seconds which is acceptable for HPC scheduling.

For very large clusters, we can use distributed scheduler architecture (each handles a medium set of nodes) [37] to maintain low overhead and high scalability.

VII. Related Work

Far Memory System. A high-performance FM system requires efficient software runtimes. The *hFM* uses RDMA to access the memory on another node. Some use OS swap kernel to handle data transmission such as Infiniswap [26], FastSwap [33], and Fastswap [12]. AIFM [36] and Kona [14] offload data in cache-line size for higher efficiency. Some works provide user-defined frameworks to gain high performance, such as Freeflow [30], FaRM [24], LITE [41], Fargraph [44], etc. The *vFM* takes local non-volatile memory and storage as far memory. Most of them are page-based, such as zswap [31] and Hybridswap [52]. Others adopt memory objects with variable size, such as pDPM [40], XMemPod [15], and TMO [49], etc. None of the above works addresses the problem of managing hybrid FMs. They also do not consider the performance sensitivity of co-located tasks.

Far Memory Scheduling. A few recent scheduling methods have taken FM into account. For example, CFM [12] calculates how much data can be retained in local memory. A Zswap-based software-defined far memory [31] presents a cold data compression strategy and swaps cold pages from local memory to FM (swap space in DRAM). XMemPod [15] adopts hierarchical far memory system and extends VMs' memory on virtual host memory, then on FM, and last on disk. The above works neither consider task sensitivity nor smart co-hosting. Allot [17] proposes a memory resource abstraction and data placement strategy for an RDMA-enabled distributed hybrid memory pool (DHMP) which manages hybrid NVM and DRAM memory efficiently. These works do not consider workload balance based on sensitivity and they do not support adaptive memory allocation.

Classic Cluster Scheduler. There are many works on efficient task management, such as open-source HPC schedulers [6], [8], [11], Google's Omega [37] and Alibaba's Sigma [18], etc. Schedulers like Borg [43], Quasar [22] and Paragon [21] run in a monolithic machine and process all jobs with the same logic. Two-level schedulers like Mesos [28] and Yarn [42] allow for application-specific scheduling. Omega [37] and Apollo [13] are shared-state schedulers that each scheduler maintains a copy of cluster states. There are also hybrid scheduler architectures [18], [20], [23], [29]. HyFarM is well complementary to these schedulers.

Memory-constrained Systems. There are plenty of works on running large-scale applications on memory-constrained systems. Some works [38], [45], [51], [53] analyze performance implications of classic memory-consuming workloads such as graph processing, AI model training, etc. Some works design specific memory access approaches based on read and write behavior and utilize data transfer channels to improve memory bandwidth utilization [16], [44], [47]. Some works design memory-saving techniques and data offloading strategies on limited local memory and far memory space [44], [46], [47]. HyFarM can fit well with existing systems and support excavating the performance and efficiency potential of memory-constrained applications.

VIII. Conclusion

The problem of improving the efficiency of far memory (FM) enabled parallel machines has recently attracted a lot of attention. We take the first step to explore sensitivity-aware task orchestration in a hybrid FM environment. We propose HyFarM, a high-performance task orchestration scheme for hybrid FM. It prudently collocates FM-sensitive and FM-tolerant tasks together for better efficiency. Importantly, we develop an performance optimization approach that combines both memory expansion and extension. Our design greatly improves memory utilization and performance per bit.

Acknowledgment

We thank all the anonymous reviewers for their valuable feedback. This work is supported in part by the National Natural Science Foundation of China (No.62122053 and No.61972247), and by Alibaba Innovative Research Program. Corresponding author is Chao Li.

References

[1] "C++ standard library headers," https://cppreference.com/.

[2] "Compute express link," https://www.computeexpresslink.org/.

[3] "Disaggregated memory," https://pmem.io/blog.

[4] "Ffmpeg," http://ffmpeg.org.

[5] "Infiniband architecture," https://developer.nvidia.com/networking.

[6] "Open grid scheduler," http://gridscheduler.sourceforge.net/.

[7] "Opencapi specification," https://opencapi.org/.

[8] "Slurm," https://github.com/chaos/slurm.git.

[9] "Solid state drives," https://www.samsung.com/us/computing/memory-storage/solid-state-drives.

[10] "Tensorflow," https://github.com/tensorflow.

[11] "Torque," https://github.com/adaptivecomputing/torque.

[12] E. Amaro, Branner-Augmon *et al.*, "Can far memory improve job throughput?" in *European Conference on Computer Systems (Eurosys)*, 2020, pp. 1–16.

[13] E. Boutin, J. Ekanayake *et al.*, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 285–300.

[14] I. Calciu, M. T. Imran *et al.*, "Rethinking software runtimes for disaggregated memory," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 79–92.

[15] W. Cao and L. Liu, "Hierarchical orchestration of disaggregated memory," *IEEE Transactions on Computers (TC)*, vol. 69, no. 6, pp. 844–855, 2020.

[16] Y. Cao, C. Li *et al.*, "Dr dram: Accelerating memory-read-intensive applications," in *International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 301–309.

[17] T. Chen, H. Liu *et al.*, "Resource abstraction and data placement for distributed hybrid memory pool," *Frontiers of Computer Science (FCS)*, vol. 15, no. 3, p. 153103, 2021.

[18] Y. Cheng, A. Anwar, and X. Duan, "Analyzing alibaba's co-located datacenter workloads," in *IEEE International Conference on Big Data (Big Data)*, 2018, pp. 292–297.

[19] E. Choukse, M. B. Sullivan *et al.*, "Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus," in *International Symposium on Computer Architecture (ISCA)*, 2020, pp. 926–939.

[20] P. Delgado, F. Dinu *et al.*, "Hawk: Hybrid datacenter scheduling," in *USENIX Annual Technical Conference (ATC)*, 2015, pp. 499–510.

[21] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, p. 77–88.

[22] C. Delimitrou, C. Kozyrakis *et al.*, "Quasar: Resource-efficient and qos-aware cluster management," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 127–144.

[23] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Annual Symposium on Cloud Computing (SoCC)*, 2015.

[24] A. Dragojević, D. Narayanan *et al.*, "Farm: Fast remote memory," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 401–414.

[25] Y. Fan, Z. Lan *et al.*, "Scheduling beyond cpus for hpc," in *High-Performance Parallel and Distributed Computing (HPDC)*, 2019, pp. 97–108.

[26] J. Gu, Y. Lee *et al.*, "Efficient memory disaggregation with infiniswap," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 649–667.

[27] Z. Guo, Y. Shan *et al.*, "Clio: A hardware-software co-designed disaggregated memory system," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[28] B. Hindman, A. Konwinski *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011, p. 295–308.

[29] K. Karanasos, S. Rao *et al.*, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *USENIX Annual Technical Conference (ATC)*, 2015.

[30] D. Kim, T. Yu *et al.*, "Freeflow: Software-based virtual rdma networking for containerized clouds," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019, pp. 113–126.

[31] A. Lagar-Cavilla, J. Ahn *et al.*, "Software-defined far memory in warehouse-scale computers," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 317–330.

[32] K. Lim, J. Chang *et al.*, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 37, no. 3, 2009.

[33] L. Liu, W. Cao *et al.*, "Memory disaggregation: Research problems and opportunities," in *International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1664–1673.

[34] P. Lotfi-Kamran, B. Grot *et al.*, "Scale-out processors," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 40, no. 3, pp. 500–511, 2012.

[35] D. Ren and W. Tang, "Cqsim," https://github.com/SPEAR-IIT/CQSim.

[36] Z. Ruan, M. Schwarzkopf *et al.*, "Aifm: High-performance, application-integrated far memory," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 315–332.

[37] M. Schwarzkopf, A. Konwinski *et al.*, "Omega: flexible, scalable schedulers for large compute clusters," in *European Conference on Computer Systems (Eurosys)*, 2013, pp. 351–364.

[38] C. Shao, J. Guo *et al.*, "Oversubscribing gpu unified virtual memory: Implications and suggestions," in *International Conference on Performance Engineering (ICPE)*, 2022, pp. 67–75.

[39] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Principles and practice of parallel programming (PPoPP)*, 2013, pp. 135–146.

[40] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-valuestores," in *USENIX Annual Technical Conference (ATC)*, 2020, pp. 33–48.

[41] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 306–324.

[42] V. K. Vavilapalli, A. C. Murthy *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Annual Symposium on Cloud Computing (SoCC)*, 2013.

[43] A. Verma, L. Pedrosa *et al.*, "Large-scale cluster management at Google with Borg," in *European Conference on Computer Systems (Eurosys)*, 2015.

[44] J. Wang, C. Li *et al.*, "Excavating the potential of graph workload on rdma-based far memory architecture," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2022, pp. 375–386.

[45] L. Wang, J. Ye *et al.*, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Principles and practice of parallel programming (PPoPP)*, 2018, pp. 41–53.

[46] P. Wang, C. Li *et al.*, "Skywalker: Efficient alias-method-based graph sampling and random walk on gpus," in *Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 304–317.

[47] P. Wang, J. Wang *et al.*, "Grus: Toward unified-memory-efficient high-performance graph processing on gpu," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 2, pp. 1–25, 2021.

[48] X. Wang, T. E. Huang *et al.*, "Frustratingly simple few-shot object detection," in *International Conference on Machine Learning (ICML)*, 2020, pp. 9919–9928.

[49] J. Weiner, N. Agarwal *et al.*, "Tmo: transparent memory offloading in datacenters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 609–621.

[50] D. Wentzlaff, P. Griffin *et al.*, "On-chip interconnection architecture of the tile processor," vol. 27, no. 5, pp. 15–31, 2007.

[51] M. Zhang, Y. Wu *et al.*, "Wonderland: A novel abstraction-based out-of-core graph processing system," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 608–621, 2018.

[52] P. Zhang, X. Li *et al.*, "Hybridswap: A scalable and synthetic framework for guest swapping on virtualization platform," in *IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[53] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX Annual Technical Conference (ATC)*, 2015, pp. 375–386.