Job Scheduling in High Performance Computing Systems with Disaggregated Memory Resources

Jie Li*, George Michelogiannakis†, Samuel Maloney‡, Brandon Cook†, Estela Suarez‡§, John Shalf†, Yong Chen*
*Texas Tech University, USA
†Lawrence Berkeley National Laboratory, USA
‡Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany
§Institute of Computer Science, University of Bonn, Germany
Email: *{jie.li},{yong.chen}@ttu.edu, †{mihelog},{bgcook},{jshalf}@lbl.gov, ‡{s.maloney},{e.suarez}@fz-juelich.de

***Abstract*—Disaggregated memory promises to meet growing memory requirements of applications while improving system resource utilization in high-performance computing (HPC) systems. Compared to traditional systems—where expensive resources such as CPUs, GPUs, and memory, are assigned to jobs in units of nodes—systems with disaggregated memory introduce memory pools that can be shared among jobs; this introduces new optimization metrics to the job scheduler. In this paper, we propose a data-driven approach to evaluate job scheduling and resource configuration in HPC systems with disaggregated memory. To incorporate the memory requirements of jobs for both local and disaggregated memory resources and improve system efficiency in open-science HPC systems, we introduce a novel job scheduling algorithm called *FM* (Fair Memory). Our simulation results show that *FM* outperforms commonly-used job schedulers in terms of jobs' bounded slowdown when the shared memory pool capacity is limited, and in terms of fairness under all conditions.**

***Index Terms*—HPC, Resource Utilization, Disaggregated Memory, Scheduling Policies**

I. Introduction

Traditionally, high-performance computing (HPC) systems have been designed with a tightly coupled architecture where compute and memory resources are bundled together in nodes. These nodes are statically configured and allocated exclusively to jobs for a period of time to avoid interference from other workloads. Therefore, node resources that are not used by the job assigned to those nodes are left idle. Node sharing among jobs only partially addresses this challenge and is typically much constrained in its use, and thus infrequent in practice. Also, HPC applications have intensely diverse memory requirements. Thus, combined with today's static resource allocation, HPC systems that serve a variety of scientific applications suffer in their utilization of expensive resources and face a challenge for efficient resource management [1]–[4].

In recent years, there has been a growing interest in disaggregated system architectures to manage memory and compute resources separately. This enables finer-grained allocation of resources to more accurately match application requirements. Disaggregated memory, referred to as *remote memory* for simplicity in this paper (in contrast to local memory on the same node as compute units), does not reduce the workload's memory requirements, but rather allows a compute unit to use unused memory resources on other nodes or in a common memory pool. However, this approach introduces new challenges, such as increased and heterogeneous memory access latency due to the remote location of memory resources, increased pressure on the network, and the need to allocate local and remote memory resources to jobs to balance application runtime, cost, memory utilization, and job queuing time among other goals. Therefore, job scheduling and allocating memory resources in a disaggregated system is more complex algorithmically because of the extra metrics and parameters, such as the physical location of remote memory modules and interference with other jobs in memory modules or the network.

While prior research evaluated how assigning remote memory to a job can affect its performance and explored implementations of disaggregated memory in HPC [2], [5]–[10], limited work has been done to either evaluate how the ratio of local and remote memory capacity affects the system and applications, or to co-design scheduling policies to balance the often conflicting goals of application performance and improving system-wide utilization of memory capacity.

In this paper, we aim to address three fundamental questions using a data-driven approach on two production HPC systems. First, in an HPC system equipped with disaggregated memory, is it helpful for the job scheduler to, in addition to existing considerations, consider the location and constraints of available *remote* memory resources? Second, what method should determine the ratio of local and remote memory pool capacities to minimize impact on application and system performance, and reduce total system memory? Lastly, what advantages does disaggregated memory bring to HPC systems?

The contributions of this study are summarized below.

- We present an application performance model that quantifies the impact of the additional latency incurred when accessing remote memory. We use this model to estimate job performance in a memory-disaggregated system.
- We simulate HPC systems with disaggregated memory resources using traces collected from two production systems and evaluate both system and job performance across various memory configurations.
- We present throughput per dollar spent on memory resources as an indicator of the cost-benefit ratio for disaggregated memory systems, and identify the optimal memory per rack for our two production systems.

- We introduce a remote memory-aware job scheduler, *FM* (Fair Memory), which outperforms the next-best state-of-the-art scheduler by up to 54 % in average bounded job slowdown. Additionally, FM achieves the highest fairness among all schedulers compared.

The remainder of the paper is structured as follows. First, we provide background on memory disaggregation and job scheduling in HPC systems in Section II. Next, we present our performance slowdown model in Section III. Then, we describe our evaluation results in Section IV. Section V summarizes previous work on scheduling and allocation policies. Finally, Section VI discusses future directions and concludes this paper.

II. Background

In this section, we discuss two common system architectures for memory disaggregation and present common scheduling techniques in HPC systems and their evaluation metrics.

A. Memory Disaggregation Architectures

Disaggregating memory involves separating the allocation of memory resources from compute resources, allowing for more flexible allocation and management of the memory resources. With disaggregated memory, different applications or workloads can be allocated varying amounts of memory based on their specific needs, rather than being allocated whatever memory capacity the application's allocated nodes contain.

Disaggregating memory in HPC systems can be achieved by either (i) logically partitioning, either in hardware or software, the memory capacity of a compute node into two parts, where the first part is used exclusively by the compute node itself while the second part can be utilized by remote nodes (either in the same rack or not) [11]–[13], or (ii) by instantiating a physically separate pool of network-attached memory while the compute nodes retain their local memory resources [5], [14]. Figure 1 illustrates an example of the latter architecture. Our study focuses on this architecture, instead of entirely separating compute and memory resources in different racks, because this variation allows for lower access latency to a subset of the shared memory pool. A noteworthy variation of this scheme is when compute nodes possess *zero* private memory capacity, making them entirely dependent on shared memory. In our study, all racks have the same configuration; each node has its own private memory and also shares an on-rack memory pool (what we refer to as 'rack-scale'), as well as off-rack disaggregated memory residing in other racks (referred to as 'system-scale').

B. Scheduling in HPC

Today, almost all HPC clusters use queuing systems for resource management and job scheduling, such as SLURM, UGE, and PBS Pro [15], [16]. These systems make available to users several queues with different resource constraints and priority levels. Within each queue, scheduling policies, such as *first-come, first-served* (FCFS), define each job's priority based on its characteristics. A job i has the following characteristics: (i) *Submit Time* s_i: the timestamp at which the job was submitted, (ii) *Estimated Duration* d_i: user or system estimated time to complete the job, and (iii) *Nodes* n_i: the number of nodes requested. In today's systems, users usually request a number of nodes of a particular type, which determines the number and type of compute units (CPUs or GPUs) and memory modules based on the configuration of the requested nodes. Some schedulers also take into account (iv) *Memory* m_i: the memory requested per CPU/GPU or per node (especially useful in case jobs are allowed to share nodes), and (v) *Waiting Time* w_i: the time that a job has been waiting in the queue. Other characteristics, such as *user ID* and *group ID*, may also be considered in the job's priority. However, to emphasize job-specific attributes and simplify analysis, we do not consider that specific users or groups have a higher priority than others in this study.

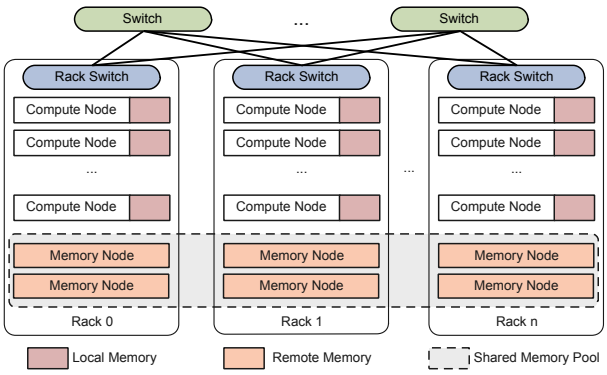


Figure 1: One example of providing a shared memory pool by distributing it across racks.

The scheduler's primary function is to allocate available resources to jobs in the queue. When there are insufficient resources to accommodate waiting jobs, jobs remain in queue. To improve resource utilization during these waiting periods, backfilling mechanisms are commonly employed. These mechanisms allow the scheduling of smaller jobs with lower priority to fill available system slots. The choice of which job to backfill can be made either by ensuring that no other waiting job experiences further delays (known as conservative backfilling) [17] or by not delaying only the job at the queue head (referred to as EASY backfilling) [18].

Scheduling policies are designed to optimize objective functions that align with the preferences of an HPC system. While there is no gold standard for evaluating scheduling policies, our study employs the following metrics that represent the goals of most HPC sites: system throughput, compute node utilization, average bounded slowdown, and fairness.

C. Evaluation Metrics

System Throughput: System throughput in HPC systems refers to the rate at which a system can process jobs. For a given time period of T, we count the total number of jobs N finished during this time and calculate the throughput as:

$$Throughput = \frac{N}{T}$$

Compute Node Utilization: Compute node utilization measures the percentage of time that compute nodes are actively processing jobs relative to the total time they are available for computation. This metric reflects the extent to which the computational resources within the HPC system are being effectively utilized. Over a given time period T, we sum the busy time b_i for each node i when it is allocated to jobs, and calculate the utilization for all nodes C as follows:

$$Utilization = \frac{\sum_{i \in C} b_i}{T * C}$$

Average Bounded Slowdown: Slowdown is the ratio of the time from job submission to completion (w_i+d_i) over the actual runtime duration (d_i), i.e., $\frac{w_i+d_i}{d_i}$. To avoid the disproportionate effect caused by exceptionally short jobs, bounded slowdown (*bsld*) is introduced as follows:

$$bsld_i = max\left(\frac{w_i + d_i}{max(d_i, \tau)}, 1\right)$$

where τ is a predefined lower bound that is typically set to 10 seconds [19]–[21]. Thus, the *average bounded slowdown* of N jobs is useful for comparing job performance in terms of job completion time ($w + d$) and is defined as:

$$avg\_bsld(N) = \frac{1}{N} \sum_{i \in N} max\left(\frac{w_i + d_i}{max(d_i, \tau)}, 1\right)$$

Fairness: We evaluate the fairness of schedulers by examining favoritism and discrimination among jobs. Following the methodology presented in [22], we analyze a stream of jobs $J_1, J_2, ..., J_N$ using *waiting time* as the comparison metric. We calculate the waiting time differences b_i for job i between the scheduler we use as a baseline (FCFS without backfilling) and the scheduler we evaluate. We categorize the b_i values of all jobs into three distinct sets: (i) positive values (indicating shorter waiting times under the evaluated scheduler) are grouped into the benefit group S_b, (ii) jobs experiencing performance deterioration are grouped into the discrimination group S_d, and (iii) jobs with no performance change are placed in the neutral group S_n. We can then calculate the total benefit B and total discrimination D as:

$$B = \sum_{J_i \in S_b} b_i, D = \sum_{J_i \in S_d} |b_i|$$

The associated fairness metrics are calculated as follows:

1) Marginal Discrimination (MD): the total discrimination in excess of the total benefits, calculated as $MD = D - B$.
2) Extreme Discrimination (D_x): the total discrimination values for the most discriminated proportion x of the jobs; for example, D_{10} and D_{20} represent the total discrimination for the top 10 % and 20 % most discriminated jobs, respectively.
3) Extreme Marginal Discrimination (MD_x): similar to marginal discrimination, calculated for the most benefited/discriminated proportion x of the jobs: $MD_x = \sum_{J_i \in S_d^x} |b_i| - \sum_{J_i \in S_b^x} b_i$.

The lower these metrics, the less discrimination is exhibited by the scheduler, indicating better fairness. It is important to recognize that while prioritizing shorter and/or small-scale jobs can mitigate system fragmentation and thereby enhance system throughput and compute node utilization, it often results in discrimination that is generally undesirable for jobs and users. Consequently, we incorporate fairness as a supplementary metric to balance our previous evaluation metrics.

III. Methodology

Memory disaggregation in HPC is an emerging research area, with no existing HPC systems fully capable of supporting this feature. Consequently, our study primarily employs simulation-based methodologies. This section introduces a performance degradation model to estimate job performance when utilizing remote memory. We then explain the collection of job traces from operational HPC systems. Using these traces alongside our performance degradation model, we simulate job performance under various remote memory configurations. In addition, we describe the simulated system configurations and present both baseline schedulers and our proposed scheduler.

A. Performance Prediction due to Disaggregated Memory

The performance of a job in an HPC system is influenced by a multitude of factors such as communication patterns, network congestion, and physical distance between compute tasks. With disaggregated memory, performance can additionally be influenced by the increased latency and reduced bandwidth between compute and memory resources. Previous studies have demonstrated that hardware that implements memory disaggregation can satisfy the maximum escape bandwidth of each memory and compute resource in today's HPC systems [23]. That said, as we reduce memory modules to reduce capacity, we also inevitably reduce available memory bandwidth. Therefore, a job may need to reserve more memory modules than strictly necessary for its capacity requirements, simply to satisfy its memory bandwidth requirements. However, previous studies showed that high memory bandwidth is seldom used by HPC jobs within the open-science NERSC workload [3]. Consequently, the effect of memory latency is dominant and *cannot be avoided* due to the longer physical distance of disaggregated memory [10], [23], [24]. Given that future memory technologies are expected to offer more bandwidth per module, thereby mitigating memory bandwidth concerns, our study focuses primarily on the impact of latency.

In our study, we use job traces from production HPC systems that include the start and end times of each job. These traces already encompass factors that affect execution time for the particular system and conditions at the time each job initiated. Therefore, for each experiment we configure our simulated system similarly to the system each trace is from and build a model for the additional performance penalty from disaggregation that focuses exclusively on capturing the performance degradation due to the added latency between compute and memory resources, a traditionally latency-sensitive path.

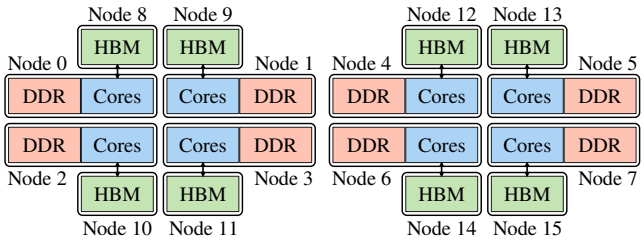


Figure 2: NUMA node/domain configuration of the two-Socket Intel Xeon Max 9462 Sapphire Rapids CPU in flat mode with SNC4 clustering, as used for latency sensitivity testing. This figure is adapted from Figure 14 in [25].

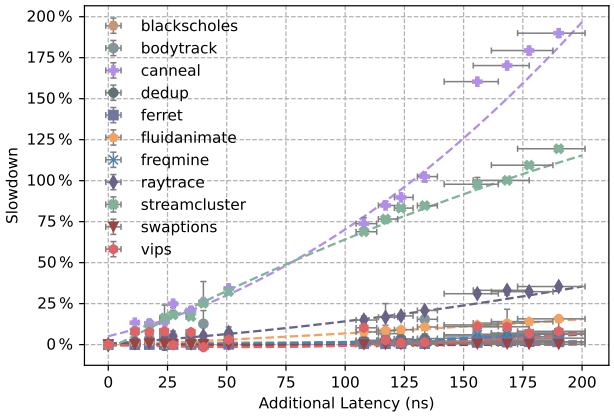


Figure 3: Performance slowdown of PARSEC workloads for different additional latencies between the LLC and the main memory. The dashed lines are the polynomial functions of degree 2 generated from the data points of each workload.

1) Sensitivity to Latency: We study the latency sensitivity of a variety of application kernels by executing the PARSEC3.0 benchmark suite [26] on Sapphire Rapids HBM nodes, equipped with dual Intel Xeon Max 9462 CPUs configured in flat mode and employing SNC4 clustering [25], as illustrated in Figure 2. PARSEC workloads contain a range of compute kernels and are representative of HPC applications and follow the trends of a previous study that evaluated two more application suites [23]. We execute each benchmark in single-threaded mode to focus on the effect of latency and avoid memory bandwidth from becoming a bottleneck; our goal is to measure on hardware, not simulation, the impact of memory latency on application performance.

To achieve a precise measurement of memory latency, we utilize the *numactl* utility to affinitize the execution context to NUMA domain 0, thereby standardizing the execution core across all tests. Subsequently, we vary the memory allocation across the 16 available NUMA domains to assess the latency impact from the perspective of domain 0. We use the Intel memory latency checker tool to measure the loaded latency between domain 0 and the other domains, under conditions where a single thread is responsible for generating memory traffic. The x-location of the data points in Figure 3 is the average latency measured across the range of memory traffic a single thread can generate, with the x-error bars showing the min/max measurements.

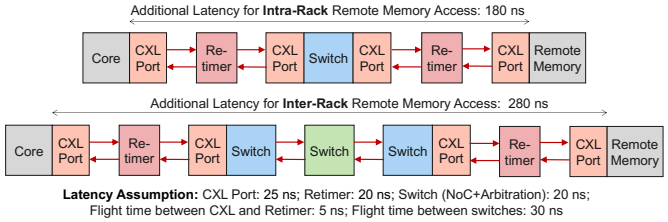


Figure 4: The additional latency for inter-rack and intra-rack remote memory access.

The PARSEC3.0 benchmark suite, with the exception of the *X264* and *Facesim* workloads that do not execute correctly, serves as the basis for our experiments. On each NUMA domain all of the workloads were run three times, with the median run used for the data point in Figure 3 and the other runs providing the y-error bar extents. As evident in the figure, the majority of the workloads within the suite show minimal sensitivity to variations in latency. However, a few workloads exhibit significant performance degradation in response to increased latency. Specifically, the *Canneal* and the *Streamcluster* workloads experience a slowdown of 190 % and 119 %, respectively, when subjected to an additional 190 ns of latency. The trends and orders of magnitudes of these results match other previous simulation-based studies, thus increasing the confidence of our conclusions [23].

The groupings of data points seen in Figure 3 are explained by the NUMA configuration. As shown in Figure 2, the CPU cores in each of the two sockets are divided into four groups, each associated with two NUMA domains, one for DDR memory and the other for HBM. There are only relatively small latency differences between NUMA nodes of the same memory type on the same socket, leading to four groupings of four points. The latencies on the local socket for the DDR memory (which has the lowest latency) and the HBM overlap to give the group of eight data points at the left of the figure; the HBM on the non-local socket has substantially more latency at single-thread bandwidths than the DDR memory, giving a noticeable separation between the two groups of four data points on the right of the figure.

2) Intra-rack and Inter-rack Latency: In our simulated HPC system, we assume a Dragonfly network where the nodes, including the computing and memory nodes within the same rack, belong to the same Dragonfly group; groups are interconnected in a fully connected graph. Each compute node connects through a single switch (i.e., a rack switch) to reach memory nodes in the same rack and traverses three switches (i.e., two rack switches and one inter-rack switch) to access memory nodes in other racks. Additionally, our system models the Compute Express Link (CXL) interconnect standard for memory disaggregation [10], [27]. We use the following latency assumptions for each component in this architecture,

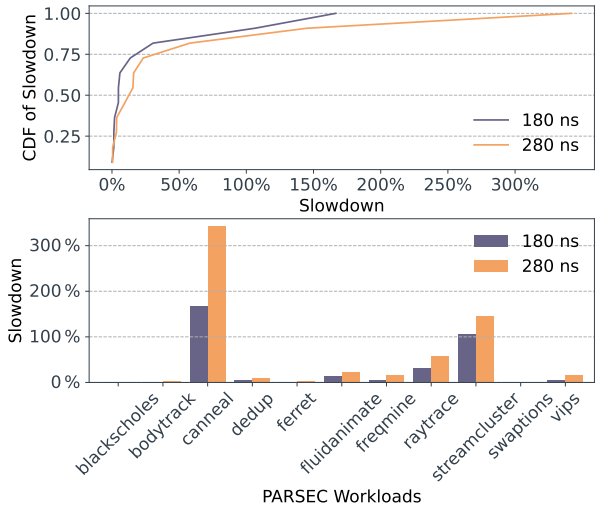


Figure 5: Bottom: The performance slowdown of PARSEC workloads for 180 ns and 280 ns of additional LLC memory latency. Top: The same PARSEC performance slowdowns presented as CDFs.

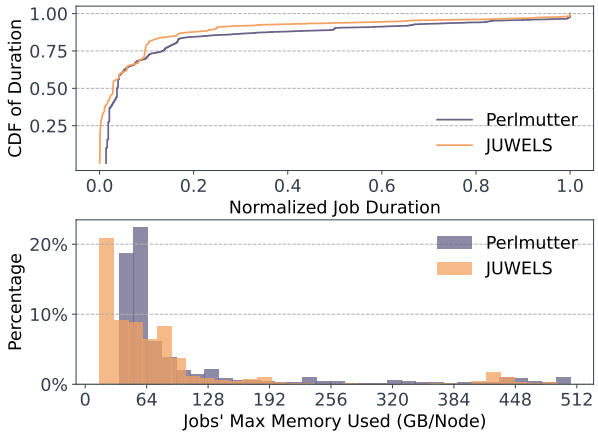


Figure 6: Bottom: The distribution of the maximum memory usage (GB/node) per job. Top: Normalized job duration distribution presented as CDFs.

adopted from [10]: each CXL port has a latency of 25 ns, each retimer has 20 ns, the total NoC and arbitration latency on each switch is 20 ns, the flight time between a CXL port and a retimer is 5 ns, and the flight time between switches is 30 ns. Therefore, the additional latencies for intra-rack and inter-rack remote memory access in our model are 180 ns and 280 ns, respectively, as depicted in Figure 4.

Drawing upon our latency sensitivity results, we then construct polynomial functions of degree 2, depicted by dashed lines in Figure 3. These functions are used to quantify the performance degradation at additional latencies of 180 ns and 280 ns for HPC workloads. The performance slowdowns are detailed in the lower section of Figure 5, where the baseline performance is when using memory on the same NUMA domain as the executing CPU core. For intra-rack remote memory access, an added latency of 180 ns leads to an average performance slowdown of 31 %, with the slowdown ranging from a minimum of 0.1 % to a maximum of 167 %. In the case of inter-rack access with an additional 280 ns of latency, the average performance slowdown is observed to be 53 %, with variations ranging from 0.75 % to 319 %.

3) System Performance Prediction Model: Because our system job traces inevitably lack application-specific information due to the scale and duration of those traces as well as privacy concerns, and because PARSEC contains a range of HPC compute kernels whose trends were confirmed by more benchmark suites [23] as we mentioned previously, we use the distribution of our aforementioned PARSEC performance slowdown results to model the slowdown of jobs in the system as a function of the additional latency to reach disaggregated memory. Figure 5 at the bottom shows the performance slowdown of PARSEC benchmarks for 180 ns and 280 ns and at the top the corresponding cumulative distribution function (CDF) for 180 ns and 280 ns. To capture the variability of job sensitivity to memory access latency as well as the effectiveness of latency hiding techniques such as remote memory page prefetching [13], [28], [29] for different applications, for each job in our trace, we assign a random value between 0 and 1 to model its performance sensitivity to memory latency. This value is then transformed into a slowdown factor (denoted as *sld_factor*) using the aforementioned CDF of the intra-rack (180 ns) or the inter-rack (280 ns) slowdown, depending on the disaggregation scope of each experiment.

The performance degradation model discussed above is based on the impact of additional latency in accessing memory resources. Fundamentally, the longer physical distance to reach remote memory makes higher access latency compared to local memory inevitable. In our simulations, we instantiate compute nodes that have local memory resources but can also utilize the shared memory pool. Because our production system traces cannot possibly capture each job's memory access patterns, we model a job's performance degradation as directly proportional to the "remote memory ratio" (denoted as *rm_ratio*), calculated by dividing the amount of remote memory allocated to a job by its total memory allocation. As expected, a job not utilizing remote memory ($rm\_ratio = 0$) incurs no performance penalty from memory disaggregation. Conversely, the more a job relies on remote memory, the greater the performance penalty it faces. The maximum penalty, occurring when $rm\_ratio = 1$, corresponds to the slowdown predicted by the aforementioned degradation model of Figure 5. An intermediate value of *rm_ratio* means the slowdown is only a fraction of Figure 5 because only a fraction of the memory accesses incur the additional latency to reach remote memory.

Therefore, a job's runtime on HPC systems with disaggregated memory can be modeled as follows:

$$original\_duration * (1 + sld\_factor * rm\_ratio)$$

TABLE I: Simulated system configurations. Perlmutter refers to CPU nodes whereas JUWELS refers to GPU nodes.

Parameter	Perlmutter trace	JUWELS trace
Total number of nodes	1536	960
Number of racks	6	20
Number of nodes per rack	256	48
Memory pool per rack (TB)	4, 8, 12, ... 48	2, 4, 6, ... 24
Node memory (GB)	64	
Number of warm-up jobs	3000	
Baseline scheduling algorithms	SJF, FCFS, WFP3, F1, FAIR	
Warming up scheduling	FCFS without EASY backfilling	

Note: The simulated system configurations are simplified and do not represent the actual configurations of Perlmutter and JUWELS. In particular, one of the racks in JUWELS has only 24 nodes, but here we assume them all uniform for ease of simulation.

B. Characteristics of Job Traces

In our study, we collected real job traces from the CPU nodes in NERSC's Perlmutter (referred to as the Perlmutter trace) [30] and the GPU nodes in the JUWELS Booster Module at Jülich Supercomputing Centre (referred to as the JUWELS trace) [31], [32]. These job traces were collected from SLURM and include actual memory usage metrics obtained from LDMS [33] on Perlmutter and LLview [34], [35] on JUWELS. The job traces contain the following fields:

1) *Submit Time*: This refers to the timestamp when a job enters the system queue. We preserve this field to reflect the real submission pattern observed in HPC systems.
2) *Duration*: This is the time it takes a job to complete once it starts executing. We utilize the aforementioned performance degradation model to adjust the job's duration based on the memory type (local or remote) it is allocated and the capacity of each type.
3) *Number of Nodes*: This is the number of nodes exclusively allocated to a job. We concentrate on node allocation instead of processor allocation as neither Perlmutter nor JUWELS currently allow sharing nodes between jobs. Thus, nodes are exclusively used by one job.
4) *Maximum Memory Used*: This denotes the maximum memory used per node across all nodes allocated to a job. We record the peak memory usage through LDMS or LLview to accurately reflect the real maximum memory requirements of jobs.

To ensure that our simulation experiments can be conducted within a practical time frame while still revealing scheduler behaviors, we select jobs submitted over three consecutive days, resulting in a total of 13 930 and 13 844 distinct jobs for the Perlmutter and JUWELS traces, respectively. Note that, the Perlmutter trace excludes jobs shorter than 10 minutes in duration while shorter than 10 minutes jobs are still present in the JUWELS trace. To speed up the simulation, we reduced both the job submission time and duration by a factor of 10, which accelerated the simulation without altering the results.

Figure 6 (bottom) displays the distribution of maximum memory capacity usage across the traces. The maximum memory usage per node for each job predominantly falls within the range of [32, 64) GB for the Perlmutter trace and [0, 32) GB for the JUWELS trace, which is considerably lower than the provisioned memory resources of 512 GB per node in both systems. The lower memory usage in the JUWELS trace is likely in part explained by the continued presence of jobs shorter than 10 minutes, as such jobs have less time to reach higher memory usage. To further analyze the duration of jobs, we calculate the normalized duration of each job by dividing its duration by the maximum duration observed across all jobs in each trace. The results are illustrated in Figure 6 (top).

TABLE II: List of Scheduling Algorithms.

Scheduler	Priority Function	Symbol Definitions
SJF	$-r$	r: duration
FCFS	$-s$	s: submit time
WFP3	$(w/r)^3 * n$	w: waiting time
F1	$\log_{10}(r) * n + 870 * \log_{10}(s)$	n: number of nodes
FAIR	w/r	

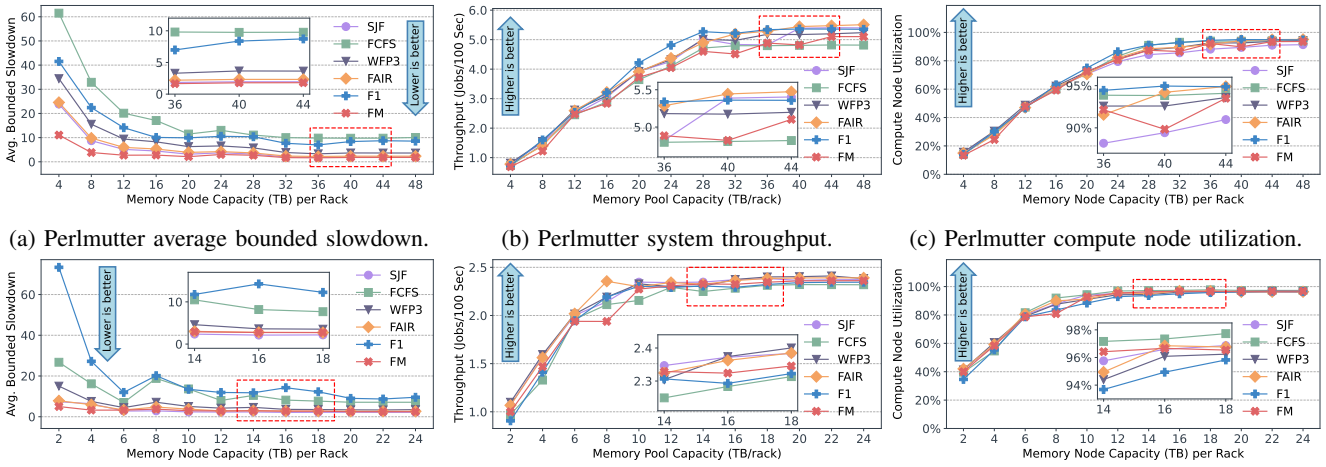
C. Simulated System Configurations

Table I presents the configurations of the simulated systems. We note that for JUWELS, the basic building block of the network is actually a 'cell', which for JUWELS Booster are composed of two physical cabinets; however, for simplicity we use the term 'rack' for both systems. For both systems, we reduce the node memory capacity from 512 GB to 64 GB to reduce system cost, compelling jobs with large memory requirements to utilize remote memory. The shared memory pool is configured on a per-rack basis. Additionally, we warm up each system with the first 3 000 submitted jobs, respectively, using an FCFS scheduling approach (without backfilling). After this initial stage, we transition to scheduling algorithms that incorporate EASY backfilling. The warm-up jobs and the jobs terminated after the start of the last job are removed from the evaluation to remove the warm-up and cool-down effect, ensuring that our analysis is based on data collected during the effective (stable-state) operation phase of each simulated system.

D. Baseline Scheduling Algorithms

Table II provides a list of scheduling algorithms used as baselines in our evaluation, explained below:

- *SJF (Shortest Job First)*: Selects and executes the job with the shortest requested runtime from the queue.
- *FCFS (First-Come, First-Served)*: Prioritizes jobs based on submission order, giving preference to earlier submissions.
- *WFP3*: Gives preference to both older and shorter jobs to prevent starvation of larger jobs [36].
- *F1*: A state-of-the-art scheduling algorithm developed through brute-force simulation and nonlinear regression, aimed at minimizing the average bounded slowdown [37].
- *FAIR*: Similar to WFP3, it prioritizes older and shorter jobs but does not consider job scale in its selection criteria.



(a) Perlmutter average bounded slowdown. (b) Perlmutter system throughput. (c) Perlmutter compute node utilization.

(d) JUWELS average bounded slowdown. (e) JUWELS system throughput. (f) JUWELS compute node utilization.

Figure 7: Performance comparison of various job scheduling algorithms for different memory pool capacities. Insets within the figures zoom in the areas enclosed by dashed rectangles.

For this study, we operate simulated systems in a non-preemptive mode, where jobs run to completion once they start executing, and enable EASY backfilling for all policies, leveraging its straightforward implementation and documented advantages [18]. We assume that the runtime requested by users accurately reflects the actual job runtime. This assumption allows us to focus on the fundamental differences between scheduling policies without the variability introduced by user-estimated runtimes, which can be influenced by account charging policies and upcoming deadlines.

E. FM: Novel Remote-Memory-Aware Job Scheduling

Scheduling algorithms in current HPC systems typically do not take into account a job's expected memory requirements. However, this factor could have a significant impact on job scheduling in memory-disaggregated systems. In light of this, we introduce *memory overload* (denoted as m) as a measure of each job's expected remote memory requirement, which is the memory occupancy beyond that of the local memory in the job's requested nodes. Memory overload is defined as 1 if the local memory on compute nodes is sufficient for the job. Otherwise, it is calculated as follows:

$$memory\_overload = \frac{job\_max\_memory}{compute\_node\_memory\_capacity}$$

In addition, we introduce a novel heuristic scheduling algorithm, called *FM* (Fair Memory), which takes into account the *memory overload* factor. The priority function of *FM* is defined as follows:

$$priority = \frac{w}{(\log_{10}(n)+1) * r * m}$$

It is worth noting that *FM* behaves identically to *FAIR* for jobs that request only one node and do not require remote memory. For other jobs, *FM* assigns lower priorities to those with longer durations, larger compute node requirements, and excessive memory demands. FM incorporates a job's memory requirement (expressed by memory overload m) into its priority function and tries to reduce memory pool fragmentation by prioritizing jobs with lower remote memory capacity requirements, while also considering the job's waiting time to avoid starvation.

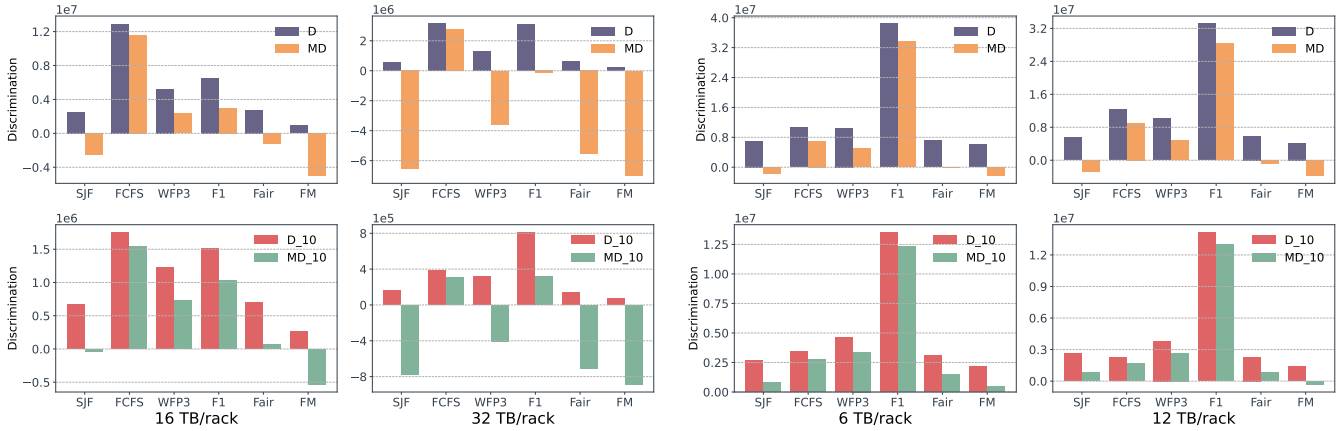
To illustrate, consider a simplified scenario where all jobs have identical compute demands and estimated durations (i.e., consistent values of n and r). Here, a job's priority is solely determined by its waiting time w and memory overload m. Under such conditions, for jobs entering the queue simultaneously, FM strategically assigns a lower priority to those requesting greater amounts of remote memory. While the hypothetical scenario described simplifies the explanation, it underscores the core logic of FM in prioritizing jobs to enhance memory pool efficiency. In addition, w in the priority function gives higher priorities to jobs with longer waiting times, effectively reducing the starvation of large-scale jobs and thereby improving fairness.

IV. Experimental Results

A. Performance Comparison of Schedulers

We conduct experiments to compare the performance of the baseline schedulers and our novel *FM* scheduler using rack-scale memory disaggregation with various memory pool capacity configurations. The shared memory pool in these initial experiments is only accessible to jobs running on nodes in the same rack, also referred to as intra-rack disaggregation [3]. The job placement policy of all these schedulers is to consolidate the allocated nodes in the same rack as much as possible while maintaining the load balance among racks, which is a common policy used in production HPC systems.

1) Average Bounded Slowdown: Figures 7a and 7d provide a comparative analysis of the job scheduling algorithms in terms of average bounded slowdown, where—as expected—the slowdown decreases as memory capacity increases, i.e.,



(a) Perlmutter trace.

(b) JUWELS trace.

Figure 8: Comparison of fairness metrics across different memory pool capacities: Discrimination (D) and Marginal Discrimination (MD) are shown at the top, while Discrimination (D_{10}) and Marginal Discrimination (MD_{10}) for the 10 % most discriminated jobs are displayed at the bottom. *FM* consistently displays the lowest values, indicating it maintains the highest level of fairness among the jobs.

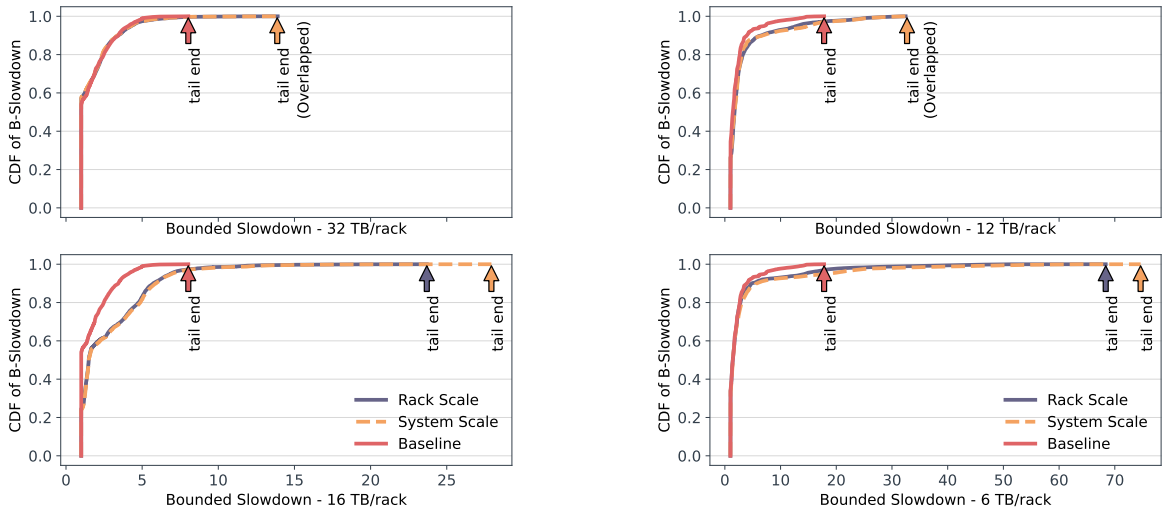
jobs experience shorter bounded slowdowns and have shorter response times. Across various memory pool capacity configurations in the Perlmutter trace, *FM* consistently boasts the lowest average bounded slowdown that is lower than the second-best *SJF* by up to 54 %. In the JUWELS trace, *SJF* performs marginally better than *FM* when the memory pool capacity reaches 6 TB/rack. However, in both traces, *FM* has a noticeably lower average bounded slowdown compared to the baselines when the memory pool capacity is limited. In such scenarios, *FM* delays jobs with excessive memory demands and prioritizes jobs with lower memory requirements. This approach allows the shared memory pool to accommodate more jobs, thereby reducing the waiting time for those preferred jobs.

2) System Throughput: Figures 7b and 7e illustrate system throughput, measured in jobs per 100 seconds, on the y-axis against the provisioned memory pool capacity per rack on the x-axis for the Perlmutter and JUWELS traces, respectively. Both figures display a similar trend: across all schedulers, as the memory pool capacity per rack increases, system throughput rises and eventually plateaus once the memory pool capacity reaches a certain threshold. This by itself is a valuable observation for HPC system procurement because it provides a methodology to determine a cost-effective capacity of system memory for a given workload. Also, given that the number of compute nodes remains constant across these experiments, the correlation between increased system throughput and enhanced memory pool capacity suggests that memory pool capacity is a potential bottleneck that hinders job scheduling when limited, despite the availability of computing nodes.

In terms of system throughput across different schedulers, when memory capacity is constrained (e.g., at 16 TB/rack for the Perlmutter trace), the difference in system throughput among the schedulers is not obvious for the Perlmutter trace. With such scarce memory capacity and relatively large memory requirements, schedulers have limited options to show a significant difference in system throughput. However, this difference becomes more pronounced for a higher memory pool capacity. Under such conditions, for the Perlmutter traces, the *FAIR* scheduling algorithm achieves the highest system throughput at 5.5 jobs per 100 seconds, while *FCFS* records the lowest at 4.8 jobs per 100 seconds. Similarly, the JUWELS trace records the lowest system throughput with *FCFS*. In both traces, our proposed scheduling algorithm, *FM*, demonstrates moderate performance when the memory pool capacity is sufficient.

3) Compute Node Utilization: The compute node utilization for the Perlmutter and JUWELS traces is illustrated in Figures 7c and 7f, respectively. Consistent with the trends observed in the system throughput analysis, compute node utilization increases with memory pool capacity and then plateaus once the memory pool capacity is sufficient. This further corroborates the previous finding that a limited memory pool capacity hinders jobs from running, and clearly shows that low memory capacity causes compute resources to be underutilized when memory is the scarce resource since jobs require a certain minimum memory capacity in addition to compute resources. The performance differences among schedulers are more obvious in the Perlmutter trace, where more jobs have longer durations and larger memory requirements compared to the JUWELS trace. For the Perlmutter trace at 44 TB/rack configuration, *FAIR* reaches the highest utilization rate of 95 %, whereas *SJF* shows the lowest at 91 %. For the JUWELS trace, *FCFS* achieves the highest compute node utilization at 98 %, and *F1*, although it performs the least effectively, still manages a utilization rate of 96 %. Among all these schedulers, *FM* has moderate performance, similar to the system throughput analysis.

System performance, characterized by system throughput and compute node utilization, often conflicts with job performance,



(a) Perlmutter trace.

(b) JUWELS trace.

Figure 9: CDF of bounded slowdown for rack-scale and system-scale memory disaggregation, for sufficient (top) and limited (bottom) memory pool capacity.

as indicated by the average bounded slowdown. In traces like Perlmutter's, which contain more jobs with longer durations and larger memory requirements, *FAIR* generally exhibits the best system performance. In contrast, our proposed scheduler, *FM*, diverges from *FAIR* by assigning lower priority to jobs with larger memory demands through the *memory overload* factor. This approach may delay jobs that could otherwise alleviate resource fragmentation, thereby potentially hindering system performance.

4) Fairness: Figure 8 presents a fairness comparison using the metrics D, MD, D_{10}, and MD_{10}. Since these metrics evaluate the level of discrimination (decision biases based on job characteristics) jobs receive from the schedulers, a lower discrimination value indicates greater fairness. Figure 8a demonstrates that *FCFS* exhibits the highest discrimination for Perlmutter while *FM* shows the lowest discrimination in both configurations, according to D and MD metrics. When considering the discrimination and marginal discrimination of the 10 % most discriminated jobs, specifically D_{10} and MD_{10}, *F1* appears less fair in the 32 TB/rack configuration. However, across both configurations and all four metrics, *FM* consistently displays the lowest values, indicating it maintains the highest level of fairness among the jobs.

We repeat the fairness analysis on the JUWELS trace, as depicted in Figure 8b. In both memory configurations, *F1* exhibits the highest discrimination levels, whereas *FM* consistently shows the lowest for the metrics D, MD, D_{10}, and MD_{10}.

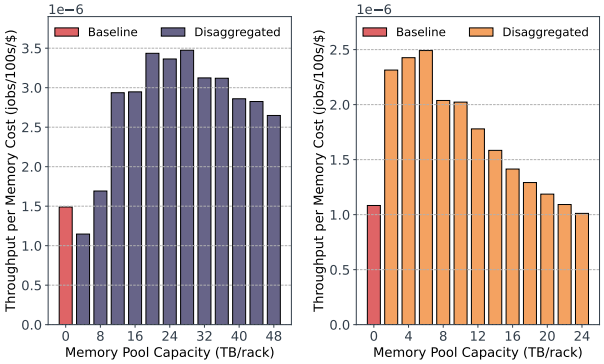
In summary, for all our evaluations so far and although scheduler performance varies among different job traces with distinct characteristics, *FM*—being the only scheduler that considers the remote memory property in its algorithm—consistently exhibits a lower average bounded slowdown when memory pool capacity is limited. It also shows comparable average bounded slowdown to other schedulers when memory pool capacity is sufficient. *FM* also significantly enhances the fairness of the scheduler across all configurations and job traces.

B. Memory Disaggregation Scopes

In the subsequent experiments, we employ *FM* as the scheduling algorithm for all simulations. We compare the performance of different memory disaggregation configurations against a baseline configuration where the systems retain their original memory setup without utilizing memory disaggregation.

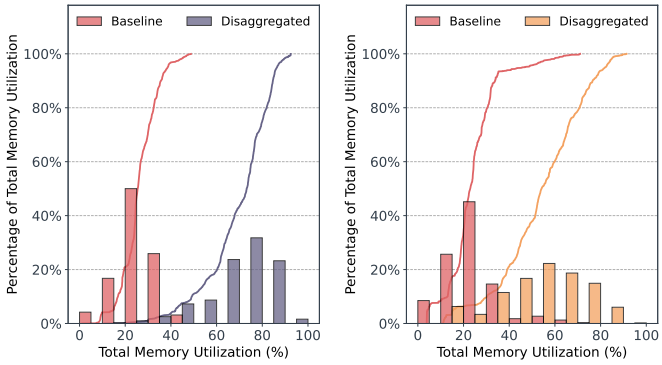
Figure 9 presents the trade-off between system-scale and rack-scale memory disaggregation. From the figure, we can observe that when memory pool capacity is limited, such as 16 TB/rack for Perlmutter and 6 TB/rack for JUWELS, rack-scale memory disaggregation exhibits a shorter tail, indicating that fewer jobs compared to system-scale disaggregation experience a significant bounded slowdown. This outcome is understandable, as rack-scale disaggregation inherently offers lower access latency to the shared memory pool compared to cross-rack access, thus reducing performance penalties. When memory capacity is sufficient, the performance of rack-scale and system-scale disaggregation is similar, as memory resources are likely to be available within the same rack, even when cross-rack memory accessing is an option.

Additionally, it is noteworthy that as available memory is reduced, the Cumulative Distribution Functions (CDFs) of the three illustrated lines of Figure 9 become more different. As previously noted, an insufficient memory pool capacity hinders the allocation of computing nodes to jobs, resulting in longer waiting times (i.e. larger bounded slowdown) in the queue for both rack-scale and system-scale configurations.



(a) Perlmutter trace. (b) JUWELS trace.

Figure 10: Cost-benefit measured as throughput per memory cost.



(a) Perlmutter trace. (b) JUWELS trace.

Figure 11: Comparision of the total memory utilization.

C. Cost Benefits

In this subsection, we analyze the cost versus benefit for the two HPC systems in our evaluation across various memory pool capacity configurations. Based on our earlier results in this paper, we consider rack-scale memory disaggregation for its lower performance penalty and better tail performance. Since the number of computing nodes remains unchanged, the cost-benefit is assessed based on memory expenses. According to the latest DDR5 prices [38], the average cost per gigabyte is $4.9. We quantify our evaluation metric by calculating the throughput (jobs per 100 seconds) per dollar spent on total memory capacity, which includes both the compute node's local memory and the remote memory pool.

Figures 10a and 10b present a detailed analysis of throughput per memory cost for the Perlmutter and JUWELS traces, respectively. The x-axis quantifies the memory pool capacity, while the y-axis measures the throughput per dollar spent on memory. The baseline, representing the throughput per dollar with each node configured with 512 GB of local memory, serves as a reference to gauge the benefits of memory disaggregation.

For the Perlmutter trace, the data clearly shows that as memory pool capacity increases from 4 TB/rack to 20 TB/rack, the throughput per dollar rises significantly. The throughput per dollar remains consistent between 20 TB/rack and 28 TB/rack, peaking at 28 TB/rack. However, a decline in throughput per dollar beyond 28 TB/rack indicates a turning point where additional investment in memory yields diminishing returns. Compared to the baseline, the maximum throughput per dollar at 28 TB/rack represents a 2.1× improvement in cost-benefit.

A similar trend is observed in the JUWELS trace, with a pivotal point at 6 TB/rack. Before this point, increasing investments in memory significantly boosts throughput per dollar. However, beyond this point, the throughput per dollar declines and may even fall below that of the baseline configuration. The highest throughput per dollar, observed at 6 TB/rack, is 2.3× that of the baseline.

When configuring the systems for optimal cost-benefit, we can easily calculate the total memory capacity and the associated costs. For the Perlmutter trace, the baseline total memory capacity is 768 TB, which can be reduced to 264 TB, resulting in a savings of 66 % (about 2.5M in US dollars). Similarly, in the JUWELS trace, the total memory capacity can be reduced from 480 TB to 180 TB, leading to a savings of 63 % (about 1.5M in US dollars).

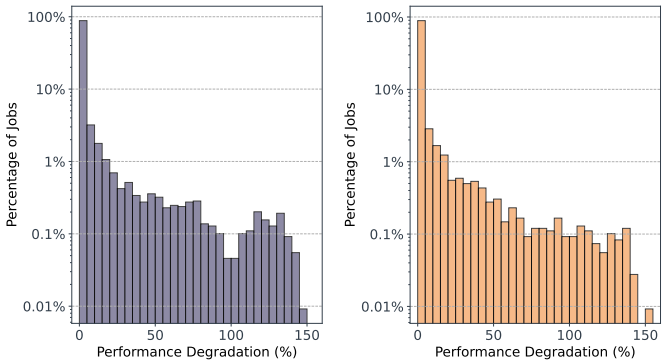
D. Memory Utilization

To further quantify the benefits of disaggregated memory, we now analyze memory utilization. Figure 11 illustrates the distribution of total memory utilization with disaggregated memory compared to the baseline. The lines represent CDFs, while the histograms depict the utilization percentages divided into 10 % bins. We conducted simulations with 64 GB of local node memory and remote memory capacities of 28 TB/rack and 6 TB/rack for the Perlmutter and JUWELS traces, respectively. These configurations were chosen for their optimal cost-benefit found previously. Both simulations employed the *FM* job scheduling algorithm, with memory resources disaggregated at the rack-scale.

The figure shows that in non-disaggregated systems, memory utilization typically ranges from 20–30 % for both traces. In contrast, memory utilization significantly increases in disaggregated systems, indicating more efficient resource utilization: utilization shifts to the range of 70–80 % for the Perlmutter trace and 50–60 % for the JUWELS trace. Consequently, the average total memory utilization increases dramatically—from 26 % in non-disaggregated systems to 69 % for the Perlmutter trace, and from 24 % to 54 % for the JUWELS trace.

E. Performance Degradation

Figure 12 displays the distribution of job performance degradation associated with the use of disaggregated memory in both the Perlmutter and JUWELS traces. The configurations for these simulations are identical to those used in the memory utilization analysis of Section IV-D. The histograms categorize performance degradations into 5 % bins; the y-axis is log-scaled to enhance visibility for the lower percentages of large performance degradations.



(a) Perlmutter trace. (b) JUWELS trace.

Figure 12: Distribution of job performance degradation.

From the figure, it is evident that the majority of jobs experience less than 5 % performance degradation—specifically, 88 % in the Perlmutter trace and 89 % in the JUWELS trace. In both traces, fewer than 0.01 % of jobs experience as much as 150 % performance degradation. Accordingly, the average performance degradation is only 4.5 % for the Perlmutter trace and 4.0 % for the JUWELS trace. This observation means that in order to significantly improve average slowdown such as in Figure 7, we only have to provide software or hardware solutions for a few jobs, such as a set of nodes with high local memory capacity. Also, while those few jobs may be important, the vast majority of jobs will be minimally affected by memory disaggregation.

V. Related Works

Extensive research has been carried out in HPC job scheduling [21], [36], [37], [39]–[45]. These efforts have primarily aimed to devise scheduling policies that range from simple schedulers like FCFS to more intricate and expert-customized approaches such as WFP3 [36]. Additionally, researchers have explored various techniques, including integer linear programming, non-linear algorithms, and neural networks, to derive novel scheduling policies [37], [39]–[43]. More recently, there has been a growing interest in leveraging deep reinforcement learning for job scheduling [21], [44], [45].

In addition to resource scheduling for compute nodes, numerous studies delved into multi-resource scheduling, addressing various aspects such as scheduling burst buffer resources to alleviate I/O contention [46], [47] and implementing power-aware scheduling to optimize node utilization while adhering to power constraints [48]–[51], among others. However, there has been a notable scarcity of research on scheduling and resource allocation within the context of disaggregated memory in HPC environments. An exception to this is the work by Zacarias et al. [52], where they extended an existing Slurm simulator to accommodate disaggregated memory. Their approach involved the use of a multi-node slowdown model to predict job performance degradation in scenarios where memory resources are shared among jobs, particularly in heterogeneous setups where compute nodes with a large memory capacity provide shared remote memory. Notably, their research primarily focused on extending the resource allocation plugin within Slurm to support remote memory allocation, with limited exploration into job scheduling algorithms and their potential impact on overall system performance.

Unlike previous research that primarily addresses traditional HPC architectures and relies on job attributes from the Standard Workload Format (SWF) to formulate priority functions, our study focuses on scheduling in HPC systems featuring memory disaggregation. We introduce a novel attribute called “memory overload” to quantify a job’s remote memory requirements. As demonstrated by our experimental results, our proposed scheduler (*FM*) that uses a heuristic priority function and incorporates the memory overload attribute, can outperform the state-of-the-art scheduler *F1* in terms of average bounded slowdown and fairness in a memory disaggregated system.

VI. Conclusion

In this study, we conducted a comprehensive investigation of job scheduling in HPC systems that feature memory disaggregation. We developed a performance degradation model and used real-world job traces from two production systems to estimate job runtimes when accessing remote memory resources. Our findings highlighted the superior performance of our novel *FM* scheduling algorithm, particularly in terms of bounded slowdown and fairness. Additionally, we explored the performance differences between rack-scale and system-scale memory disaggregation, revealing that rack-scale disaggregation reduces maximum job slowdown performance when memory pool capacity is limited. We also performed a cost-benefit analysis to determine the most efficient memory pool capacity configurations. Additional evaluations of memory utilization and performance degradation highlighted the benefits and trade-offs of a disaggregated HPC system compared to today’s non-disaggregated configurations. Our results indicate substantial savings in memory costs—over 60 %—with minimal impact on job performance of approximately 4 % on average.

Acknowledgment

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02- 05CH11231, and partially by the National Science Foundation under grants OAC-1835892, CNS-1817094, and CNS-1939140. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility. This work has received funding from the European Commission’s H2020, and EuroHPC Programmes, under Grant Agreement number 955606 (DEEP-SEA). The EuroHPC Joint Undertaking (JU) receives support from the European Union’s Horizon 2020 research and innovation programme and Germany, France, Spain, Greece, Belgium, Sweden, and Switzerland.

References

[1] G. Panwar, D. Zhang, Y. Pang, M. Dahshan, N. DeBardeleben, B. Ravindran, and X. Jian, "Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 821–835.

[2] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 183–190.

[3] G. Michelogiannakis, B. Klenk, B. Cook, M. Y. Teh, M. Glick, L. Dennison, K. Bergman, and J. Shalf, "A case for intra-rack resource disaggregation in hpc," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 2, pp. 1–26, 2022.

[4] J. Li, G. Michelogiannakis, B. Cook, D. Cooray, and Y. Chen, "Analyzing resource utilization in an hpc system: A case study of nersc's perlmutter," in *International Conference on High Performance Computing*. Springer, 2023, pp. 297–316.

[5] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH computer architecture news*, vol. 37, no. 3, pp. 267–278, 2009.

[6] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out numa," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 3–18, 2014.

[7] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap." in *NSDI*, 2017, pp. 649–667.

[8] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, and H. P. Hofstee, "Thymesisflow: A software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 868–880.

[9] 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 *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[10] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[11] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "{LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 69–87.

[12] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "{AIFM}:{High-Performance},{Application-Integrated} far memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 315–332.

[13] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 79–92.

[14] K. Keeton, S. Singhal, and M. Raymond, "The openfam api: a programming model for disaggregated persistent memory," in *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity: 5th Workshop, OpenSHMEM 2018, Baltimore, MD, USA, August 21–23, 2018, Revised Selected Papers 5*. Springer, 2019, pp. 70–89.

[15] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[16] B. Nitzberg, J. M. Schopf, and J. P. Jones, "Pbs pro: Grid computing and scheduling attributes," in *Grid resource management: state of the art and future trends*. Springer, 2004, pp. 183–190.

[17] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *IEEE transactions on parallel and distributed systems*, vol. 12, no. 6, pp. 529–543, 2001.

[18] D. A. Lifka, "The anl/ibm sp scheduling system," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1995, pp. 295–303.

[19] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'98 Workshop Orlando, Florida, USA, March 30, 1998 Proceedings 4*. Springer, 1998, pp. 1–24.

[20] E. Gaussier, D. Glesser, V. Reis, and D. Trystram, "Improving backfilling by using machine learning to predict running times," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–10.

[21] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "Rlscheduler: an automated hpc batch job scheduler using reinforcement learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[22] J. Ngubiri and M. van Vliet, "A metric of fairness for parallel job schedulers," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 12, pp. 1525–1546, 2009.

[23] G. Michelogiannakis, Y. Arafa, B. Cook, L. Y. Dai, A. H. Badawy, M. Glick, Y. Wang, K. Bergman, and J. Shalf, "Efficient intra-rack resource disaggregation for hpc using co-packaged DWDM photonics," *arXiv preprint arXiv:2301.03592*, 2023.

[24] D. S. Berger, D. Ernst, H. Li, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, L. Hsu, I. Agarwal, M. D. Hill *et al.*, "Design tradeoffs in cxl-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, 2023.

[25] Intel, "Intel Xeon CPU Max Series configuration and tuning guide," Sep. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/769060/intel-xeon-cpu-max-series-configuration-and-tuning-guide.html

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[27] D. D. Sharma, "Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing," in *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2022, pp. 5–12.

[28] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 331–345. [Online]. Available: https://doi.org/10.1145/3297858.3304024

[29] T. Wang, H. Liu, and H. Jin, "Efficient remote memory paging for disaggregated memory systems," in *Algorithms and Architectures for Parallel Processing: 22nd International Conference, ICA3PP 2022, Copenhagen, Denmark, October 10–12, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 1–20. [Online]. Available: https://doi.org/10.1007/978-3-031-22677-9_1

[30] NERSC's Perlmutter configuration. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/

[31] Jülich Supercomputing Centre, "JUWELS Cluster and Booster: Exascale pathfinder with modular supercomputing architecture at Juelich Supercomputing Centre," *Journal of large-scale research facilities*, vol. 7, p. A183, Oct. 2021. [Online]. Available: https://doi.org/10.17815/jlsrf-7-183

[32] Jülich Supercomputing Centre. (2024) Hardware Configuration of the JUWELS Booster Module. [Online]. Available: https://apps.fz-juelich.de/jsc/hps/juwels/configuration.html#hardware-configuration-of-the-system-name-booster-module

[33] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden *et al.*, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 154–165.

[34] Jülich Supercomputing Centre. (2023, Oct.) LLview. [Online]. Available: https://llview.fz-juelich.de

[35] Y. Müller, F. Souza Mendes Guimarães, C. Karbach, and W. Frings, "LLview v2.2.3-base," Zenodo, Feb. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10221407

[36] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-aware, utility-based job scheduling on blue, gene/p systems," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[37] D. Carastan-Santos and R. Y. De Camargo, "Obtaining dynamic scheduling policies with simulation and machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–13.

[38] Dramexchange. [Online]. Available: https://www.dramexchange.com/#memory

[39] E. S. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed systems*, vol. 5, no. 2, pp. 113–120, 1994.

[40] C. A. Floudas and X. Lin, "Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications," *Annals of Operations Research*, vol. 139, pp. 131–162, 2005.

[41] A. Agarwal, S. Colak, V. S. Jacob, and H. Pirkul, "Heuristics and augmented neural networks for task scheduling with non-identical machines," *European Journal of Operational Research*, vol. 175, no. 1, pp. 296–317, 2006.

[42] D. E. Akyol and G. M. Bayhan, "A review on evolution of production scheduling with neural networks," *Computers & Industrial Engineering*, vol. 53, no. 1, pp. 95–122, 2007.

[43] H. Al-Daoud, I. Al-Azzoni, and D. G. Down, "Power-aware linear programming based scheduling for heterogeneous computer clusters," *Future Generation Computer Systems*, vol. 28, no. 5, pp. 745–754, 2012.

[44] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, and M. E. Papka, "Deep reinforcement agent for scheduling in hpc," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 807–816.

[45] Q. Wang, H. Zhang, C. Qu, Y. Shen, X. Liu, and J. Li, "Rlschert: an hpc job scheduler using deep reinforcement learning and remaining time prediction," *Applied Sciences*, vol. 11, no. 20, p. 9448, 2021.

[46] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 69–80.

[47] Y. Fan, Z. Lan, P. Rich, W. E. Allcock, M. E. Papka, B. Austin, and D. Paul, "Scheduling beyond cpus for hpc," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 97–108.

[48] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[49] F. Kaplan, J. Meng, and A. K. Coskun, "Optimizing communication and cooling costs in hpc data centers via intelligent job allocation," in *2013 International Green Computing Conference Proceedings*. IEEE, 2013, pp. 1–10.

[50] S. Wallace, X. Yang, V. Vishwanath, W. E. Allcock, S. Coghlan, M. E. Papka, and Z. Lan, "A data driven scheduling approach for power management on hpc systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 656–666.

[51] T. Cao, W. Huang, Y. He, and M. Kondo, "Cooling-aware job scheduling and node allocation for overprovisioned hpc systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 728–737.

[52] F. V. Zacarias, P. Carpenter, and V. Petrucci, "Improving hpc system throughput and response time using memory disaggregation," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2021, pp. 283–290.