Dynamic Memory Provisioning on Disaggregated HPC Systems

Felippe Vieira Zacarias
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Barcelona, Spain
fvieira@bsc.es

Paul Carpenter
Barcelona Supercomputing Center
Barcelona, Spain
paul.carpenter@bsc.es

Vinicius Petrucci
Micron Technology
Austin, United States
vtavarespetr@micron.com

ABSTRACT

Disaggregated memory is under investigation as a way to break the rigid boundaries between node memory hierarchies in order to provide memory as a system-wide pooled resource. The resource manager allocates the system's disaggregated memory to jobs, based on the memory requirements defined by the user at job submission time. It is hard for the user to know the job's precise peak memory footprint, and prior work has shown that users have an incentive to overestimate their needs. This overestimation leads to a significant overallocation of memory, and the majority of the physical memory in the system is wasted. This paper presents a way to reclaim much of this overallocated memory. We extend the Slurm job scheduler to dynamically reallocate memory, according to the job's current memory footprint. We enhance an existing Slurm simulator to model this situation and combine publicly available traces to model an HPC system on up to 1490 nodes. Our results show that the dynamic memory provisioning approach increases the throughput per dollar by up to 38%, compared to a system with static allocation of disaggregated memory.

KEYWORDS

Disaggregation, Throughput, Response time, Resource scheduling, Resource provisioning, Slurm

ACM Reference Format:
Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. 2023. Dynamic Memory Provisioning on Disaggregated HPC Systems. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624174

1 INTRODUCTION

High-Performance Computing (HPC) applications have widely varying per-node memory footprints due to diverse application characteristics, differing problem sizes, and strong scaling [21, 26, 49]. In a typical HPC cluster architecture, memory is tightly coupled to the CPUs running the jobs, leading to stranded memory capacity and an inefficient use of the memory resources. In fact, 25% to 76% of the total memory capacity typically remains idle [18, 23, 28, 30]. Disaggregated memory offers a way to improve memory utilization, as memory becomes a pool that can be dynamically composed to match the needs of the workloads [22]. It enables fine-grained allocation of memory capacity to jobs [16, 21, 24, 30, 37, 48], while maintaining the cost-effectiveness and scalability of a cluster architecture [18].

HPC job schedulers generally allocate resources statically [29], so disaggregated memory resource management systems require the user to specify the peak memory demands at submission time [29, 30, 45]. It is difficult for users to know the precise maximum memory footprint, and they have an incentive to overestimate this figure to avoid an out-of-memory error, which would terminate the job [21, 34, 38, 39]. A recent paper investigated the incentive for users of a disaggregated memory system to provide accurate memory estimates. They showed a tragedy of the commons effect: for example, a single user overestimating their memory demands by 60% increases their response time (from submission to completion) by just 8%, but the combined result of everybody doing the same would be a 5 times increase in response time and 25% reduction in throughput [46].

This paper makes a case for dynamic reallocation of disaggregated memory. While we see a small benefit from the difference between the job's peak and average memory consumption, there is a large benefit from the difference between the job's peak memory consumption and the memory demand that the user would specify in the job submission. We propose a strategy for dynamic memory allocation that reclaims overallocated memory and we evaluate the policy using the Slurm simulator [1]. The simulation approach allows work to proceed before the availability of a large-scale HPC system with disaggregated memory and complete software stack, as well as enabling rapid at-scale evaluation of multiple scenarios without occupying a real system. We find that even assuming a conservative approach where the users correctly estimate their maximum memory usage, system performance increases by up to 8%. When memory demands are overestimated by 60%, the improvement in performance for an underprovisioned system is up to 13%. Moreover, employing the dynamic approach results in equivalent performance to the baseline while using fewer resources, specifically 40% less memory provisioning for comparable throughput, within 5%.

Dynamic resource assignment has been explored in the context of malleability [12, 17]. Unlike approaches for malleability, our approach does not need any modifications to the application. Other studies investigate disaggregated memory but are done at a relatively small scale [22, 31]. In contrast, our simulation approach has allowed the evaluation to be performed on up to 1490 nodes. To the best of our knowledge, no prior studies have tackled the specific

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624174

problem of cluster-level dynamic job scheduling on large-scale disaggregated memory systems. A more extensive comparison with related work is in Section 5.

In summary, we make the following contributions:

(1) We extend Slurm's memory allocation policy and the Slurm simulator to support disaggregated memory with dynamic memory reallocation.
(2) We present a methodology to create complete HPC job traces with dynamic memory footprints from publicly available datasets.
(3) We evaluate a set of simulated scenarios using synthetic and real-world traces and investigate how the job memory allocation affects overall system throughput, response time, utilization, and cost–benefit in HPC systems.
(4) We demonstrate that dynamic memory assignment delivers improvements up to 13% in throughput, 38% in throughput-per-dollar, and up to 69% reduction in median job response time, compared to a static policy, when there are imbalanced memory usage and overestimated demands on underprovisioned systems.

Please note that our dynamic memory allocation scheme and job trace generation methodology are accessible as open source [6, 44]. We encourage others to not only replicate our work but also to contribute to its further enhancement.

2 DISAGGREGATED MEMORY MANAGEMENT

2.1 Simulation infrastructure for disaggregated memory

Slurm [43] is an open-source, scalable resource management system widely used in HPC. Jobs are scheduled by Slurm to nodes that can satisfy the requested core and memory resources [18]. Allocation of nodes is by default exclusive, meaning that unused resources cannot be assigned to another job, given the possible negative impact of inter-workload interference [28, 30].

We leverage the Slurm simulator [1, 19], which provides a scalable environment to develop and evaluate HPC job scheduling policies. It takes as input a job trace based on the Standard Workload Format (SWF) [2, 10]. Slurm simulator employs most of Slurm's original source code, enabling an accurate evaluation that captures all parameters and behavior that occur in a real environment.

Zacarias *et al.* [45] extend Slurm to allocate disaggregated memory using a static policy. Their policy tries to run the job on nodes with enough free memory. If this is not possible, then it will choose nodes with the most free memory and borrow the remaining memory from other nodes. A node that has already lent memory can still run new jobs, as long as it has lent in total no more than half of its total memory capacity. At that point, the node temporarily becomes a memory node that can lend memory but not run new jobs. All memory allocation is done based on the memory request in the job submission. Any job that exceeds its memory request is killed.

The policy of Zacarias *et al.* quantifies the performance slowdown due to remote memory access latency and bandwidth contention using a contention model [45, 47]. Each application is characterized by a sensitivity curve, relating memory bandwidth contention to performance as well as a contentiousness figure, which measures the memory bandwidth at full performance. The model uses remote memory bandwidth since remote accesses do not create cache contention in their disaggregated memory system. Application profiling is only needed for the simulation-based evaluation methodology, and it is not an input to the resource management policy that would be used in production.

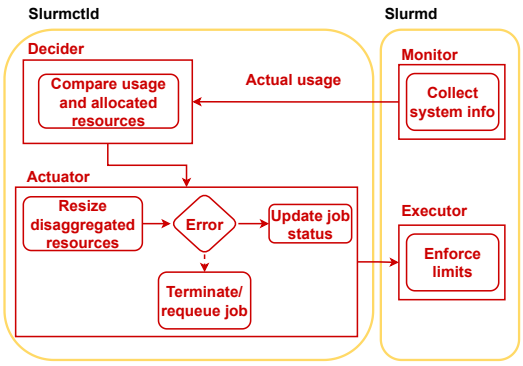
2.2 Dynamic memory allocation policy

We enhanced Slurm simulator, extended for disaggregated memory by Zacarias *et al.* [45], to support dynamic memory allocation. This allowed us to develop and evaluate the dynamic memory allocation policy without requiring a large-scale dedicated HPC system with disaggregated memory (which is impractical as the technology is still in its infancy). Slurm uses a configurable architecture with an extensive set of plugins and a centralized manager (or controller), *Slurmctld*, which allocates resources to jobs, monitors job execution, and mediates contention to resources through a queue of pending jobs. Each compute node executes an instance of the *Slurmd* daemon, which communicates with the controller to receive work and manage job execution on the node.

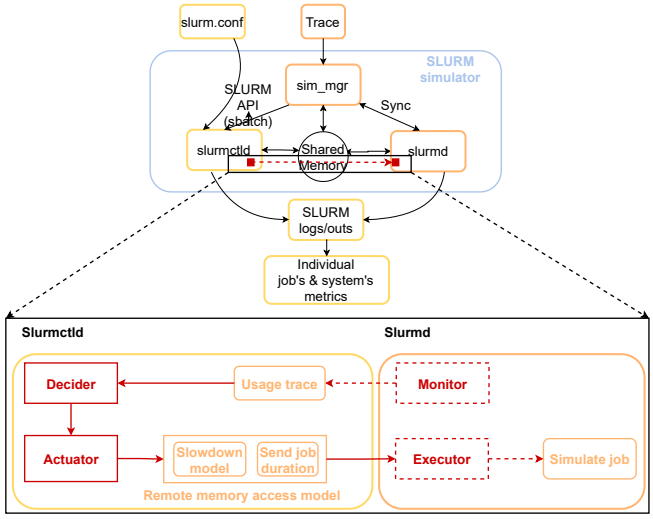
Figure 1a depicts our memory allocation scheme in the context of the Slurm resource manager. The scheme is divided into the *Monitor*, *Decider*, *Actuator*, and *Executor* modules, and it works as follows. The initial allocation of a job is done in the same way as Zacarias *et al.*, based on the memory request in the job's submission script. But once the job begins, its actual memory consumption is monitored over time, by the *Monitor* module in Slurmd, which runs on every node. The memory usage information is collected by the system for all running jobs. Prior work [7, 42] has already demonstrated low overhead for data collection to track job executions, with sampling intervals on the order of seconds. The updating interval is a critical parameter, as overly frequent adjustments incur additional memory management overheads whereas infrequent adjustments fail to accurately capture the actual memory usage. In our work, we update the memory usage on average every 5 minutes, which is the same as that used in [42]. The usage information is passed to the Slurm controller, which updates the job memory allocations.

When Slurm receives the updated current memory consumption for a particular node in a job, it will make a decision based on the current allocation (*Decider* module). Next, the memory will be updated by the *Actuator* module. If the current memory usage on the node is lower than the current allocation, the resource manager will deallocate memory. It will deallocate remote memory before deallocating local memory. On the other hand, if the new usage is higher than the current allocation, the resource manager will allocate memory locally, if possible, and then remotely if necessary. The idea is to maximize the local-to-remote ratio, thus decreasing the impact of remote memory accesses. Finally, the controller will update the job's access to physical memory on the node using the *Executor* module, which runs on each node. This module will reset memory capacity constraints available to the job locally and remotely.

We assume the system has a proper allocation management that will prioritize local memory rather than remote memory. Therefore, it should have an efficient mechanism for moving the accessed data to the local memory, while unused data will be in the remote region. An important management question is what to do when the system runs out of memory. Dynamic memory allocation intentionally



(a) Dynamic disaggregated memory allocation in Slurm.



(b) Evaluation of dynamic disaggregated memory using Slurm simulator.

Figure 1: Proposed dynamic allocation of disaggregated memory and its integration into Slurm and Slurm simulator.

allows the peak memory demands of the running jobs to exceed the system's total physical memory. When a job increases its memory usage, the system may not have enough free memory to satisfy its needs. An invalid approach would be to block the job until memory becomes available, but this would clearly risk deadlock.

Two valid approaches for dealing with jobs running out of memory are Fail/Restart (F/R) and Checkpoint/Restart (C/R). In both approaches, the *Actuator* terminates the job, releases its resources, and resubmits the job to execute later. F/R restarts the job from the beginning whereas C/R restarts from a recent checkpoint. We may expect C/R to perform better, but it is more complex. Most C/R libraries require the application to control checkpointing and restart. Nevertheless, we found that out-of-memory errors at the system level are rare. In fact, in the most extreme scenario,[1] less than 1% of jobs fail due to insufficient memory. We conclude that F/R is sufficient, and present all results using the F/R approach. Nevertheless, in production, the proposed dynamic memory allocation may sometimes cause jobs to fail repeatedly, due to abnormal system load or an application with outlier phases allocating memory. In order to mitigate this problem, the resource manager can take several actions to ensure fairness. One approach would be to increase the job's priority and allocate additional resources after a specified number of failures. Alternatively, the resource manager could opt to initiate the job without dynamic resource allocation, instead assigning resources to the job in a static and guaranteed manner.

[1] 100% large jobs, 50% system, +100% overestimation (see Section 3).

2.3 Dynamic memory allocation in the Slurm simulator

Figure 1b shows our approach to evaluate the dynamic allocation of disaggregated memory using the Slurm simulator. It is based on the work of Zacarias *et al.* [45], and our modifications are depicted as orange boxes. The functions that are partially implemented or adapted to run in a simulation environment rather than a real system are represented with dotted boxes. The *Decider* module receives the memory usage from the offline memory usage trace (details in Section 3.2), rather than receiving the memory status from the nodes in the cluster. This step mimics the *Monitor* module feeding the current memory usage to the dynamic memory allocation policy to enforce the memory usage in case a job exceeds its memory allocation. Our extension works by executing the following steps: once the system has jobs running, the simulator will calculate at which simulated time it must issue commands to update the jobs. To calculate the expected simulation time it uses the job's progress, which is its elapsed running time. Since multiple jobs run concurrently, the simulator will use the job's earliest progress to update the timer to enforce the new usage in the system.

Once the simulation reaches a particular time, it issues commands to update the jobs whose progress is within the period. A command specifies the node identification and its new memory usage. The resource manager then receives a list with the job and usage of each node to apply the new allocation. The *Actuator module* then allocates or deallocates memory to match the node's current memory usage. We consider the memory demand to be the maximum memory usage in the time period between the current progress and the next update, as represented in the original trace. Next, the *Actuator* module applies the contention model to update the simulation and job duration, and the calculated job progress is sent to the *Executor* module. Since the simulated Slurmd daemon is a simplified version that emulates job execution for all nodes in the system, it only updates the job duration and the queue of jobs being simulated instead of actually reconfiguring memory capacity locally and remotely.

3 METHODOLOGY

3.1 Job traces

We used three sources of job traces, which are summarized in Table 1 and described in more detail below:

3.1.1 Grizzly trace. In 2019, Los Alamos National Lab (LANL) released an HPC memory usage trace, which details the memory usage of three HPC clusters in the period from late 2018 through early 2019 [5, 28]. We chose the dataset for the largest system, Grizzly [4], a mid-range TOP 500 supercomputer with 1490 nodes, each with 128 GB DRAM. The complete Grizzly dataset consists

Table 1: Summary of data provided by the job traces.

Trace	Domain	Submission times	Memory request	Num. nodes	Job duration	Memory trace
Grizzly [5, 28]	HPC	×	×	✓	✓	✓
CIRNE [11, 45]	HPC	✓	✓	✓	✓	×
Google [40]	Cloud	×	×[1]	✓	✓	✓[2]

1. Some records in the Google trace have the memory request, but most do not.
2. The Google memory trace is normalized to the largest machine (we assumed 12 TB).

of 53.4 GB of (uncompressed) data, which comprises over 70,000 jobs and 560 million records. The memory usage over time is collected using the Lightweight Distributed Metric Service (LDMS) [7], which gives a snapshot, collected every ten seconds on each node, of the current job and the free and active memory. There is no information from the job scheduler, such as submission time, requested memory, or the type of job. Since each job is identified using a unique ID, it is possible to identify parallel jobs running on multiple nodes and therefore deduce the job's number of nodes and duration [28]. As shown in Table 1, the Grizzly trace provides all necessary information except job submission times, memory requests, and the slowdown model.

3.1.2 CIRNE model. The CIRNE Comprehensive Model [11] generates a synthetic HPC job trace based on the job arrival pattern, time limits, numbers of nodes, system loads, and time durations observed in real environments. We use the extended model from Zacarias *et al.* [45], which also characterizes the slowdown of each job as a function of disaggregated memory accesses and contention. As shown in Table 1, it provides all parameters required by our methodology except for the memory trace of dynamic memory consumption over time.

3.1.3 Google trace. In 2020, Google released detailed job traces from eight of its Borg cells for the entire month of May 2019 [40]. Each Borg cell is a collection of machines that together operate as a single management unit. We use only the data from cell *b*, which, according to [40], has the largest proportion of batch jobs. Each entry in the trace is either an *alloc set*, which describes a resource reservation, or a single *job* submission, which describes the computation to run and the resources it needs. Each job may run several *tasks*, which inherit properties such as the priority and resource request [40]. Jobs are classified according to their priority and scheduling class, i.e. latency sensitivity, distinguishing user-facing service jobs vs. non-production or batch jobs [42]. The memory usage on each node is sampled once per second and recorded as average and maximum values over 5-minute windows [42].

Some hardware characteristics in the Google trace have been obfuscated for confidentiality reasons. For instance, memory sizes are normalized relative to the largest machine memory capacity [42]. It was reported that the maximum capacity of a system in operation at the time was 12 TB [3], so we used this figure to denormalize the data. As shown in Table 1, after denormalizing the memory usage, this trace provides all necessary information except job submission times and memory requests.

3.2 Generating the job traces

None of the traces described in Section 3.1 provide all of the information that is required for the analysis. Our approach is to generate two sets of traces. The first is based on LANL's Grizzly trace, augmented by the job submission times and model from the CIRNE model. The second uses the Google trace shaped by HPC job statistics from CIRNE and Archer [41]. Both traces use the slowdown model from Zacarias *et al* [45].

3.2.1 Adapting the Grizzly trace. We sampled the Grizzly dataset (see Section 3.1.1), to obtain a smaller trace that was feasible to simulate. Figure 2 shows all the one-week periods in the Grizzly trace, in terms of CPU utilization (on the x-axis), maximum job node–hours (on the y-axis of the left-hand plot) and maximum job memory usage (on the y-axis of the right-hand plot). The CPU utilization was calculated as the total node–hours of the jobs divided by the total node–hours over the period. The simulated periods are shown as blue triangles and the remaining periods are shown as grey dots. We took a random sampling of the weeks with the utilization of 70% or more, which is representative of HPC [28]. We then randomly chose seven periods to simulate. Figure 2 shows that the chosen periods are representative of the important periods during which utilization is relatively high.

The trace of memory consumption over time is reduced in size using the Ramer–Douglas–Peucker (RDP) algorithm [13, 32]. We generated the submission times using the CIRNE Comprehensive Model [11]. Although the job's actual peak memory consumption is known, the memory demand that the user would specify in the job submission script is unknown. We therefore provide a sweep on the overestimation factor, from +0% (demand equals peak memory use) to +100% (demand is double the peak memory use). The job type is needed by the multi-node slowdown model [47] described above. We match the job to a profiled application by minimizing the Euclidean distance of the size and runtime.

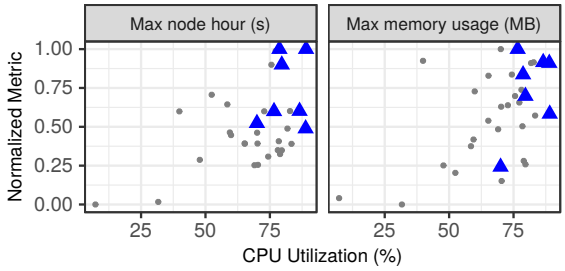


Figure 2: Sampling the Grizzly trace. Each point represents a 1-week period. Simulated periods, depicted as blue triangles, are representative of weeks with CPU utilization ≥ 70%.

3.2.2 Synthetic model plus Google trace. We generate a set of synthetic input files using the CIRNE Comprehensive Model [11] (Section 3.1.2) jointly with the per-job memory traces from the Google trace (Section 3.1.3). The overall process is shown in Figure 3. The first four steps follow the methodology of Jokanovic *et al.* [19] as extended by Zacarias *et al.* [45]. We first generate a synthetic trace using the CIRNE Model (Step 1). We select the job from a pool of applications that have been previously profiled regarding size, runtime, memory bandwidth, read/write ratio, and local/remote access memory ratio (Step 2). We map each real application to the

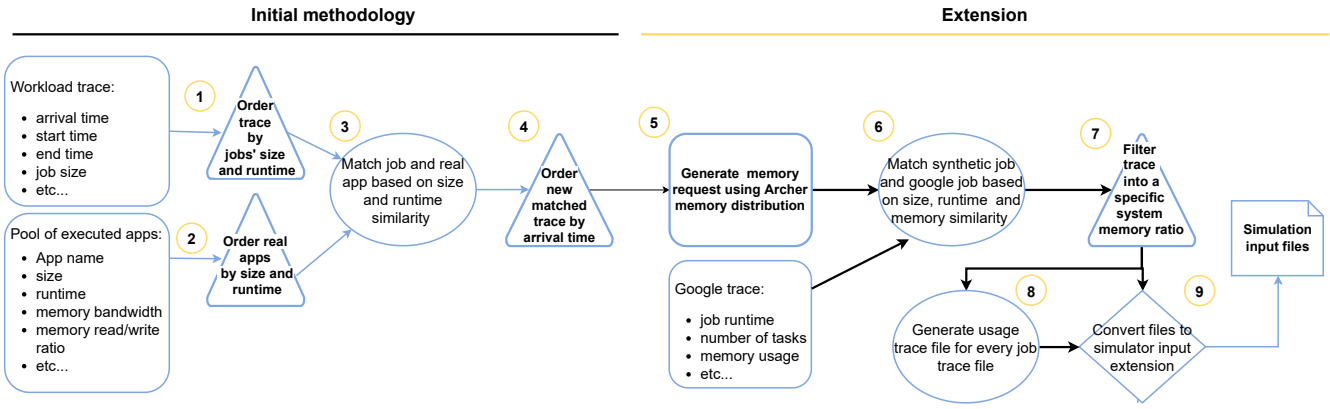


Figure 3: Augmenting the workload trace with real application data and per-job memory usage from the Google trace. Methodology adapted and extended from [19, 45].

nearest profiled job based on the Euclidean distance between their sizes and execution times (Step 3) and finally sort the trace by job arrival time (Step 4).

The remaining steps in Figure 3 are new. Since the synthetic trace does not specify the memory request, we follow the memory request distribution of the Archer supercomputer [41] (Step 5). The distribution is displayed in Table 2. We then map the job to a Google job, by minimizing the the Euclidean distance of size, runtime, and memory demand to obtain the trace of memory consumption over time (Step 6). This trace can be long, so we use the RDP [13, 32] algorithm to reduce the number of data points. We then filter the trace to obtain the target proportion of large-memory jobs (Step 7) and generate the memory usage traces and job trace binaries needed by the simulator (Steps 8 and 9).

The Google trace contains a large range of jobs in a cloud environment, and it required some adaptation before it could be used. Since HPC jobs are typically batch jobs, we selected only best–effort batch jobs. We also filtered on the job's priority and scheduling class to extract latency-insensitive batch jobs. Since these jobs were sometimes killed to make room for high-priority jobs, we kept only the jobs that finished normally at least once.

The Google trace reports the average and maximum memory usage for each 5 min interval. We use the maximum used memory to define the usage for the period between two measurements. To capture the behavior of the whole job, we scaled the runtime of the memory trace to match the wallclock duration of the job.

Table 2: Maximum memory usage per node. Each figure is the percentage of jobs. Small jobs are ≤32 nodes and large jobs are >32 nodes. Synthetic figures are adapted from [41].

Max memory	Synthetic			Grizzly		
(GB/node)	All	Normal	Large	All	Normal	Large
(0,12)	61.0%	69.5%	53.0%	73.3%	63.5%	77.8%
[12,24)	18.6%	19.4%	16.9%	12.4%	20.2%	8.9%
[24,48)	11.5%	7.7%	14.8%	8.2%	8.5%	8.0%
[48,96)	6.9%	3.0%	11.2%	5.7%	7.0%	5.0%
[96,128)	2.0%	0.4%	4.2%	0.5%	0.8%	0.3%

3.3 Trace characterization

3.3.1 Synthetic traces. Table 3 presents the characteristics of the normal memory and large memory jobs. The memory demand of normal jobs is less than the capacity of a normal node (see Section 3.4 for node definition), whereas all large jobs demand more memory than a normal node capacity. The generated input job traces for the simulator are sampled without replacement, in the appropriate proportions, from these two distributions. The distribution for maximum, average usage, and requested memory broken down by job size is presented in Figure 4. In our traces, the average usage is much lower than the maximum usage, which opens up room for improvements during resource allocation. On the other hand, the maximum usage and requested memory have similar distributions. This shows that we take a conservative approach in our study.

(a) Average memory usage from usage trace profile.

(b) Maximum memory usage from usage trace profile (equal to requested memory with 0% overestimation).

Figure 4: Trace memory heatmap distribution versus job size for the synthetic trace.

Table 3: Normal and large memory job characteristics.

	Normal memory jobs		Large memory jobs	
Metric	**Memory (MB)**	**Node–hours**	**Memory (MB)**	**Node–hours**
Min	0	0	65538	0
1st Qu.	4037	132	76176	256
Median	8089	2717	86961	6720
3rd Qu.	15341	29264	99956	77028
Max	65532	23082880	130046	23329920

3.3.2 Grizzly trace. Table 2 presents the distribution of the peak memory demand for the Grizzly dataset. Although the system CPU utilization is reported to be *78%* [28], it is clear from its distribution that memory-wise the system is underutilized and provisioned to run the worst cases. The majority of jobs use less than 24 GB per node. According to Panwar *et al.* [28], average node level memory utilization is 18% of its capacity and there is a large gap between the node's worst–case memory usage and its common case utilization.

3.4 Simulated system configuration

The simulated systems are given in Table 4. For both datasets, we separated the systems into *normal* nodes, which have the typical memory capacity, and *large* nodes, which have double the memory capacity of the *normal* nodes. We similarly define a job to be *large* if it requires a large capacity node to run with the baseline policy. A job is *normal* if it can execute on a normal capacity node. Individually the systems have either 128 GB (as in Grizzly [4] and Archer [41] supercomputers) or 64 GB large nodes. We further divided each simulated scenario to correspond to a system with different ratios between large and normal nodes, varying from all normal nodes (0%) to all large nodes (100%).

Table 4: Simulated system configurations.

Parameter	**Synthetic trace**	**Grizzly trace**
System size	1024 nodes	1490 nodes
Number of cores per node	32 cores	
Memory per node (GB)	32, 64, 128	
Allocation policy	Baseline, Disaggregated	
Scheduling policy	Backfill	
Queue and Backfill size	100	
Backfill and Scheduling interval	30 s	
% Large nodes	0, 15, 25, 50, 75, 100	
Cost per node (excl. memory)	\$10,154† [27]	
Cost per 128 GB	\$1280 [27]	

† Cost per node includes node, network, switches, and small storage.

All allocation policies have exclusive access to all CPUs of a node, which implies that the Baseline allocation also considers exclusive access to the memory as well. In our experiments we do not consider a swap system as in our experience, HPC systems typically do not have swap enabled.

The cost–benefit analysis in Section 4.3 uses the estimated component costs given in Table 4, which were taken from a recent analysis of a small-scale HPC cloud platform [27]. The interconnect is a torus, sized as recommended by prior work [35, 36].

3.5 Allocation Policies

We will present the results for the following memory allocation policies:

- **Baseline**: no disaggregated memory (each job has exclusive access to all resources on the node).
- **Static**: disaggregated memory with fixed memory allocation specified in the job submission (Zacarias *et al.* [45]).
- **Dynamic**: disaggregated memory with dynamic memory allocation policy (Section 2).

4 RESULTS

4.1 System throughput (jobs per second)

Each plot in Figure 5 shows the normalized throughput, in jobs per second, on the y-axis, as a function of the system's total amount of provisioned memory, on the x-axis. The throughput is normalized by dividing the throughput by that of the baseline approach (no disaggregation) on a system with 100% memory (rightmost point on the x-axis). The total system memory capacity is normalized by dividing it by the total memory capacity of a 100% large node system. The panels in the top row correspond to +0% overestimation, i.e., the users specify the exact peak memory footprint, for every job, at job submission time. The panels in the bottom row correspond to a more realistic 60% overestimation. The columns show different proportions of large jobs for the synthetic trace, together with the Grizzly trace at the right-hand side.

If the demand for memory consumption is low, e.g. in the top-left panel corresponding to +0% overestimation (top) and 0% large jobs (left), there is little difference between the performance of the three policies. Since normal jobs require up to 64 GB per node, the baseline policy achieves full performance with 50% of the full system memory capacity, i.e., all normal capacity nodes. It is not possible to reduce the memory provisioning further for the baseline policy (hence the missing bars below 50%). By sharing memory capacity, the static and dynamic disaggregated memory policies are both able to share and maintain full performance with a lower memory provisioning of 37%.

As the proportion of large jobs increases, along the top row, memory has an increasing effect on system throughput, and the difference between the three policies increases. We see a large difference between the baseline and static approaches, and up to 8% difference between the static and dynamic approaches. By reclaiming most of the unused memory from the jobs, so that each job's average memory provisioning matches its average (not peak) memory demands, more jobs are able to run concurrently.

In the bottom row of Figure 5, the peak memory footprint is overestimated by a more realistic +60%. In this case, some of the jobs cannot be executed by the baseline policy, so results are only shown for the two disaggregated memory policies. We also see a significant difference between the static and dynamic approaches. For example, with 75% large jobs and a system with 50% total memory, the dynamic approach achieves throughput over 95%, which is 13% above that of the static approach. In summary, the largest benefit from the dynamic approach is seen for underprovisioned systems with a high number of large jobs and also for scenarios in which the users overestimate their memory demands.

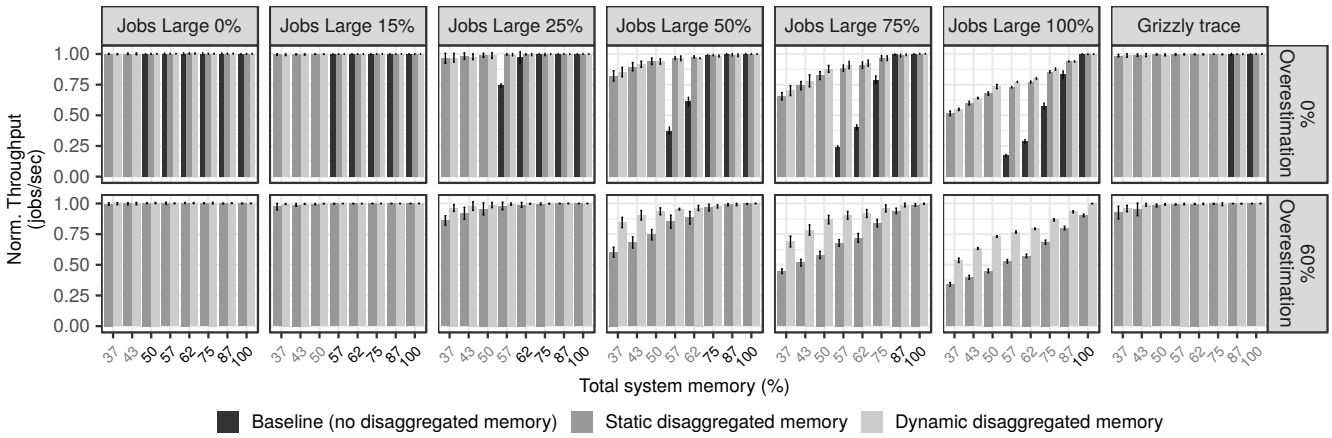


Figure 5: Normalized throughput (y-axis) for each memory configuration (x-axis) for various job mixes. Bold x-axis labels identify overprovisioned memory systems. Missing bars in the plots indicate there are not enough large memory nodes to run all jobs. The largest benefit from the dynamic approach is seen for underprovisioned systems with high numbers of large jobs.

4.2 Job response time

Figure 6 shows the empirical cumulative distribution function (ECDF) of the job response times (waiting time plus runtime). The x-axis is the response time on a logarithmic scale and the y-axis is the cumulative empirical probability, from 0 to 1. We divide the results into three scenarios: overprovisioned (when the job mix demands fewer large nodes than is available), matching (job mix demands an equal number of large nodes), and underprovisioned (job mix demands more large nodes than is available). For +0% overestimation (top row), all three scenarios show little difference in performance between the static and dynamic disaggregated memory approaches, with a maximum difference in quantile response time of 5%. For +60% overestimation (bottom row), the matching and underprovisioned systems show a reduced response time for the dynamic approach, as jobs are able to be scheduled more quickly, leading to a shorter waiting time in the queue. For underprovisioned systems, the median response time (y-axis equals 0.5), with overestimation, is reduced by 69%. This is because the dynamic approach releases unused resources and allows jobs to start earlier, therefore decreasing the job response times.

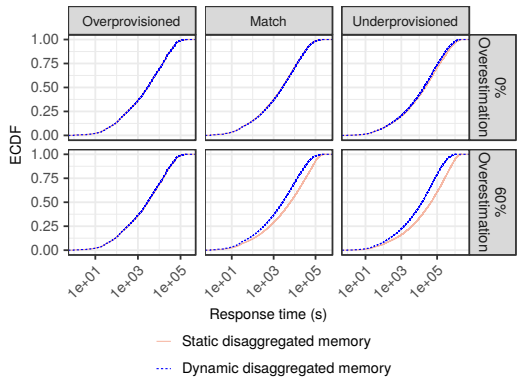


Figure 6: Empirical cumulative distribution of response time for different systems and job mixes. For +60% overestimation and underprovisioned systems (bottom right), the dynamic approach has a 69% lower median response time (note: logarithmic x-axis).

4.3 Cost–benefit analysis

Figure 7 shows the results of the cost–benefit analysis, assuming the component costs given in Table 4. The y-axis is the throughput (jobs per second) per dollar and the x-axis is the percentage of large jobs. As before, the top row is for +0% overestimation and the bottom row is for +60% overestimation. Different system configurations are shown in different panels from left to right. The conservative approach is a system with 100% memory provisioning (128 GB per node), shown in the left-hand panels.

Depending on the expected memory demands of the jobs during the production, the operator must choose a memory provisioning, which corresponds to choosing one of the panels from left to right. If it is expected that most jobs will have small memory demands, then the demand will be for 0% large jobs, which is the leftmost point on the x-axis of each panel. Choosing the 25% memory (top-right panel), rather than the 100% memory (top-left panel) improves

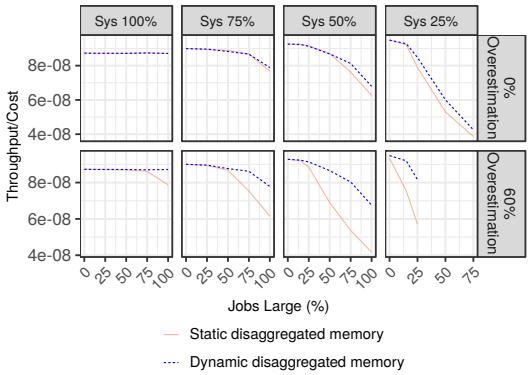


Figure 7: Cost–benefit analysis: throughput per cost (y-axis) as a function of the job mix (x-axis). The dynamic approach has a gentler drop in throughput when memory demand is high, reducing the risk of memory underprovisioning.

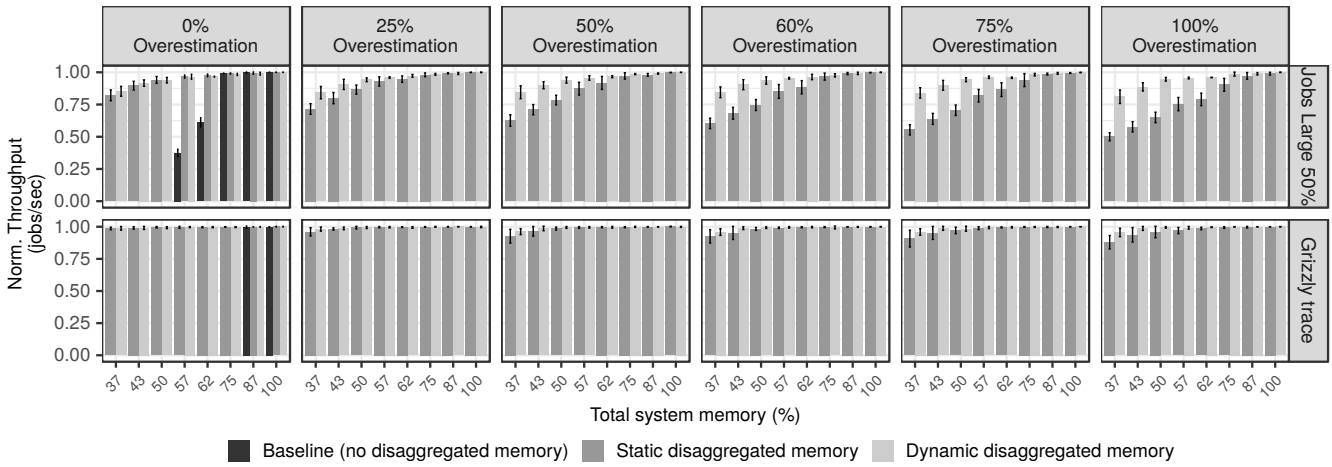


Figure 8: Effect of memory overestimation on throughput. Each panel is a different overestimation factor, showing throughput (y-axis) vs. total system memory (x-axis). Missing bars in the plots indicate there are not enough large memory nodes to run all the jobs. Compared with the static approach, the dynamic approach is less affected by memory overestimation.

throughput-per-dollar by 8%, which is seen by comparing the left-most datapoint in the two panels. But the underprovisioned system is sensitive to the job mix, because during periods with high proportions of large memory jobs, the throughput drops dramatically, which is seen by the slope of the curve. The cost–benefit calculations for the static and dynamic approaches are similar, but the dynamic approach consistently achieves slightly better throughput by up to 8%.

With a realistic +60% overestimation in job memory demands, seen in the lower row of panels, the static approach has a much steeper fall off in throughput due to large memory jobs, while the dynamic approach has behavior that is roughly the same as in the top row. The gentler fall off for the dynamic approach reduces the risk of provisioning a system with less memory and it improves the throughput per dollar by up to 38%.

4.4 System throughput vs. overestimation

Figure 8 shows the throughput (y-axis) as a function of the system memory capacity (x-axis). Running from left to right, each panel shows a different amount of overestimation, from +0% to +100%. The top row is for the synthetic trace with 50% large jobs and the bottom row is for the Grizzly trace. We observe that for the baseline case (+0% overestimation), the static and dynamic approaches have similar performance, with the difference appearing when the system is underprovisioned. However, the difference shows up when the jobs start to overestimate their demands. It becomes more compelling when the systems are underprovisioned to run the job mix (on x-axis systems below *75%* total memory). The dynamic approach experiences slowly the effects of overestimation compared to the static approach. We can clearly see that having a mechanism to release unused memory is beneficial to the system with fewer resources as it is able to run more jobs concurrently. For the worst case (+100% overestimation) the difference between static and dynamic approaches is over 38% on a system with 37% of its total memory. In this scenario, the dynamic is even able to keep the throughput over 80%. The dynamic approach is able to run higher load demands on less resources, therefore reducing the investment in resources.

4.5 Minimizing memory for defined throughput

Figure 9 shows the amount of memory resource necessary to keep the system throughput at a desired threshold (95% of the baseline throughput). We see that the static approach needs more resources to meet the threshold as we increase the amount of memory overestimation. On the other hand, the dynamic approach can reach 95% of the throughput using further underprovisioned systems even doubling the memory demands. In the best case, the dynamic achieves the threshold, saving almost 40% more memory than the static approach. The dynamic approach is able to maintain close to maximum throughput with much fewer resources even in the presence of overestimation.

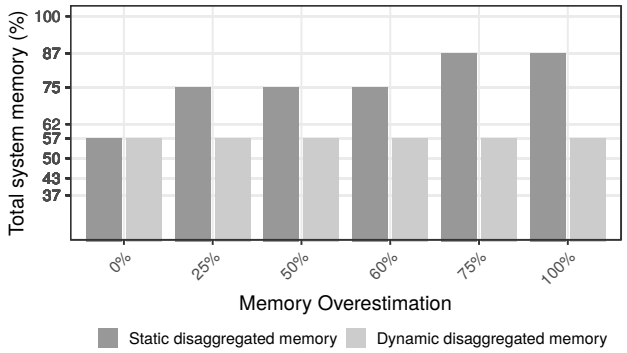


Figure 9: System resource provisioning (y-axis) as a function of the memory demand overestimation (x-axis) to achieve 95% of the fully provisioned throughput. Results are shown for synthetic trace with 50% large jobs.

5 RELATED WORK

Memory disaggregation systems. Peng *et al.* [30] implement a user-space remote paging library for disaggregated memory to handle nodes with fast, small local memories and large, slow remote memories. The library evicts local pages and retrieves remote pages when local memory is full. The UNIMEM [20] approach was developed by the EUROSERVER [14], ExaNoDe [33] and EuroEXA [15] projects. It implements a global physical address space for Arm architecture accessible by load–store instructions and RDMA.

Resource allocation/usage. Amaro *et al.* [9] increase throughput via a swapping mechanism using remote memory over RDMA. Their remote memory-aware scheduler splits each job's memory demand between local and remote memory. Amaral *et al.* [8] develop a controller to manage disaggregated resources and an algorithm to optimally place workloads in virtualized data-centers. Li *et al.* [23] propose a system software layer that exposes a CXL memory pool as a zero-core virtual NUMA node. Simulations show that the memory capacity of a cloud system can be reduced up to 10%. Michelogiannakis *et al.* [25] quantifies the appropriate level of disaggregation for HPC workloads. They demonstrate that resource reduction would satisfy the worst-case average rack utilization.

Dynamic resource assignment. Pinto *et al.* [31] present *ThymesisFlow*, a software-defined disaggregated memory prototype that uses a compute node that accesses remote memory and a memory node that exposes part of its local memory as disaggregated memory. Koutsovasilis *et al.* [22] memory policy migrates memory pages between local and disaggregated memory to increase performance compared with swap. They also introduce a memory orchestration stack that monitors the state of each node and scales its allocation of disaggregated memory according to the node's memory use. We differ from [22, 31] as we deal with a large-scale HPC system and adapt the job to the system through a resource manager capable of modifying the allocated memory. D'Amico *et al.* [12] present a dynamic job scheduling policy integrated into the Slurm resource manager. They implement a variant of backfill to leverage minimizing the system slowdown and co-scheduling malleable jobs. Iserte *et al.* [17] provides an approach to dynamically reconfigure the jobs' size by enhancing the collaboration between the OmpSs runtime and Slurm resource manager. We also differ from [12, 17] as we dynamically reassign memory but not compute resources.

6 CONCLUSION

Disaggregated memory breaks the rigid boundaries between nodes to provide memory as a system-wide pooled resource. State-of-the-art resource management systems for disaggregated memory statically allocate memory to each job, according to the memory requirement specified at job submission time. This paper makes a case for a dynamic approach, which adapts to the actual memory usage, improving throughput and waiting time, and increasing throughput per dollar by up to 38%. It reduces the need for the user to provide an accurate bound on the memory footprint. Our experiments are based on publicly available traces, and our implementation and methodology are available as open source [6, 44] in the hope that others reproduce and build upon our work.

ACKNOWLEDGMENTS

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 754337 (EuroEXA); it has been supported by the Spanish Ministry of Science and Innovation (project TIN2015-65316-P and Ramon y Cajal fellowship RYC2018-025628-I), Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), and the Severo Ochoa Programme (SEV-2015-0493).

REFERENCES

[1] 2021. Slurm Simulator. https://github.com/BSC-RM/slurm_simulator. Accessed: 2021-01-20.
[2] 2021. The Standard Workload Format. https://www.cs.huji.ac.il/labs/parallel/workload/swf.html. Accessed: 2021-01-20.
[3] 2022. Google Cloud Adds Compute, Memory-Intensive VMs. https://www.sdxcentral.com/articles/news/google-cloud-adds-compute-memory-intensive-vms/2019/08/. Accessed: 2021-03-22.
[4] 2022. LANL CTS-1 Grizzly - Tundra Extreme Scale, Xeon E5-2695v4 18C 2.1 GHz, Intel Omni-Path. https://www.top500.org/system/178972. Accessed: 2022-01-20.
[5] 2022. Memory statistics from open clusters - LA-UR-19-28211. https://usrc.lanl.gov/data/LA-UR-19-28211.php. Accessed: 2022-01-20.
[6] 2023. Disaggregated memory Slurm simulator and allocation policy. https://github.com/felippezacarias/slurm_simulator. Accessed: 2023-04-08.
[7] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. 2014. 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*.
[8] Marcelo Amaral, Jordà Polo, David Carrera, Nelson Gonzalez, Chih-Chieh Yang, Alessandro Morari, Bruce D'Amora, Alaa Youssef, and Malgorzata Steinder. 2021. DRMaestro: orchestrating disaggregated resources on virtualized data-centers. *Journal of Cloud Computing* (2021).
[9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the Fifteenth European Conference on Computer Systems*.
[10] Steve J Chapin, Walfredo Cirne, Dror G Feitelson, James Patton Jones, Scott T Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. 1999. Benchmarks and standards for the evaluation of parallel job schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*.
[11] Walfredo Cirne and Francine Berman. 2001. A comprehensive model of the supercomputer workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. IEEE.
[12] Marco D'Amico, Ana Jokanovic, and Julita Corbalan. 2019. Holistic slowdown driven scheduling and resource management for malleable jobs. In *Proceedings of the 48th International Conference on Parallel Processing*. ACM.
[13] David H Douglas and Thomas K Peucker. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization* (1973).
[14] Yves Durand, Paul M Carpenter, Stefano Adami, Angelos Bilas, Denis Dutoit, Alexis Farcy, Georgi Gaydadjiev, John Goodacre, Manolis Katevenis, Manolis Marazakis, et al. 2014. Euroserver: Energy efficient node for european microservers. In *17th Euromicro Conference on Digital System Design*.
[15] EuroEXA project. 2009. H2020 project number 754337. Accessed: 2021-09-20.
[16] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.
[17] Sergio Iserte, Rafael Mayo, Enrique S Quintana-Ortí, Vicenç Beltran, and Antonio J Peña. 2017. Efficient scalable computing through flexible applications and adaptive workloads. In *International Conference on Parallel Processing Workshops (ICPPW)*. IEEE.
[18] Changyeon Jo, Hyunik Kim, Hexiang Geng, and Bernhard Egger. 2020. RackMem: a tailored caching layer for rack scale computing. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*.
[19] Ana Jokanovic, Marco D'Amico, and Julita Corbalan. 2018. Evaluating SLURM simulator with real-machine SLURM and vice versa. In *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*.
[20] Nikolaos D Kallimanis, Manolis Marazakis, and Manolis Skordalakis. 2018. Use-cases for Remote Memory in the Unimem Architecture. In *ExascaleHPC: the ExaNoDe, ExaNeSt, EcoScale, and EuroEXA projects workshop at HiPEAC*.
[21] Vamsee Reddy Kommareddy, Amro Awad, Clayton Hughes, and Simon David Hammond. 2018. Exploring Allocation Policies in Disaggregated Non-Volatile

Memories. In *Proceedings of the Workshop on Memory Centric High Performance Computing*. ACM.

[22] Panos Koutsovasilis, Michele Gazzetti, and Christian Pinto. 2021. A Holistic System Software Integration of Disaggregated Memory for Next-Generation Cloud Infrastructures. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE.

[23] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. 2022. First-generation Memory Disaggregation for Cloud Platforms. *arXiv preprint arXiv:2203.00241* (2022).

[24] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *SIGARCH Computer Architecture News*. ACM.

[25] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh, Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf. 2022. A Case For Intra-rack Resource Disaggregation in HPC. *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[26] Rajiv Nishtala, Paul Carpenter, and Xavier Martorell. 2019. Performance effects on HPC workloads of global memory capacity sharing. In *MULTIPROG*.

[27] Emmanuel Kayode Akinshola Ogunshile. 2018. Viability of Small-Scale HPC Cloud Infrastructures.. In *CLOSER*.

[28] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. 2019. Quantifying memory underutilization in HPC systems and using it to improve performance via architecture support. In *International Symposium on Microarchitecture*.

[29] IB Peng, I Karlin, MB Gokhale, K Shoga, M Legendre, and T Gamblin. 2021. *A Holistic View of Memory Utilization on HPC Systems: Current and Future Trends*. Technical Report. Lawrence Livermore National Lab (LLNL), Livermore, CA.

[30] Ivy Peng, Roger Pearce, and Maya Gokhale. 2020. On the Memory Underutilization: Exploring Disaggregated Memory on HPC Systems. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE.

[31] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. 2020. 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.

[32] Urs Ramer. 1972. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing* (1972).

[33] Alvise Rigo, Christian Pinto, Kevin Pouget, Daniel Raho, Denis Dutoit, Pierre-Yves Martinez, Chris Doran, Luca Benini, Iakovos Mavroidis, Manolis Marazakis, et al. 2017. Paving the way towards a highly energy-efficient and highly integrated compute node for the Exascale revolution: the ExaNoDe approach. In *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE.

[34] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*.

[35] Konstantin S. Solnushkin. 2013. Automated Design of Torus Networks. *CoRR* (2013). arXiv:1301.6180 http://arxiv.org/abs/1301.6180

[36] Konstantin S Solnushkin. 2014. SADDLE: A modular design automation framework for cluster supercomputers and data centres. In *International Supercomputing Conference*. Springer.

[37] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, Ilias Syrigos, Maciej Bielski, Dimitris Theodoropoulos, Dionisios N Pnevmatikatos, and Georgios Zervas. 2017. A software-defined architecture and prototype for disaggregated memory rack scale systems. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*.

[38] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. 2019. Improving HPC system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*.

[39] Mohammed Tanash, Huichen Yang, Daniel Andresen, and William Hsu. 2021. Ensemble Prediction of Job Resources to Improve System Performance for Slurm-Based HPC Systems. In *Practice and Experience in Advanced Research Computing*.

[40] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*.

[41] Andy Turner and Simon McIntosh-Smith. 2017. A survey of application memory usage on a national supercomputer: an analysis of memory requirements on ARCHER. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer.

[42] John Wilkes. 2020. *Google cluster-usage traces v3*. Technical Report. Google Inc., Mountain View, CA, USA. Posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[43] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple Linux utility for resource management. In *JSSPP*. Springer.

[44] Felippe Zacarias, Paul Carpenter, and Vinicius Petrucci. 2023. Artifact for Dynamic memory provisioning on disaggregated HPC systems. https://doi.org/10.5281/zenodo.7881019

[45] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. 2021. Improving HPC System Throughput and Response Time using Memory Disaggregation. In *International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE.

[46] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. 2021. Memory Demands in Disaggregated HPC: How Accurate Do We Need to Be?. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*.

[47] Felippe Vieira Zacarias, Rajiv Nishtala, and Paul Carpenter. 2020. Contention-aware application performance prediction for disaggregated memory systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*.

[48] Georgios Zervas, Hui Yuan, Arsalan Saljoghei, Qianqiao Chen, and Vaibhawa Mishra. 2018. Optically disaggregated data centers with minimal remote memory latency: technologies, architectures, and resource allocation. *Journal of Optical Communications and Networking* (2018).

[49] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* (2017).