Improving HPC System Throughput and Response Time using Memory Disaggregation

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
University of Pittsburgh
Pittsburgh, United States
vpetrucci@pitt.edu

***Abstract*—HPC clusters are cost-effective, well understood, and scalable, but the rigid boundaries between compute nodes may lead to poor utilization of compute and memory resources. HPC jobs may vary, by orders of magnitude, in memory consumption per core. Thus, even when the system is provisioned to accommodate normal and large capacity nodes, a mismatch between the system and the memory demands of the scheduled jobs can lead to inefficient usage of both memory and compute resources.**

Disaggregated memory has recently been proposed as a way to mitigate this problem by flexibly allocating memory capacity across cluster nodes. This paper presents a simulation approach for at-scale evaluation of job schedulers with disaggregated memories and it introduces a new disaggregated-aware job allocation policy for the Slurm resource manager. Our results show that using disaggregated memories, depending on the imbalance between the system and the submitted jobs, a similar throughput and job response time can be achieved on a system with up to 33% less total memory provisioning.

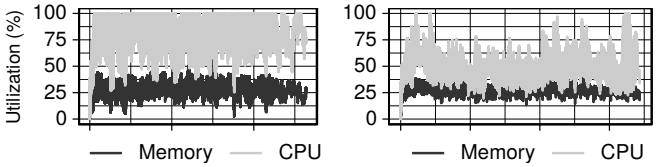
***Index Terms*—Disaggregation, Performance degradation, Performance prediction, Resource scheduling, Slurm**

I. Introduction

Over 90% of the High Performance Computing (HPC) systems in the TOP500 list are built using a cluster architecture. HPC clusters are cost-effective, well understood and scalable to thousands of nodes. In a cluster architecture, the coordination of all hardware and software falls under the control of the resource management software. It is a key component for the distribution of computing power within the cluster infrastructure. The management goal is to satisfy users demands for computation and achieve a acceptable performance in overall system's utilization by efficiently matching requests to resources.

In existing HPC systems, the rigid boundaries between compute nodes limits compute and memory resource utilization. HPC applications are rarely co-located on a compute node [1], so they have exclusive access to self-contained servers, and any of the node resources that are not used by the running application cannot be made available to other applications. This problem of stranded resources is especially critical for memory [2] because HPC application memory demands vary dramatically, by orders of magnitude, due to application characteristics and strong scaling [3], [4].

Fig. 1 shows an example timeline of total system memory and Central Processing Unit (CPU) utilization. In Fig. 1a, the mix of jobs matches the memory provisioning (system has 25% large capacity nodes and 25% of job CPU hours need large capacity nodes),[1] and average CPU utilization is high, at 81%. Fig. 1b shows the same system, but this time 50% of the jobs need large capacity nodes. In this case, both CPU and memory have low utilization (both averages less than 48%). The system clearly has abundant unused CPU and memory resources, but they are not available for use by the applications.



(a) System matches job mix (b) System mismatches job mix

Fig. 1. Resource utilization when the system matches jobs demands and when there is a mismatch.

Disaggregated memory has recently been proposed to allow a flexible and fine-grained allocation of memory capacity to compute jobs [5], [6], [1]. In this direction, this paper proposes an extension to the Slurm job manager to allocate memory capacity to jobs in a disaggregated memory system. Research in job scheduling cannot easily be done using a production system, and in any case, disaggregated memory prototypes are still at research level and system software is immature. We therefore extend an existing simulation approach using Slurm to account for memory/network bandwidth contention in disaggregated memory leveraging an extension of a slowdown based method [7]. We then use the extended Slurm simulator to determine overall system throughput, job queuing and execution time of a large-scale HPC system.

In summary, we make the following contributions:

1) **Slowdown-based method** to predict performance degradation due to sharing of disaggregated memory across multiple nodes in a cluster;
2) **Extended job scheduler simulator** to support disaggregated memory in the most-used workload manager, Slurm;
3) **Disaggregation-aware allocation policy** implemented in the Slurm workload manager;

[1]Details of the experimental evaluation are in Section V.

4) **At-scale evaluation** of our slowdown methodology and allocation policy for disaggregated memories using the Slurm simulator.

Our results show that our multi-node slowdown methodology is a good approximation for predicting the performance with a maximum error of 14%. Using a disaggregated memory approach, similar overall system throughput and job response time (waiting time plus execution time) can be achieved when compared to an existing HPC system, while using up to 33% less memory, depending on the imbalance between the system and the memory demands of the submitted jobs. The Slurm simulator extension and allocation policy are released open source [8].

II. Background

A. Disaggregated memory

Disaggregated memory has been proposed, and is under investigation in both academic and commercial research, so as to address two main problems. Firstly, current HPC systems provide little flexibility in provisioning memory, because DIMMs should be installed in a balanced way across a small number of memory controller channels, leading to coarse-grained rules of thumb like the common 2 GB per core [3]. Secondly, HPC clusters suffer from stranded resources because memory that is not used by one job cannot be used by another on a different node.

In the disaggregated design, individual components such as processor, memory and storage are interconnected over a network to share memory but without cache coherent data sharing [9], [10], [11]. The EUROSERVER [10], ExaNoDe [12] and EuroEXA [13] family of projects has pioneered a disaggregated system architecture, which provides a global physical address space and the ability for cores to access remote memory via Remote Direct Memory Access (RDMA) or direct load–store instructions. By appropriately configuring the cache policy, remote memory accesses can be cached locally.

In our work we use the model disaggregated architecture inspired by the UNIMEM approach [10]. In this design, computing units execute their own Operating System (OS), and can access memory attached to it (local memory access) as well as memory attached to another computing unit through a global interconnect (remote memory access). The remote memory access is performed through a common Global Address Space, either using ordinary load–store instructions or RDMA. The design supports caching locally at the unit that requested the access or remotely at the unit attached to the memory. For disaggregated memory, the data should be cached locally.

B. Slurm Resource Manager

Slurm [14] is a widely used open-source HPC resource and job management system. It has a multi-threaded core and a plug-in module architecture, which makes it configurable with a variety of extensions for workload, queueing, scheduling, etc. It also has a centralized manager responsible for allocating resources, monitoring job execution and mediating contention to resources through a queue of pending jobs. The default node allocation of Slurm is the exclusive mode, so even if not all resources within the node are utilized by a specific job, no other job is allowed to share the resource. Even though Slurm allows fine-grained cluster management, requests exceeding the available memory per node cannot execute and nodes with free memory but no cores will not be used on the scheduling.

C. Slurm Simulator

Optimizing the job scheduler and its policies for HPC system performance and user experience is a complex, multi-dimensional problem. It is impractical to perform large-scale experiments on a real production machine, since doing so will likely negatively impact the service delivered to users. The Slurm simulator proposed by [15], [16] enables a precise and deterministic evaluation of the job scheduler by running it in a simulation environment. It is based on the original Slurm source code so, unlike theoretical models, it is able to capture all parameters and behavior that occurs in a real environment.

The simulator receives as input a standard Slurm configuration file and a trace capturing the HPC job submissions and actual execution times. The configuration file specifies the number of nodes and queues, selection and scheduling policies, and so on. The trace binary input used for the simulation is based on the Standard Workload Format (SWF) [17], [18], which is a standardized way to describe the submission of jobs to a system. The simulator can therefore use existing real logs or traces from synthetic workload generators that are publicly available in on-line repositories.

III. Related Work

Memory Disaggregation — Gu *et al.* [5] implement a scalable and decentralized remote memory paging solution to enable memory disaggregation. It divides the swap space of each machine and distributes the pages across many remote machines using RDMA operations for all remote I/O operations. Shan *et al.* [19] propose a split kernel OS architecture to manage disaggregated systems. It breaks the OS into pieces with different functionalities, each running on and managing a hardware component. Peng *et al.* [1] implement a user-space remote paging library to allow exploration of applications using disaggregated memory. Their architecture contemplates nodes with fast but small local memories and large but slow remote memories, and it is aided by the library, which evicts local pages and fetches remote pages when the local memory is exhausted.

Amaral *et al.* [20] develop a dynamic loop-based controller to manage resources and a flow-network algorithm to determine the optimal placement of workloads on virtualized datacenters. Their approach disaggregates Graphics Processing Unit (GPU) using middleware that intercepts GPU calls and offloads its data via the network. Amaro *et al.* [21] present a swapping mechanism that uses remote memory through RDMA and a remote memory-aware cluster scheduler to split job's memory demand between local and remote memory. Then, they examine the scenarios where remote memory can increase job throughput.

Slowdown based methods — De Blanche *et al.* [22], [23] propose a slowdown based characterization method to estimate application slowdown when sharing the memory bus. Bandwidth Bandit [24] proposes a quantitative profiling method for analyzing the performance impact of contention for shared memory resources to determine the application's sensitivity to latency and bandwidth. Zacarias *et al.* [7] propose a slowdown based methodology to predict the performance degradation from remote memory contention. They characterize a single node application sensitivity curve based on the contentious pressure from remote access and the ratio of read and write of the memory access.

While prior works [25], [22], [24] use application working set size or local bandwidth as their measure of pressure to create the sensitivity curve, our approach targets performance prediction due to sharing of disaggregated memory. Cache contention characterization method is misleading for predicting the performance of applications using separated cache hierarchies, since the remote access do not create cache contention in the local node. In spite of being proved successful for multithreaded single node applications, the methodology presented in [7] does not account for distributed applications and contention from multiple sources.

IV. Extending Slurm for disaggregated memory

A. Integration into Slurm Simulator

The Slurm simulator previously assumed no contention among jobs, in terms of network, CPU and memory. This is a reasonable assumption for non-disaggregated memory systems, due, firstly, to the independent nature of the compute nodes and, secondly, to the common use of non-blocking networks in HPC systems. This assumption simplifies the simulator because the execution time of each job is independent of the scheduling and allocation policies and is known in advance. The actual execution time of each job is recorded as one of the fields in the SWF trace file.

We modified the trace format to provide also the information needed by the memory access contention model (see Section IV-B). In fact, since many of the jobs were for similar applications, we use a unique identifier for each simulated application type. The trace file specifies the baseline execution time without contention and the application type identifier.

We modified the simulator to invoke the contention model each time any job starts or finishes. The contention model calculates the estimated performance of every job that potentially has contention with the starting or finishing job. The output of the contention model is the estimated performance of the job, P_{est}, where for example $P_{\text{est}} < 1$ whenever the job suffers slowdown and a value of P_{est} equal to 1 means that the job runs without contention with other jobs. The estimated performance is interpreted as the *speed* at which the original baseline runtime is executed. After each time period, the remaining runtime is updated based on the elapsed time and the speed during this interval, as given in Equation 1.

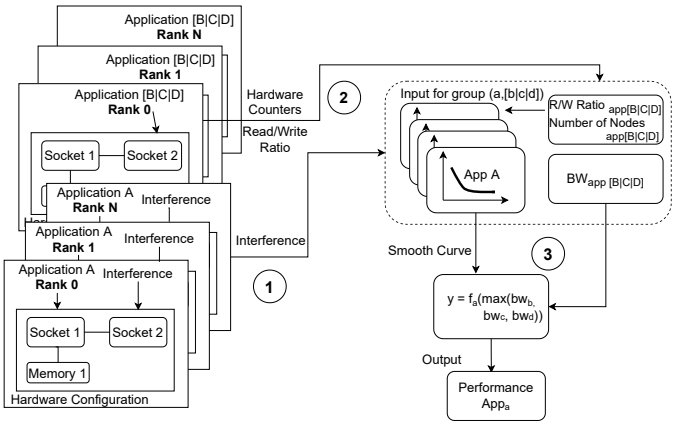


Fig. 2. Multi-node slowdown methodology.

$$runtime\_left' = runtime\_left - delta\_time \times P_{\text{est}} \quad (1)$$

B. Memory Access Contention Model

Our contention model is an extension of the model by Zacarias *et al.* [7], extended to support contention among multi-node applications. The overall approach and the model inputs and output are shown in Fig. 2.

In common with all slowdown-based models, the single-node model quantifies application performance using a sensitivity curve, which measures performance on the y-axis, normalized to the performance running alone, as a function of the contentiousness of the other application(s) on the x-axis. The contentiousness is a single variable, which for a single node is the total memory bandwidth. In our model of disaggregated memories, remote memory accesses do not create cache contention in the local node as their cache hierarchies are separate. Moving from single-node to multi-node greatly increases the complexity, because in the multi-node case, there is a separate interfering memory bandwidth per node, which is impractical to model in detail.

Most HPC applications have similar behavior on each node, and overall performance is constrained by the slowest node. For this reason it is reasonable to set the contentiousness to be the largest, i.e. worst interfering bandwidth across all nodes. However, we found that the actual level of memory bandwidth interference is subject to a reasonable amount of noise, and performance degrades as the number of interfering nodes is increased. We therefore count the number of nodes that have interference close to the maximum across all nodes, and use a family of sensitivity curves indexed by this number of nodes.

To extend the contention model, we first execute the synthetic benchmark [26] to create the sensitivity curve in parallel across a configurable number of interfering nodes (Step 1 of Figure 2). Then, we measure the sensitivity curve for 50% reads and 100% reads and use linear interpolation for intermediate R/W ratios. Prior work has shown that linear interpolation exhibits better performance than additional interfering data points [7], and that the accuracy was similar to

polynomial interpolation. Following a similar concept, so as to decrease the cost of collecting the sensitivity curve data, the number of interfering nodes was sampled between 1 and the maximum target number of nodes.

In the second part, the contentiousness of an application , bw_{app}, is collected using performance counters when running alone (Step 2). We calculate the read and write memory bandwidths using the numbers of read and write Column Access Strobe (CAS) commands [27], averaged over all nodes on which the application is executed. In the last part (Step 3) the model predicts the performance of an application "A" using its interpolated sensitivity curve, f_a, based on the read/write percentage, number of interfering nodes and the largest contentiousness, $max(bw_a, bw_b, bw_c)$, among the interfering applications.

C. Resource Selection

In Slurm, jobs ready for scheduling are selected from the global queue of pending jobs. All available nodes having enough resources, such as cores and memory capacity, are selected for further evaluation. At this point, memory is only a constraint if the requested memory is greater than the node's physical memory. Then, the resource manager's *Selection plugin* determines the nodes that best satisfy the request.

A major characteristic that prevents Slurm using disaggregation is that it has a processing and server-based architecture. Memory management is tightly coupled with availability of CPU cores, despite being configured as a controlled resource. This means that nodes without idle cores are excluded from allocations, even if there is unused memory capacity. To improve utilization and throughput of the system, we adjust the scheduling and resource selection to support the use of remote memory capacity across the cluster to create the disaggregated infrastructure for the resource manager.

Since the usage of remote memory across the cluster impacts applications, our validation through simulating the platform requires some degree of consideration for application's performance running in such configuration. To this end, we integrated the developed multi-node slowdown based method described in Section IV to characterize the slowdown experienced by the applications sharing memory resources.

D. Supporting Memory Disaggregation

After our analysis of the job scheduling process described in Section IV-C, we modified the verification performed by the job scheduling process to build the list of nodes available to the job. While the default allocation (baseline) used by Slurm returns an error when no nodes can satisfy the request or removes nodes with less free memory than is required by the job, our allocation approach differs substantially. We separate into distinct lists the nodes with available cores and memory. From this point forward, we can adopt several strategies to allocate memory that will impact application performance.

For explanatory reasons, Fig. 3 shows a schematic simple case to exemplify the best allocation strategy explored in this work. The simulated HPC systems used to present the results are described in Section V-C. The schematic figure shows a small heterogeneous system with *10* nodes, equally divided into *normal* and *large* nodes. *A, B, C* and *D* represent the order of jobs submitted to this system with different node and memory requirements.

To mitigate the issues experienced with previous strategies, we use the baseline allocation method that increases the local to remote memory ratio, and we employ the disaggregated strategy when there are insufficient nodes to satisfy the current request or a resource-hungry job is scheduled. The baseline strategy selects all nodes that have enough local memory to satisfy the job's requirement of memory per node to avoid unnecessary remote memory access. In Fig. 3, jobs *A, B*, and *D* use only local memory, since the approach is able to find the best node that satisfies the job's memory-per-node request. On the other hand, job *C* requires more memory than any node in the system. To schedule this request we use our disaggregated approach employing remote memory access.

We do not use the CPU cores of nodes that have already lent memory to another node. This means that such a node effectively becomes a memory node for other jobs. We experimented with relaxing this condition, and found that there was no improvement in the results. It is always preferable to use local memory when it is available. Our approach, then, to increase the local to remote ratio, favors nodes with higher memory available applying a weight to each node based on their free memory. Then, nodes with higher local memory available are selected, consequently decreasing the influence of remote memory access. As an example, instead of using *normal* nodes to satisfy job's *C* request, the approach uses *large* nodes, thus increasing the local memory usage. For a new submitted job, node *N9* will be a memory node since it has some memory lent.

V. Experimental Methodology

A. Environment Setup

Hardware resource — We carried out the experiments on a cluster which has servers equipped with two Intel Xeon SandyBridge-EP E5-2670 that together comprise 16 cores operating at 2.6 GHz. Each socket has 20 MB L3 cache (LLC) shared among all cores, single memory controller, and two Quick Path Interconnect (QPI) links version 1.1 operating at 8.0 GT/s. It implements the home snoop cache coherence with MESIF protocol [27]. The node has 64 GB of DDR3-1600 DIMMs, theoretical bandwidth of 51 GB/s (37 GB/s sustained) for local access and 38 GB/s (20 GB/s sustained) for remote memory access. The memory access latency is 81 ns and 133 ns for local and remote accesses respectively [28].

Benchmarks — We used nine distributed applications from several known benchmark suites. For the simulation, we incorporated the detailed profile from a total of 44 single-node applications from PARSEC (8 applications) [29], Rodinia (5) [30], NAS Parallel Benchmarks (NPB) (8) [31], Splash (5) [32] and another 15 diverse publicly available applications. We selected applications to cover a variety of computational

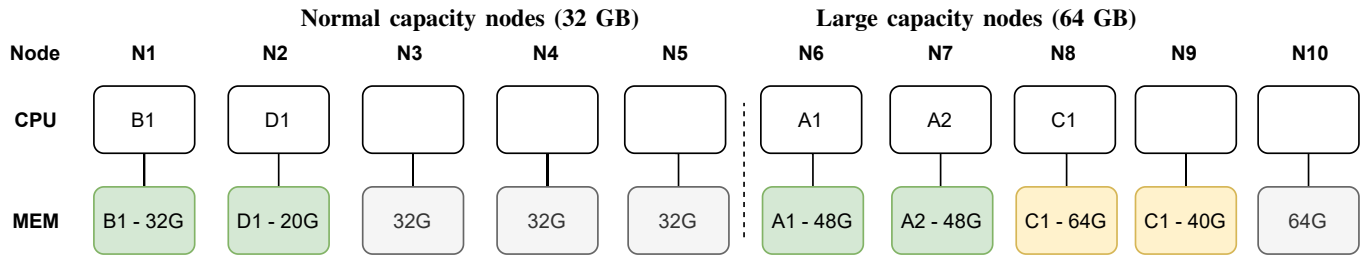


Fig. 3. Graphical scheme of the memory allocation explored in this work for a simple case considering a small system with half of its nodes having 32 GB (5 nodes on the left) and 64 GB of memory (5 nodes on the right). The system configurations used in the evaluation are much larger and presented in Section V-C.

patterns found in multithreaded and high performance codes. The single node applications were compiled with GNU/Linux GCC 7.2 and multithreading enabled, while the distributed applications were compiled using OpenMPI 1.8.1. We used *numactl* to apply affinity settings for threads and memory placement, and *Perf* tool to collect the performance counters.

B. Emulating Disaggregated Memory

Due to the absence of an available prototype of remote memory decoupled from processor [33], we follow the emulation approach proposed by Zacarias *et al.* [7]. This way, we could emulate a disaggregated shared memory architecture, without the need for real hardware, using conventional multi-socket servers. This approach takes advantage of a two-socket server and its separate LLC to create pressure only in the desired shared resource. According to Molka *et al.* [28], cache coherence traffic is not a significant bottleneck in a two-socket system. Thus, in our approach the processor and cache resources are isolated from interference while the effects of memory bandwidth contention can be observed on the shared memory resource. In addition, the latency of cross-socket memory access for our experiments is similar to those presented in disaggregated works such as [6], [34].

All threads of an interfering application (B) execute on socket 2 while issuing memory requests exclusively to the memory bank attached to socket 1 (remote access). On the other hand, all threads of a target application (A) use only its memory bank (local access), thus contending for memory controller and memory bandwidth. Thus, the impact of application B on application A can be modeled based in B's bandwidth interference. For distributed applications running across different nodes, a target application issues memory requests to the local memory bank on every node, while other applications may cause interference by issuing remote memory requests given by a particular contentiousness level and number of nodes. In this scenario, our model can predict the slowdown experienced by a given target application in the face of diverse remote bandwidth requirements.

C. System and Workload Configurations

We set up different configurations to explore heterogeneity in job requirements and node capacities. In our experiments, the HPC system is separated into *normal* nodes, which have typical memory capacity, and *large* nodes with twice the memory capacity of the *normal* nodes. To evaluate different scenarios and heterogeneous systems, we experiment with multiple ratios between large and normal nodes, varying from 0% (all normal nodes) to 100% (all large nodes). The systems have a total of 1024 nodes each having 32 cores and 32 GB (*normal* node) or 64 GB (*large* node) of memory, Slurm is configured to use the baseline or disaggregated select resource policy. The parameters for job scheduling are the same for all experiments.

We generated synthetic workloads using the CIRNE Comprehensive Model [35]. This model is based on an analysis of workload traces generated by real environments. It includes arrival pattern, requested time, job sizes, system load, status and start and finish times. In our work, we need information about the memory capacity requested by each job. Thus, we augmented the set of generated traces with memory information from the applications profiled in our real environment.

First, we generate the synthetic trace using the CIRNE Model for the required system size (Step 1). Alternatively, we use a pool of executed applications for which we have a profile regarding size, runtime, memory bandwidth, read/write ratio, local/remote access memory ratio and memory capacity requested (Step 2). Using the trace and app lists, we calculate Euclidean distances to map each real application to a similar synthetic job based on its size and runtime (Step 3). Finally, we generate the new augmented trace by the assigned arrival time (Step 4), converting it to a binary readable by the simulator (Step 5). At the end, we have a new input trace preserving the synthetic trace info that includes a memory capacity required and an identifier for the job application. The memory capacity will be used for resource scheduling, while the application identifier will be used in our multi-node slowdown based method to calculate the slowdown suffered by this particular job due to resource sharing.

We generated additional input trace files, each targeting one of the specific heterogeneous system ratios.We ensure that all traces have total *node–hours* (#nodes $\times$ runtime) of large and small jobs in the indicated ratio. The characteristics of the large and small jobs are given in Table I. All normal jobs have memory demand less than the capacity of a normal node, whereas all large jobs have memory demand greater than a normal node. In terms of baseline node–hours, the normal jobs are typically larger than the large memory jobs. We generate the input traces for the simulator by sampling without

replacement, in the appropriate proportions, from these two distributions.

TABLE I
LARGE AND SMALL JOB CHARACTERISTICS

	Normal Jobs		Large Jobs	
Metric	**Memory (GB)**	**Node–hours**	**Memory (GB)**	**Node–hours**
Min	0.12	0.0	33.0	0.0
1st Qu.	1.7	0.85	48.2	0.0
Avg	6.2	52.6	48.5	24.9
3rd Qu.	3.8	15.0	49.8	2.1
Max	27.6	6412	49.8	3659.0

VI. EXPERIMENTAL RESULTS

A. Prediction of Multi-node Slowdown

We evaluate the effectiveness of our multi-node slowdown methodology that predicts the degradation of a target application when it experiences different levels of interference while running on different nodes. We measure the prediction error, which is the absolute percentage difference between the predicted and real performance under resource contention.

In our experiments, we start both target and interfering applications at the same time on every node. Since we are dealing with distributed applications, the interfering applications may differ in terms of interference levels and number of nodes. During the experiments, if any interfering application on any node finishes, we restart it to keep the target application under contention throughout its entire execution. We continue the experiments until the target executes at least seven times. Once the target application ends, its performance degradation (delayed execution time) is recorded to be compared with the predicted performance under a resembling scenario. The degradation for an application is calculated using its normalized execution time alone in the system without interference.

Fig. 4 presents the results for a mix of interfering applications that vary in contentiousness (see Section IV-B), nodes, and read/write ratio. It presents the maximum error for several combinations of profiled applications when the target application runs locally. We notice that the max errors are lower than 10% for most of the applications. Even though we notice an increase in the prediction error when we move from 4 to 31 nodes for some applications (e.g. *stream, streamcluster, hpccg* and *hydro*), it is also noticeable that the maximum prediction error does not increase at the same rate as the number of nodes. Increasing the number of nodes from 4 to 31 ($\sim$8$\times$ increase), the maximum error for any application increased at most 3$\times$. Demonstrating that we do not compromise the accuracy of the model when increasing the number of nodes up to 31 nodes.

B. System Throughput

Leveraging the multi-node slowdown based model and the Slurm simulator, we evaluate the implemented disaggregated infrastructure described in Section IV-D simulating the different heterogeneous scenarios presented in Section V-C. In this step, we assumed that the scalability of our memory access contention model has the same behaviour presented

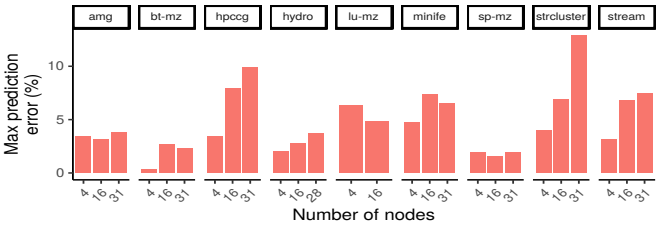


Fig. 4. Slowdown model prediction maximum error.

on Section VI-A when we scale the number of nodes. For every system configuration, we simulated different job mixes in terms of pressure on the large memory resource. Fig. 5 presents the throughput achieved for each simulated scenario when different job mixes are submitted. It is normalized towards the homogeneous 100% large nodes system, since it has enough resources to execute any job across all inputs.

The baseline approach is able to execute all job mixes except when the system has 0% large nodes, in which case the baseline cannot execute any large jobs as no node has enough memory. We therefore remove all baseline data points for the 0% system and job mixes except 0% large. For this reason, the x-axis in Fig. 5 has double bars showing the comparison between the baseline and disaggregated approaches except for the 0% system for job mixes with large jobs.

Fig. 5 shows a clear trend in the system's throughput based on the resources availability and the jobs mix. We can notice that for the baseline approach throughput is high when the job mix matches or is lower than the ratio of large and normal memory resources within the system. However, the baseline's throughput decreases substantially when the job mix has higher ratio of large memory jobs and the system is underprovisioned to satisfy the request. It indicates that the resource manager considers for allocation only a subset of nodes on the system that are able to run the large jobs, thereby the jobs waits longer to have access to the resources needed leaving aside other nodes. It contributes to increase the makespan and therefore for low utilization and throughput.

On the other hand, besides reaching the same throughput as the baseline when the mix of jobs matches the system or the system is overprovisioned, our approach increased the throughput compared to the baseline when the job mix runs on an underprovisioned system. It happens because our approach performs a disaggregated allocation that leverages the remote idle resources that are not used by other jobs or that is not possible using the baseline approach.

The memory savings provided by the disaggregated approach are noticeable. For a example, when the job mix has 50% large jobs, the baseline requires at least 50% of the nodes to have large capacity whereas the disaggregated approach has only 5% degradation with 0% large capacity nodes (Fig. 5). Since the large nodes have twice the memory capacity of the normal nodes, the disaggregated approach reduces the total memory capacity by 33%, compared with the baseline. The savings in the other scenarios are lower but still significant. For the 15%, 25% and 75% large job scenarios, the potential

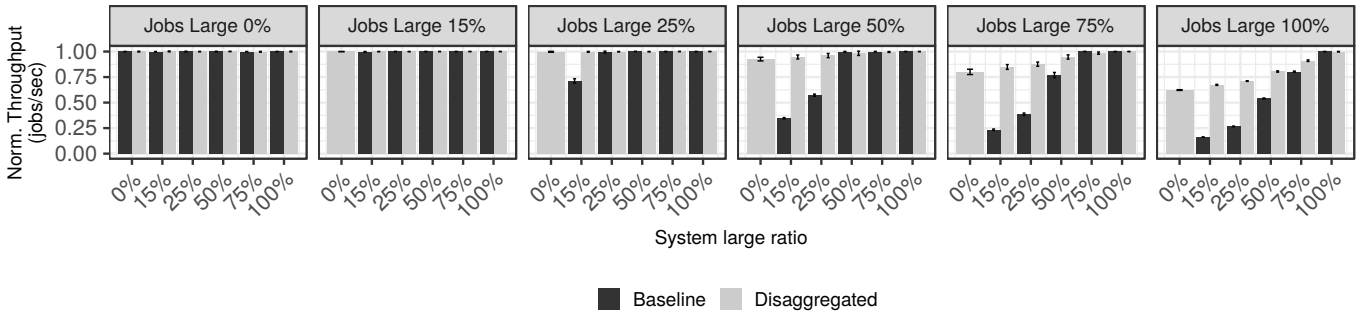


Fig. 5. Normalized throughput (y-axis) experienced by each simulated system (x-axis) for various job mixes.

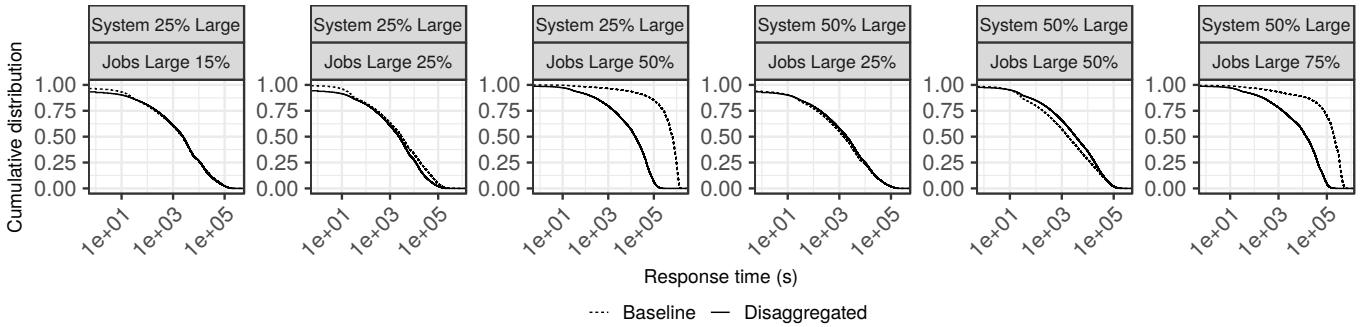


Fig. 6. Cumulative distribution of response time for two different systems and different job mixes.

memory savings are 14%, 20% and 15%.

C. System Response Time

Fig. 6 shows the cumulative distribution of the response time for two different systems and three job mixes. We show these scenarios for brevity since the others exhibit the same trend presented in this figure. When the job mixes match or run on an overprovisioned system, the approach's lines overlap showing similar performance. On the other hand, when the job mix stress more resources on an underprovisioned system, we notice that the baseline starts to increase its response time compared to our approach. The jobs will compete for a small number of resources hence increasing their waiting time. This performance penalty will start to be apparent to the users in the system as their submitted jobs will take longer to finish after its submission. The impact of decreasing resources is less noticeable with our approach as it presents lower probability of longer response times. Our approach leverages the idle resources that are deemed unable to run some jobs by the baseline, consequently decreasing the waiting time.

D. CPU and Memory System Utilization

Fig. 7 shows the CPU and memory utilization, across all executed scenarios. In all subplots the x-axis is the CPU utilization and the y-axis is the memory utilization, both relative to the maximum capacity of the memory and nodes of the system on which the trace is executed. The scenarios are divided into overprovisioned (job mix demands less large nodes than available), match (job mix demand equals number of large nodes), and underprovisioned (demands more). We

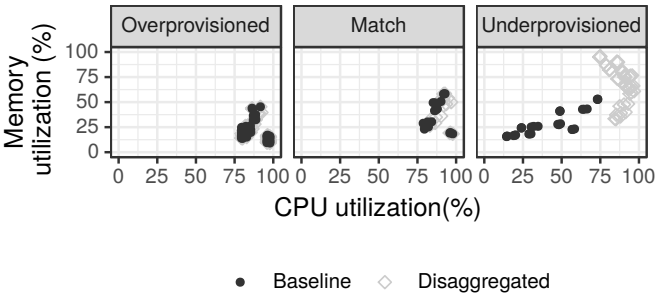


Fig. 7. Average memory and CPU utilization when using either Disaggregation or Baseline under different job/node requirements/capacities.

notice that when the system is overprovisioned to satisfy any submission of a job mix (left-hand side), our approach and the baseline have similar performance. In this scenario, both are constrained by CPUs, with a moderate utilization of memory. The same pattern goes for the scenarios where the job mix matches the system ratio (middle).

When there is a mismatch between the job mix and the system resource capacity (right-hand side) we see that the baseline performs poorly, and both memory and CPU have low utilization. This happens because the baseline is constrained by the number of large nodes, leaving normal node memory and cores idle and decreasing the overall utilization.

In contrast, our approach uses remote memory to satisfy the job requests, hence increasing CPU and memory utilization in the mismatched scenarios. The jobs are not constrained by the memory of a particular node, but by the total memory available within the system. On average, our approach increases the

memory utilization by a factor of 1.6, while having almost 90% of CPU utilization compared to the baseline.

VII. Conclusion

This paper investigates how a disaggregated-memory–aware job scheduler can make use of a disaggregated memory platform to maintain throughput and improve response time while using less total system memory. Since research in job scheduling requires a simulation platform that is both faster and less intrusive than running on a real system, this paper extended an existing Slurm simulator to support disaggregated memories. We developed a multi-node slowdown based method to quantify the impact of remote memory sharing on application performance and embedded this model into the Slurm simulator. We used the simulator to develop and evaluate at scale a disaggregated memory allocation policy implemented in Slurm. The results show that depending on the level of imbalance between the system and memory demands of scheduled jobs, memory disaggregation enables resource savings of up to 33% compared to the state-of-the-art resource manager. The Slurm simulator extension and allocation policy are released open source [8].

Acknowledgement

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] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *SBAC-PAD*, 2020.

[2] A. D. Papaioannou, R. Nejabati, and D. Simeonidou, "The benefits of a disaggregated data centre: A resource allocation approach," in *GLOBECOM*, 2016.

[3] D. Zivanovic, M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. Mckee, P. M. Carpenter, P. Radojković, and E. Ayguadé, "Main memory in HPC: Do we need more or could we live with less?" *ACM TACO*, 2017.

[4] R. Nishtala, P. Carpenter, and X. Martorell, "Performance effects on HPC workloads of global memory capacity sharing," in *MULTIPROG*, 2019.

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

[6] G. Zervas, H. Yuan, A. Saljoghei, Q. Chen, and V. Mishra, "Optically disaggregated data centers with minimal remote memory latency: technologies, architectures, and resource allocation," *Journal of Optical Communications and Networking*, 2018.

[7] F. V. Zacarias, R. Nishtala, and P. Carpenter, "Contention-aware application performance prediction for disaggregated memory systems," in *CF*, 2020.

[8] "Disaggregated memory slurm simulator." https://github.com/felippezacarias/slurm_simulator, 2021, accessed: 2021-04-08.

[9] M. Bielski, I. Syrigos, K. Katrinis, D. Syrivelis, A. Reale, D. Theodoropoulos, N. Alachiotis, D. Pnevmatikatos, E. Pap, G. Zervas *et al.*, "dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter," in *DATE*, 2018.

[10] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis *et al.*, "Euroserver: Energy efficient node for european micro-servers," in *17th Euromicro Conference on Digital System Design*, 2014.

[11] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *HPCA*, 2012.

[12] A. Rigo, C. Pinto, K. Pouget, D. Raho, D. Dutoit, P.-Y. Martinez, C. Doran, L. Benini, I. Mavroidis, M. Marazakis *et al.*, "Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: the exanode approach," in *Euromicro Conference on Digital System Design*, 2017.

[13] E. project, "H2020 project number 754337," 2009, accessed: 2019-10-16. [Online]. Available: https://euroexa.eu/

[14] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *JSSPP*. Springer, 2003.

[15] "Bsc slurm simulator," https://github.com/BSC-RM/slurm_simulator, 2021, accessed: 2021-01-20.

[16] A. Jokanovic, M. D'Amico, and J. Corbalan, "Evaluating slurm simulator with real-machine slurm and vice versa," in *PMBS*, 2018.

[17] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby, "Benchmarks and standards for the evaluation of parallel job schedulers," in *Workshop on Job Scheduling Strategies for Parallel Processing*, 1999.

[18] "The standard workload format," https://www.cs.huji.ac.il/labs/parallel/workload/swf.html, 2021, accessed: 2021-01-20.

[19] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "Legoos: A disseminated, distributed OS for hardware resource disaggregation," in *OSDI*, 2018.

[20] M. Amaral, J. Polo, D. Carrera, N. Gonzalez, C.-C. Yang, A. Morari, B. D'Amora, A. Youssef, and M. Steinder, "Drmaestro: orchestrating disaggregated resources on virtualized data-centers," *Journal of Cloud Computing*, 2021.

[21] 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.

[22] A. De Blanche and T. Lundqvist, "A methodology for estimating co-scheduling slowdowns due to memory bus contention on multicore nodes," in *International conference on parallel and distributed computing and networks*, 2014.

[23] ——, "Addressing characterization methods for memory contention aware co-scheduling," *The Journal of Supercomputing*, 2015.

[24] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Bandwidth bandit: Quantitative characterization of memory contention," in *CGO*, 2013.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO*, 2011.

[26] BSC, "Profet: Code for generating memory bandwidth load, for different read traffic ratios and bandwidth intensity." 2019, accessed: 2019-10-16. [Online]. Available: https://github.com/bsc-mem/PROFET

[27] I. Corporation, "Intel® Xeon® processor E5-2600 product family uncore performance monitoring guide," *tech. rep.*, March 2012.

[28] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of intel sandy bridge and amd bulldozer," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014.

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks: Summary and preliminary results," in *SC*, 1991.

[32] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *ISPASS*, 2016.

[33] D. Buragohain, A. Ghogare, T. Patel, M. Vutukuru, and P. Kulkarni, "Dime: A performance emulator for disaggregated memory architectures," in *Asia-Pacific Workshop on Systems*, 2017.

[34] B. Abali, R. J. Eickemeyer, H. Franke, C.-S. Li, and M. A. Taubenblatt, "Disaggregated and optically interconnected memory: when will it be cost effective?" *arXiv preprint arXiv:1503.01416*, 2015.

[35] W. Cirne and F. Berman, "A comprehensive model of the supercomputer workload," in *IEEE International Workshop on Workload Characterization*, 2001.