Job Scheduling for Disaggregated Memory in High Performance Computing Systems

Felippe Vieira Zacarias

Department of Computer Architecture
Universitat Politècnica de Catalunya

This dissertation is submitted for the degree of
Doctor of Philosophy

Barcelona, July 2023

Job Scheduling for Disaggregated Memory in High Performance Computing Systems

by

Felippe Vieira Zacarias

A Dissertation
Presented to the Department of Computer Architecture
at
Universitat Politècnica de Catalunya
in Candidacy for the Degree of
Doctor of Philosophy.

Thesis Advisors:

Paul Carpenter
Barcelona Supercomputing Center, Spain

Vinícius Petrucci
Micron Technology, USA

Xavier Martorell Bofill, Prof.
Universitat Politècnica de Catalunya, Spain

Barcelona, July 2023

This page is intentionally left blank.

Job Scheduling for Disaggregated Memory in High Performance Computing Systems

This page is intentionally left blank.

Abstract

In a typical High Performance Computing (HPC) cluster system, a node is the elemental component unit of this architecture. Memory and compute resources are tightly coupled in each node and the rigid boundaries between nodes limits compute and memory resource utilization. The problem is increased by the fact that HPC applications have a widely varying per-node memory footprint due to diverse application characteristics, differing problem sizes, and strong scaling. In fact, 25% to 76% of the system's total memory capacity typically remains idle. 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. It enables fine-grained allocation of memory capacity to jobs while maintaining the cost-effectiveness and scalability of a cluster architecture.

A key component for the distribution of computing power within the cluster infrastructure is the Resource and Job Management System (RJMS) or simply resource manager. Its goal is to satisfy users' demands and achieve acceptable performance in the overall system utilization by efficiently matching requests to resources. Even though several researches on RJMS have been carried out to solve problems related to the current state–of–the–art on HPC systems, memory disaggregation is still under development. Therefore, adopting a disaggregated architecture means redesigning the resource manager services. In this thesis we propose an efficient memory disaggregated infrastructure for a cluster resource manager and its evaluation at scale through a structured simulated experimental methodology employing a contention model that models the impact of shared resources in disaggregated scenarios.

Sharing common memory devices or interfaces in a disaggregated infrastructure may incur an unsatisfactory loss of performance because concurrent memory access can saturate the resource; we start our study by introducing a systematic methodology to build a contention model. Extensive real-machine experimentation and the results of workloads have shown that our contention model predicts performance degradation with at most an average error of 1.19 % and max error of 14.6 %. Compared with the state–of–the–art, the relative improvements are almost 24 % on average and 33 % for the worst case.

In sequence, we argue that it is possible to increase throughput and utilization using memory disaggregated in a resource manager. We 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. In addition, on average, it can increase the memory utilization by a factor of 1.6, while having almost 90% of Central Processing Unit (CPU) utilization.

In our study, we also investigate how critical memory demand bounds are for maximising system throughput and minimising job response time. We analyse to what degree the users would have a natural incentive to provide accurate memory bounds. We demonstrate that even when there is a large effect on system throughput (-25%) and response time (5 times higher), there is a very little direct incentive for the users to be accurate in their estimates, with only an 8% increase in response time. We further demonstrate that taking advantage of memory temporal and spatial imbalance among jobs delivers improvements up to 18% in throughput, 38% in throughput per dollar, and up to 69% reduction in job response time (median) when there are imbalanced memory usage and overestimated demands on underprovisioned systems.

Overall, we believe our study provides valuable insights on the importance of design space exploration for disaggregated memory HPC systems. We demonstrate that by understanding disruptive architectural changes on future systems and the demands of the workloads, system provisioning can be carefully designed to achieve the best cost–benefit.

Contents

This page is intentionally left blank.

List of Figures

List of Tables

To,
my family,
and the memory of my grandmother
Nair Zacarias.

acknowledgements

FIRST and foremost, I would like to thank God for the blessing and the gift of life. He has given me strength and encouragement through all the challenging times to reach the conclusion of this dissertation and succeed in this part of my life's journey so far.

The completion of this thesis could not have been possible without the participation and assistance (directly and indirectly) of several people. The biggest thanks go to DR. PAUL M. CARPENTER for all the help, advice, and support. I'm grateful for the opportunity he gave me to complete my doctorate in such a prestigious and amazing institution as the Barcelona Supercomputing Center (BSC), in which I had the opportunity to meet some incredible people. I also thank him for all the learning and guidance in shaping and polishing the papers to produce high-quality results. It was a great pleasure having him as my advisor. I also would like to thank DR. XAVIER MARTORELL who was always available to help me with the bureaucratic part of my Ph.D. studies.

I would like to thank DR. VINÍCIUS PETRUCCI who took me in as a pupil during my master's and introduced me to Paul and the opportunity to reach higher academic standards abroad. He pushed me and encouraged me to finish my master's degree as quickly as possible to work with Paul. Even though he did not participate at the beginning of my Ph.D., he got in on the act. He willingly accepted the invitation to be my co-advisor and since then, he has supported the work with valuable insights which contributed to shaping this thesis as it is today.

I am also thankful to all my friends in Barcelona that were a spark of light and joy during the dark solitude that was living alone in a distant country far away from my family. I especially thank (in alphabetic order) CHENLE YU, GUSSEPE BRAVO, JIE SONG, OMAR SHABAN and PEINI LIU. They certainly made my days in Barcelona even more delightful. Thanks for all chit-chats, coffees, dinners, and trips we shared. I really enjoyed your companionship and I undoubtedly gained a significant amount of knowledge about your culture. Hope someday I will be able to visit you wherever you are.

Me gustaría agradecer a los parceros del “Chiringuito 1k” Gussepe Bravo, Jean Piers y Rick Delgado por las fiestas, pichangas, bromas y chelas que compartimos durante eses años. Por las historias que puedo contar y no tanto por los billetes que dejé en los chiringuitos. Deseo volver a encontrarme con ustedes en futuros veranos.

I would like to thank the pre-defense committee and external reviewers that contributed to shaping this thesis with their valuable and constructive comments. Without their input, this thesis would not have the quality it has. I also would like to thank all my roommates and friends from football groups who shared good times with me and helped me to enjoy the city.

Finalmente, gostaria de agradecer a minha família que mesmo sabendo que estaria longe me encorajaram a perseguir essa oportunidade. Minha gratidão vai especialmente para minha mãe Elisangela, minha madrinha Cristina e meu primo Rafael. Obrigado por surportarem essa distância e sempre fazerem memoráveis as minhas férias. Obrigado pelos fantásticos São João que passei em casa. Saibam que vocês são tudo para mim. Também gostaria de agradecer as pessoas maravilhosas que conheci no São João de 2022 e que fizeram desse um ano muito especial. Em especial a Jessica que espero compartilhar mais brejas e Lane (usando o apelido porque ela não gosta do nome) com quem espero viver junto.

Como ápice desta dedicatória, gostaria de dedicar esse trabalho em memória de minha vó Nair que acompanhou o começo dessa jornada mas infelizmente não está mais aqui para celebrar comigo o fim desta etapa e o começo de muitas outras. A senhora que desde muito cedo foi minha inspiração e o alicerce de nossa família. Sem dúvidas lhe devo tudo que sou hoje. Espero continuar sempre sendo seu motivo de orgulho.

Thanks all!!!

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).

Acronyms

API	Application Programming Interface.
BSC	Barcelona Supercomputing Center.
BSC	Bull Coherent Switch.
CAS	Column Access Strobe.
ccNUMA	Cache Coherent Non Uniform Memory Access.
CPU	Central Processing Unit.
CXL	Compute Express Link.
DDR3	Double Data Rate type three.
DIMM	Dual In-Line Memory Module.
DRAM	Dynamic Random-Access Memory.
ECDF	Empirical Cumulative Distribution Function.
FaaS	Function as a Service.
GCC	GNU Compiler Collection.
GPU	Graphics Processing Unit.
HPC	High Performance Computing.
I/O	Input and Ouput.
IaaS	Infrastructure as a Service.
IMC	Integrated Memory Controller.

IPC Instructions per Cycle.

LANL Los Alamos National Lab.

LDMS Lightweight Distributed Metric Service.

LLC Last Level Cache.

LSF Load Sharing Facility.

MESIF Modified, Exclusive, Shared, Invalid and Forward.

NPB NAS Parallel Benchmarks.

NUMA Non Uniform Memory Access.

OS Operating System.

PBS Portable Batch System.

QoS Quality of Service.

QPI Quick Path Interconnect.

RAM Random-Access Memory.

RDMA Remote Direct Memory Access.

RDP Ramer–Douglas–Peucker.

RJMS Resource and Job Management System.

SDM Software Defined Memory.

SGI Silicon Graphics.

SLO Service Level Objectives.

SST Structural Simulation Toolkit.

SWF Standard Workload Format.

TORQUE Terascale Open-source Resource and QUEue Manager.

VM Virtual Machine.

Part I:

Prologue

Vou pedir licença pra contar a minha história

Rita de Cássia

CHAPTER 1

Introduction

In distributed systems, HPC infrastructures involve a large number of nodes interconnected by a network to solve a wide range of problems. From a higher perspective, the node is a fixed set of computing resources (e.g., processor, memory, storage) connected to a single motherboard that can only be used by the applications running on the given node [1]. These infrastructures tend to be designed with a fixed memory capacity per node based on the most memory-demanding applications that run in the system in order to deal with occasional peaks of data [2–4]. Due to the current node-based constrained architecture, large idle fractions of processing or memory capabilities can not be accessed by other nodes when Input and Ouput (I/O) intensive tasks are running in a given node [5]. Since applications have a wide range of memory demands, from tens or hundreds of megabytes up to gigabytes per core [6, 7], the mismatch between fixed proportionalities and diverse sets of workloads leads to substantially underutilized resources [8, 9].

Furthermore, processor and memory technologies are advancing at a diverging rate in the past two decades. Upgrading components can be a challenge, especially for the memory resource, since these two resources need to be upgraded together [10, 11, 2]. To keep up with emerging technologies and deal with different requirements of applications, the system should have the ability to be flexibly augmented with new memory technologies [2]. As a result, to mitigate scalability issues of node-based systems and efficiently satisfy memory-driven applications, disaggregated approaches have been proposed in order to allow a flexible and finer-grained allocation of resource capacity to compute jobs [2, 12–14].

In the disaggregated design, individual components such as processor, memory, and storage are interconnected over a network [15–19] rather than being restricted to a bus on a single board [20]. These resources exhibit different trends in terms of cost, performance, and power scaling [12]. While storage has been one of the first resources to be disaggregated, memory is much more challenging [21]. However, advances in network speed and scalability with new technologies, are enabling fast access to hardware components that are disaggregated across the network [22]. Such trends are making memory disaggregation feasible and pushing forward research efforts toward its realization. The EuroEXA family of projects (ExaNoDe, ExaNeSt, ECOSCALE) [16, 23], for instance, 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.

Adopting a disaggregated architecture also means redesigning the environment and services that execute on HPC clusters to manage its new features. A key component in this infrastructure is the RJMS, throughout this document also called resource manager. It is responsible for efficiently tracking the cluster's physical resources, submitting and scheduling jobs, managing the queue of pending jobs, and reporting the status to the users. A lot of research around resource managers has been carried out to solve problems related to the current state–of–art on HPC systems. As an example, we have the Slurm [24] resource manager, a popular open-source RJMS that provides a modular and plugin architecture highly configurable for several extensions. However, since disaggregated systems are still under development, all resource managers assume the prevalent node-based architecture which couple together memory and processing resources within node limits. Memory management is still bonded by the node's processing unit availability, which means that nodes without idle cores are excluded from further allocations, even though there is memory capacity available to be used.

Allocation policies are in their infancy for these novel disaggregated memory systems. It is important to investigate them to discover the best algorithm or heuristics to be employed for both performance and efficiency. To facilitate research efforts in such new architecture without practical prototypes, our methodology and validation take advantage of the design presented by the Barcelona Supercomputing Center (BSC)'s Slurm simulator. It is based on the original source code of the Slurm resource manager and its modifications comprise mainly job execution and synchronization. Furthermore, as demonstrated in [25] the simulator has stability through deterministic results from multiple same-input executions and accuracy at the level of real machines, which will allow us to seamlessly simulate different developed strategies using disaggregated memory on several distinct systems.

Nevertheless, in disaggregated memory systems, requests from resource-hungry applications can be executed consuming resources from other nodes, while others can share memory to better exploit capacity and improve performance [26]. As a consequence, sharing common memory devices or interfaces may incur an unsatisfactory loss of performance (called degradation or slowdown) because concurrent memory access requests can saturate the component [22]. Several general approaches to predict the performance degradation had been proposed [27–29], including the Slowdown based method which relates computing demands to applications degradation. They had been successful for single node co-scheduling, but they have not been applied to disaggregated memory.

My[1] thesis focuses on *modeling the impact of sharing resources in disaggregated scenarios* (Part II), *proposing an efficient memory disaggregated infrastructure for a cluster resource manager and its evaluation at-scale*, and *proposing a structured simulated experimental methodology based upon controlled submission of workloads traces* (Part III).

1.1 Challenges and Contributions

In this thesis, our principal interest lies in achieving the same overall performance by reducing the system's total memory capacity. Concretely, we address the following challenges:

Estimating Performance Degradation. The allocation of memory capacity for a disaggregated system to compute nodes is performed in a more flexible way since the resources are interconnected over a network [30]. Applications running on different nodes can share memory devices and interfaces, thus performance can be affected by contention [22]. However, in the disaggregated architecture, cache capacity is not shared among multiple applications, which removes a major contributor to application performance. Prior works have been successful for single node coscheduling [27–29], but they have not been applied to disaggregated memory. For this reason, our analysis is driven by the demand for memory bandwidth, which has been shown to have an important effect on application performance.

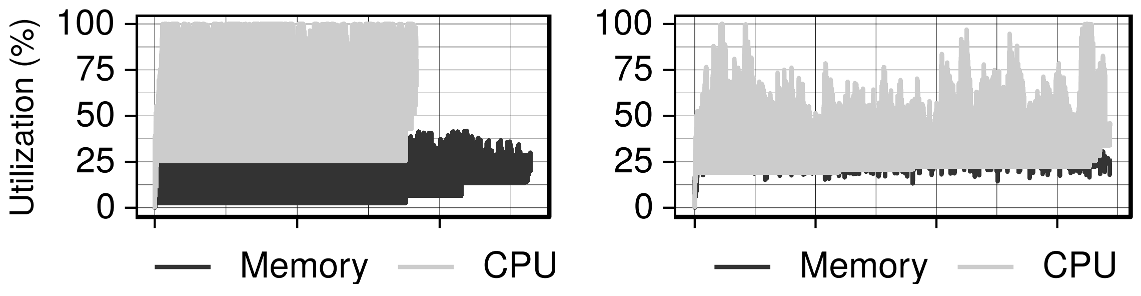
Part II of my thesis focuses on performance predictions in a shared resource contention environment. Before actually implementing and supporting disaggregated features in modern resource managers, we must have a reliable mechanism of assessment for scheduling decisions and job placement. We must also understand the degradation

[1] In what follows, unless stated otherwise, I will use the words: I, My, We, and Our to refer to my exclusive contributions.

caused by resource sharing in disaggregated memory scenarios to guide future proposals and developments. Applications sharing resources may suffer unacceptable amounts of degradation, thus the resource manager must be able to avoid such scheduling and maximize resource utilization while keeping application performance. Estimating performance is a challenge for multiple reasons: ① We must remove the effect of cache resource in our contention model. ② Disaggregated memory prototypes are still at the research level.

In Chapter 4, we introduce a systematic methodology to build a Slowdown based method. We show that profiling the application slowdown often involves significant experimental error and noise, and to this end, we improve the accuracy by linear smoothing of the sensitivity curves. We also show that contention is sensitive to the ratio between read and write memory accesses, and we address this sensitivity by building a family of sensitivity curves according to the read/write ratios. Extensive real-machine experimentation and the results of workloads have shown that the models predict performance degradation with at least an average error of 1.19 % and max error of 14.6 %. Compared with state-of-the-art, the relative improvements are almost 24 % on average and 33 % for the worst case.

Increasing Throughput and Utilization. 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 [10], so they have exclusive access to self-contained nodes, 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 [1] because HPC application memory demands vary dramatically, by orders of magnitude, due to application characteristics and strong scaling [6, 7].



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

Figure 1.1 Resource utilization when the system matches the job's demands and when there is a mismatch.

Figure 1.1 shows an example timeline of total system memory and CPU utilization. In Figure 1.1a, the mix of jobs matches the memory provisioning (system has 25% large capacity nodes and 25% of job CPU hours need large capacity nodes),[2] and average CPU utilization is high, at 81%. Figure 1.1b 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 average less than 48%). The cluster system clearly has abundant unused CPU and memory resources, but they are not available for use by the applications.

In Part III of my thesis, we address the problem of using memory disaggregated in a resource manager and show that it is possible to increase throughput and utilization using remote memory capacity. In Chapter 6 we 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. In addition, on average, it can increase the memory utilization by a factor of 1.6, while having almost 90% of CPU utilization compared to the Baseline.

Simulated Experimental Methodology. Research in job scheduling cannot be easily done using a production system. Optimizing its policies for HPC system performance and user experience is a complex and 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. Furthermore, disaggregated memory prototypes are still at the research level, and system software is immature. In Part III of my thesis we also address the problem by providing a simulated experimental methodology based upon controlled submission of workload traces to evaluate the resource manager's allocation decisions and their implications of considering memory as a disaggregated resource.

We use a simulation approach for two main reasons. Firstly, there are no large-scale HPC systems with disaggregated memory hardware including a complete software stack. Secondly, and more importantly, simulations allow studies to be performed more quickly without occupying the resources of large-scale production systems. We, therefore, extend an existing simulation approach using Slurm resource manager to account for memory bandwidth contention in disaggregated memory leveraging an extension of our contention model (Part II). We then use the extended Slurm simulator to determine the overall system throughput, job queuing, and execution time of distinct large-scale HPC systems.

[2]Details of the experimental evaluation are in Chapters 5 and 6.

Implication of Memory Demands on System Performance. Even though dynamic resource assignment has been explored in [31–34], HPC job schedulers require the definition of the upper bound usage as the initial request to statically assign resources to jobs and to guarantee the necessary resources throughout its complete execution [11, 35]. The request is based on the user's knowledge or experience estimating the resources, whose configuration remains unchanged once allocated to a job [11]. Nevertheless, users likely overestimate their demands to avoid having the job killed [36–38, 2]. Even in a scenario of memory disaggregation, system throughput and response times have large effects when users overestimate their requests.

In addition to this problem, Peng *et al.* [10, 11] demonstrated that jobs in current HPC systems use only a fraction of the memory capacity, with imbalanced memory usage happening across compute nodes and time in a job. To shorten the time to solution, applications with good scalability are distributed over a large number of nodes to accelerate execution, resulting in low memory consumption on a single node [10]. So, the imbalanced usage on a few nodes in a job causes severe underutilization of resources allocated due to the homogeneously configured nodes [11].

In Part III of this thesis, we investigate how critical memory demand bounds are for maximising system throughput and minimising job response time (defined to be waiting time in the queue plus execution time). We analyse to what degree the users would have a natural incentive to provide accurate memory bounds. We further investigate the potential of dynamically managing disaggregated memory. To better understand disruptive architectural changes on future systems, job memory usage details are critical for guiding design space exploration. In Chapter 7 we demonstrate that even when there is a large effect on system throughput (-25%) and response time (5 times higher), there is very little direct incentive for the users to be accurate in their estimates, with only an 8% increase in response time. We further demonstrate in Chapter 8 that taking advantage of memory temporal and spatial imbalance among jobs, delivers improvements up to 18% in throughput, 38% in throughput per dollar, and up to 69% reduction in job response time (median), compared to a static policy, when there are imbalanced memory usage and overestimated demands on underprovisioned systems.

1.2 Outline of Thesis

The most significant contributions of my research, detailed in the previous section, are as follows. (1) A Slowdown based method to predict applications performance on shared disaggregated memory scenarios. (2) An efficient disaggregated infrastructure implemented in a popular RJMS. ③ A structured simulation methodology based on the submission of job traces to study several distinct scenarios and the impact of memory provisioning on the system.

This thesis is organized into five *Parts*, with nine *Chapters* arranged according to the outlined structure. The *Prologue*, which serves as Part I, introduces and provides details about the thesis and is followed by the subsequent chapters:

- Chapter 1 briefly contextualizes the research and introduces the motivations of this work. We also present the challenges faced during the completion of this work and its contributions.

- Chapter 2 describes basic concepts to the understanding of this thesis. We provide a background on RJMS, the concept of resource disaggregation, and different workload datasets that can be employed to simulate the system's performance.

- Chapter 3 discusses the extensive body of related work. We begin listing works related to estimating performance degradation and how we differentiate from them. We further list previous works toward realizing memory disaggregation and we close the Chapter by summarizing the considerations of disaggregated works.

The Part II of this thesis, titled *Modelling Contention-Aware Performance Prediction Technique*, elaborates on our methodology and the results of our study on modeling the effect of shared resources in disaggregated scenarios. This is discussed in the subsequent chapter:

- Chapter 4 describes the performance prediction methodology and the developed contention model for disaggregated memories. It also details the hardware used to run our experiments, the set of single and multi node applications we profiled to create the contention model, and the concept of disaggregated memory employed in this work. Much of the content in this Chapter is presented in our papers [39, 40].

In Part III of this thesis, titled *Allocation and Scheduling In Disaggregated Memory Systems*, we present the proposed disaggregated infrastructure for a cluster resource manager and its evaluation at-scale using a structured simulated experimental methodology. It is extensively detailed in the following chapters:

- In Chapter 5 we introduce the methodology and simulated infrastructure used in this work to evaluate the proposed disaggregated approaches. We also detail the methodology applied to generate the workload employed in all our experiments. The methodology detailed in this Chapter is employed in our papers [40–42].

- Chapter 6 introduces how we extended the resource manager to support disaggregated memory, its evaluation at scale, and how we integrated the developed contention model in our simulated environment. The content of this Chapter can be found in our paper [40].

- In Chapter 7 we investigated how the system's overall throughput and response time would be affected, according to various assumptions on the user's ability to predict the memory consumption. We demonstrate that users should receive incentives to provide accurate memory usage estimates, therefore having less impact on the overall performance. The contributions presented in this Chapter are published in our paper [41].

- Chapter 8 investigates the impact of dynamic memory management and memory demands on system throughput, response time, and cost–benefit. We find that even when users correctly estimate their maximum memory usage, system performance increases up to 12%. We can achieve even higher improvements in performance when the demands are overestimated in underprovisioned systems. Our paper [42] under submission presents much of the content of this Chapter.

Finally, in *Epilogue* (Part IV) which is essentially the Chapter 9, followed by the *Bibliography* (Part V) we summarize the conclusions of all thesis contributions, give future research directions and list the references used during this research.

1.3 Publication List

This thesis is based in part on the following published papers:

① **Felippe Vieira Zacarias**, Rajiv Nishtala, Paul Carpenter, "Contention-aware application performance prediction for disaggregated memory systems", in Proceedings of the 17th ACM International Conference on Computing Frontiers. 2020.

② **Felippe Vieira Zacarias**, Paul Carpenter, and Vinicius Petrucci, "Improving HPC System Throughput and Response Time using Memory Disaggregation", In IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS), 2021.

③ **Felippe Vieira Zacarias**, Paul Carpenter, and Vinicius Petrucci, "Memory Demands in Disaggregated HPC: How Accurate Do We Need to Be?", **Best Paper Award** in the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). IEEE, 2021.

The following publication is under preparation:

① **Felippe Vieira Zacarias**, Paul Carpenter, Vinicius Petrucci, "Dynamic Memory Provisioning on Disaggregated HPC Systems".

1.4 Artifact List

This thesis also provides all materials used during our work in the hope that others can extend and build new insights upon it. The products of this work are available to the public and can be found in:

① Artifact for Contention-aware Application Performance Prediction for Disaggregated Memory Systems. **DOI**: 10.5281/zenodo.5647805

② Artifact for a Simulation approach to evaluate disaggregated memory. **DOI**: 10.5281/zenodo.5806389

③ Artifact for a simulation approach to evaluate the effects of memory demands on disaggregated memory. **DOI**: 10.5281/zenodo.5537135

④ Artifact for dynamic memory provisioning on disaggregated HPC systems. **DOI**: 10.5281/zenodo.7881019

CHAPTER 2

Background

IN this Chapter, we introduce some basic concepts that will provide the bases to understand the problem discussed in this thesis. We define the concept of RJMS and its relevance to HPC systems. We then describe a popular resource manager for HPC systems used in this work, as well as the applied simulation infrastructure based on it. We also give an overview of the novel concept of memory disaggregated solutions and the importance of researching HPC resource management for this upcoming architecture. Finally, we detail the characteristics of the datasets applied to the performance analysis.

2.1 Resource and Job Management System

The increasing number of computational resources and heterogeneity in modern HPC systems has created a diverse and enormous infrastructure that can be used to deal with a wide range of problems. However, exploiting the power of these systems is not an easy task as node architectures have become more complex [43]. The lack of proper software responsible for efficiently allocating the varied available resources may ultimately yield fragmentation, low system utilization, and increased user waiting times. Accordingly, a varied number of RJMS have been developed to manage these large scale compute clusters. The task associated with this middleware is to distribute computing power to users within the parallel infrastructure. It attempts to satisfy users' requests and achieve good system utilization by efficiently assigning it to the resources [44]. User allocation requests, which include various constraints and specifications of resources, will be treated in the system as a scheduling object called *Job*.

The two main concepts around RJMS that capture its basic activities are the *job scheduler* and *resource manager* [45, 46]. Job schedulers will match jobs to resources from several users with different priorities, resource requests, and duration. While the resource manager deals with distinct resources capabilities and availability to well utilize them given the jobs being executed. According to [45, 46], the key functions associated to a RJMS are depicted in the Figure 2.1.

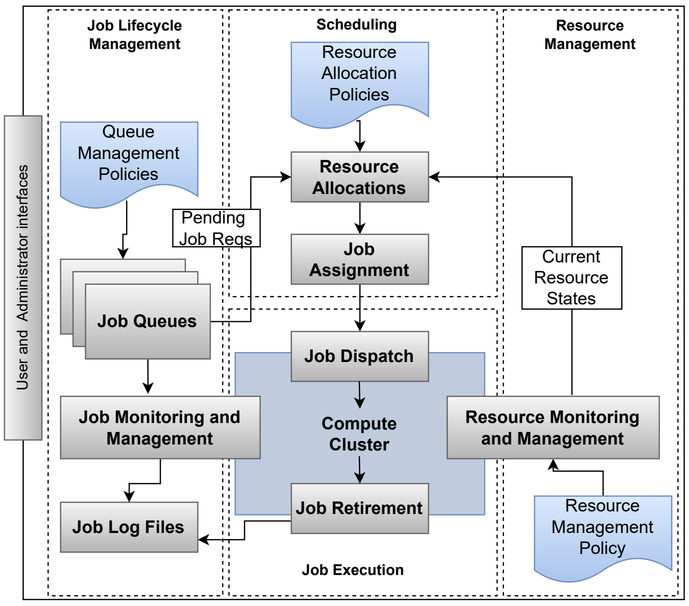


Figure 2.1 Resource and job management system architecture. Figure reproduced from [46, 45].

The typical workflow of a RJMS is as follows: Users send jobs through the job lifecycle management function, which in turn places the jobs in the pending queue to wait for scheduling. The function will also be responsible for applying queue management policies to sort and prioritize pending jobs according to some criteria. The resource management function provides the scheduler with aggregated information from the detectable resources within the system. Further, the scheduler allocates and assigns the resources to run the job. Finally, the job execution function launches the job on the resource and manages closing down and cleaning it upon job completion. Statistics about the jobs will be logged by the job lifecycle function for historical purposes of investigation or administrative use.

There has been significant evolution in the sophistication of resource management and job scheduling as resources and jobs have become more diverse. A number of schedulers have been developed over the years to address various supercomputing and parallel data analysis as well as computer, network, and software architectures [46, 44, 45]. We can list several softwares as Portable Batch System (PBS) [47, 48], Load

Sharing Facility (LSF) [49], LoadLeveler [50], Moab [51]. And open-source alternatives as Slurm [24], Maui Cluster Scheduler [48], Terascale Open-source Resource and QUEue Manager (TORQUE) [52] and OpenLava [53]. Among them, Slurm [24] is one of the most popular [37] and used resource and job management system on current HPC systems.

2.1.1 Slurm Resource Manager

Slurm is an open-source HPC resource and job management system which features a scheduler with a multi-threaded core and a plug-in module architecture [46]. Since it uses a modular architecture, it is highly configurable with a variety of extensions for workload, queueing, scheduling, sharing, etc. Figure 2.2 illustrates the main components of Slurm's architecture.

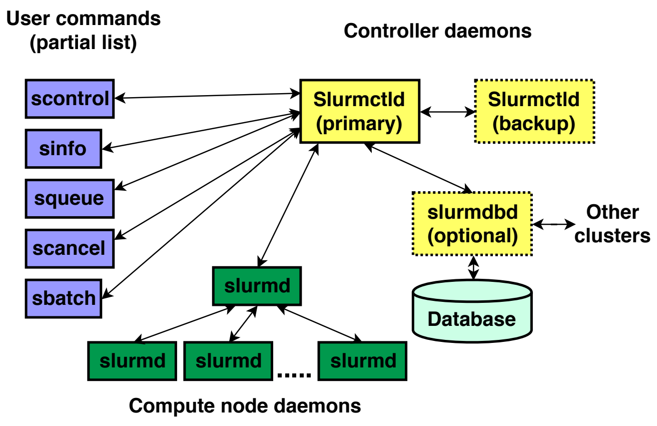


Figure 2.2 Slurm's architecture. Figure reproduced from [54].

Slurm uses a centralized manager (or controller), *slurmctld*, which is responsible for scheduling, allocating resources to jobs, monitoring job execution, and mediating 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, manage job execution on the node and return the completion status. Users interact with Slurm through a set of command-line tools to submit jobs, terminate queued or running jobs, and request system information and job status. The setup can also have an optional Slurm database daemon, called *slurmdbd*, responsible for recording the cluster account information in a single database. System administrators can use available administrative tools to monitor and/or modify configuration and state information on the cluster.

The default node allocation of Slurm is the exclusive also known as server-based or node-based allocation mode. In this allocation mode a job is placed only if a node can satisfy the amount of requested core and memory resources [55], so even if not all resources within the node are utilized by a specific job, no other job is allowed to share the resource. As a matter of fact, many HPC systems disallow the colocation of different workloads on the same compute node to minimize the negative impact caused by inter-workload interference [56, 10].

To deal with the inefficiency of the exclusive mode, Slurm allows fine-grained cluster management by tracking core and memory resource usage. However, in environments where memory is a resource of interest, the user must specify the maximum amount of real memory per node or per core. Due to its node-based approach, requests for more memory per node than is available cannot execute, because Slurm does not allow jobs to use more than the physical memory capacity. Furthermore, memory and processing unit are tightly coupled, which means that if there is a memory resource available but no cores, the resource will not be used in the scheduling process. To deal with this problem, users tend to explore the job scalability, distributing it over a large number of nodes, therefore decreasing the memory required per node.

The allocation and scheduling of a job works as follows, after the submission of a job, the resource manager executes one preliminary analysis of the request through the same process applied in the real scheduling. However, instead of launching the job, it only asserts whether the job can run in that particular architecture given its demands. This quick evaluation promptly raises possible errors for the user preventing the execution of the job. Figure 2.3 presents a simplified diagram of the sequential interactions that take place during the job scheduling process.

Jobs ready to scheduling are selected from the global queue of pending jobs. Then, all available nodes that meet the job's demands in terms of resources (ex. cores and memory) are selected to further evaluation. At this initial point, memory is only a constraint if the requested memory is higher than the node's physical memory. Once the list of potential nodes is completed, the *Selection plugin* decides the nodes that best satisfy the request. This stage consists of removing nodes that at this moment don't have enough idle processing or memory resources to support the job and later, selecting nodes based on a specified policy. The final step modifies the system state information to allocate the resources and initiate the batch job. When the batch job finishes, allocated resources are freed and become available to be used by other pending jobs.

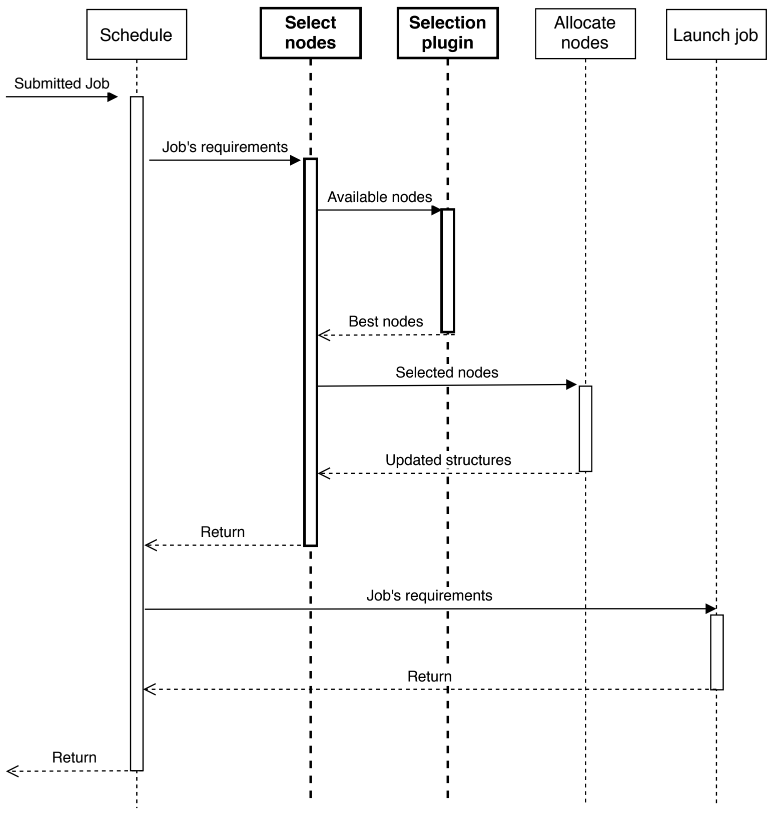


Figure 2.3 Simplified sequence diagram for Slurm scheduling process.

2.1.2 Slurm Simulator

Optimizing the job scheduler and its policies for HPC system performance and user experience is a complex and 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 BSC Slurm simulator proposed by [57, 25] enables a precise and deterministic evaluation of the job scheduler by running it in a simulation environment (it can be found in [57]). It is based on the original Slurm source code so, unlike theoretical models, it is able to capture all parameters and behavior that occur in a real environment. This feature gives the simulator the possibility of applying Slurm parameters used in real environments to have a more precise evaluation of its executions.

The Slurm simulator uses the standard *slurmctld* controller and a simplified *slurmd*, which emulates job execution on a cluster node. In addition, the Slurm simulator manager (*sim_mgr*) reads the input trace, submits jobs to the controller at the correct submission time, and manages the overall simulation. Figure 2.4 depicts the simulator's architecture and its main components. It also details in orange the modules included or modified in the Slurm structure, while the yellow boxes represent modules with little or no modifications.

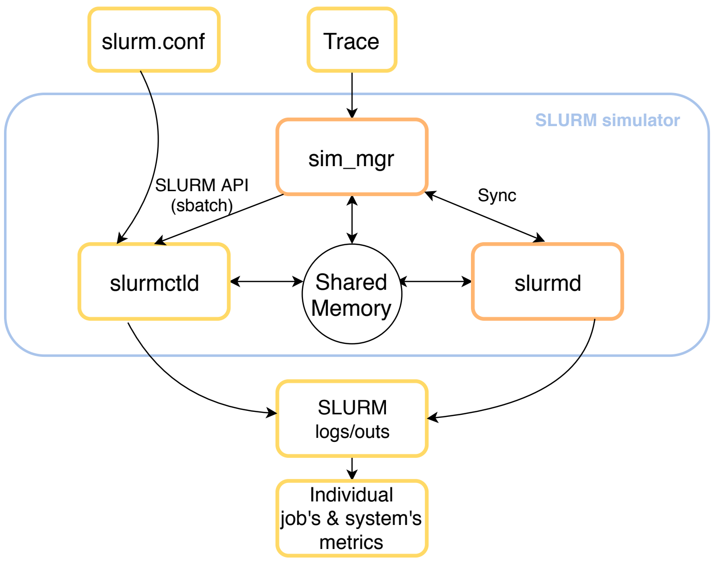


Figure 2.4 Slurm simulator's architecture. Figure reproduced from [25].

The components coordinate the execution through a multi-semaphore synchronization approach depicted in Figure 2.5. *Slurmd* unlocks the first semaphore after finishing all message exchanges with the controller. It enables the simulator manager to add one second to the simulation and process all the events of that second (ex. submitting jobs to the controller). Then, the second semaphore is incremented by the simulator manager to unlock *slurmd* communication with the controller. During its execution, the simulator generates standard output logs for job completion and daemon execution.

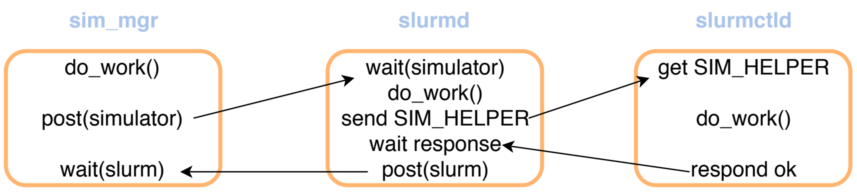


Figure 2.5 Slurm simulator's synchronization process. Figure reproduced from [25].

The simulator receives as input a standard Slurm configuration file (*slurm.conf*) 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) [58, 59] (see details in Section 2.3.1), 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 online repositories as in [60].

2.2 Disaggregated Memory

Although Cache Coherent Non Uniform Memory Access (ccNUMA) machines are available with hundreds of sockets, for example, Silicon Graphics (SGI) Altix [61] and Bull Coherent Switch (BSC) [62], it is difficult to scale cache coherent systems to thousands of nodes. Modern HPC systems are therefore typically built from thousands of ccNUMA nodes communicating via a fast interconnect such as InfiniBand or OmniPath. However, they often suffer from memory underutilization [55].

According to [2, 10, 55], the fundamental reason is the current node-based memory allocation. Computing and memory resources are tightly coupled in each node, resources are requested in units of nodes, and job memory allocation is based on peak usage. Furthermore, HPC systems provide little flexibility in provisioning memory, because Dual In-Line Memory Module (DIMM) 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 [6].

However, memory utilization varies widely from one job to another over time across the nodes, which results in underutilization when there are jobs with small memory footprint [10, 55]. Additionally, many jobs on HPC systems also overestimate their memory demands, thus underutilizing memory slots dedicated to the specific jobs [2]. Consequently, HPC systems suffer from stranded resources because memory that is not used by one job cannot be used by another on a different node.

Enabled by the advances in network technologies, several approaches have been proposed towards a general-purpose architecture to disaggregate resources, as an alternative to the monolithic node-based approach. By providing a fine-grained allocation of resources, they aim to solve the problem of imbalance in memory usage and expand memory capacity by exposing the global memory to all machines. This scenario is boosted due to several factors which we can mainly cite the increase of in-memory data processing, instead of providing large memory nodes with lower efficiency regarding cost and power, the memory disaggregation allows more memory than locally supported by a node, and the overestimation when users allocate worst-case scenarios to request for computational resources.

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 [63, 16, 18]. Figure 2.6 presents an example of a disaggregated design. In this design, every resource is managed as a pool, and a node is assembled based on the user's request. The EUROSERVER [16], ExaNoDe [23] and EuroEXA [64] 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 RDMA or direct load–store instructions. By appropriately configuring the cache policy, remote memory accesses can be cached locally. The dReDBox project [63] built a low-power architecture with an optical network connecting hot-pluggable modules that provide compute, memory, and accelerator resources. Its software-defined global memory and peripheral resource management offer fine-grained resource allocation on-demand to improve utilization.

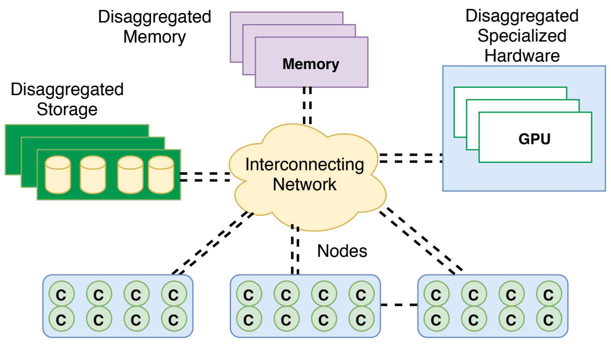


Figure 2.6 Example of disaggregated architecture. Figure reproduced from [2].

In this context of continuous efforts from the academia and industry to realize memory disaggregation with low monetary cost, high performance, and low latency, the new concept of the Compute Express Link (CXL) [65–67] emerged. It is an industry-supported cache-coherent interconnect for processors, memory expansion, and accelerators. Designed to be an open standard interface for high-speed communication, CXL maintains memory coherency between CPU memory and the attached devices [66, 67]. Its reduced software stack complexity yields low-performance impact, as the direct-connected memory can be accessed with roughly the same latency as a Non Uniform Memory Access (NUMA) hop [68, 69]. Therefore, CXL provides a practical basis to implement memory disaggregation in terms of low latency interconnect protocol [68]. However, it has not been made for production yet [30].

In our work we use the disaggregated architecture model shown in Figure 2.7. It is inspired by recent disaggregated memory models and the UNIMEM approach [16], which implements a global address space and allows its access either using ordinary load–store instructions or RDMA. Similar to these designs, our architectural model features computing units that 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 design supports caching locally at the unit that requested access or remotely at the unit attached to the memory.

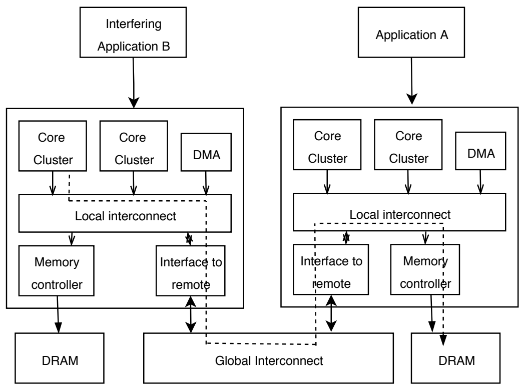


Figure 2.7 Model architecture for memory disaggregation. Remote memory accesses are routed to the appropriate node through a global interconnect.

2.3 Workload and Job Traces

According to [70, 71, 58, 72] the study and design of computer systems requires good models of the workload because it has a large effect on the observed performance. Therefore, realistic workloads are crucial to determine performance in practice. The need for realistic workloads is important in evaluating supercomputers because they are very expensive, therefore it is rarely an option to conduct extensive experiments in production [70].

Most systems maintain accounting logs for administrative use, describing valuable information about all the activity on the machine, and also about the attributes of each job that was executed [58]. However, there is no standard for parallel schedulers, and each one defines its own log format [73]. On the other hand, workload models are based on some statistical analysis of workload logs, in order to expose their underlying principles. They enable the creation of new workloads that are statistically similar to the observations but can also be changed at will [58].

2.3.1 Standard Workload Format

To help the researchers use real or synthetic workloads and ease the use of workload logs and models, Chapin *et al.* [58] defined the concept of the SWF. Applying this standard, programs that analyze workloads or simulate systems only need to parse a single format applicable to several workloads [59]. A major breakthrough in this design is the possibility of using the format for both real and synthetic workloads [58].

The SWF follows some principles, it stores each workload in a single file with one line per job, has space-separated fields, and has exclusive use of numerical values [73]. The fields were chosen so that all information from logs would be saved except very rare fields [58]. Irrelevant or missing fields for a specific log or model appear with a value of -1, and all other values are non-negative. The standard allows comments starting with the semicolon [59], which are ignored during the parse. Usually, at the beginning of the file, there are several such lines describing the workload and the environment from which the traces were collected. It also uses the line number as the unique job identifier. Identifiers for jobs from workloads converted to the standard format are discarded since they are not always integer or unique [58].

The standard was established when the main concerns were arrival time and basic resources such as processors and runtime. Feilteson *et al.* [73] pointed out that researches suggest it might be important to follow the dynamics of resource usage. To use such data, the standard workload would need to be augmented with additional data that includes multiple records of the same job. Both the original data about the jobs and the additional file including the dynamic records would be associated based on the job identifier.

2.3.2 CIRNE Model

The CIRNE Comprehensive Model [70] provides a model for supercomputer workload to tackle problems found with previous workload models, more specifically modeling request time and the possibility of cancellation. In order to do that, it uses four logs from different distributed memory machines. It models the arrival pattern, execution time, requested time, status, and job sizes of those logs to generate similar workload logs. To better capture the dynamics of the system, the model takes into account the seasonal working day cycle to fit the arrival instant, as typically more jobs are submitted during the day than at night. Another insight captured by Cirne *et al.* [70] when modeling the logs is that there is a strong correlation between short execution time and poor request accuracy, as failed jobs usually terminate sooner and therefore having in general poorer accuracy than completed jobs. The model then derives the execution time from the explicitly modeled accuracy and requested time to complete the reproduction of synthetic workload traces.

2.3.3 Google Trace

To support the research on scheduling for large-scale compute clusters, in 2020 Google released detailed scheduling information from 8 different Google compute clusters (*cells*) [74]. Each cell is a collection of machines that operate as a single management unit. The traces describe several days of workload on a single cell and it is separated into several tables, which store information provided by the management system and the individual machines. The trace has data from two kinds of resource requests the cluster scheduler receives, a *job* which describes resources needed and computations a user wants to run, and an *alloc set*, which describes the resources reservation the job can run. Each job may run several tasks, which inherit several properties from the job, e.g. priority, resources request [74].

Jobs may be classified according to their assigned priority, which defines how they will be scheduled. Jobs with the lowest priorities usually incur low or no internal charges, and have weak or no Service Level Objectives (SLO). These jobs can be evicted to ensure that higher priority jobs receive their expected level of service, as jobs in this category require high availability [74]. All jobs also have a *scheduling class* property that roughly represents how latency-sensitive the job is. Higher values represent more latency-sensitive tasks (e.g. user-facing service jobs) and lower values represent non-production tasks or often batch jobs (e.g. development, non-business critical analyses, etc.) [75].

The trace does not have the exact hardware specification of each type of node, as some attributes have been obfuscated for confidentiality reasons, especially resource requests and utilization measurements. Memory sizes are normalized and scaled by dividing their values by the maximum machine memory size observed across all traces [75]. The usage values are reported from a series of non-overlapping measurements, typically 5 min long. During each measurement period, the data is sampled once per second and the memory usage collected every sampling period is aggregated into average and maximum usage over the time window [75].

2.3.4 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 [56, 76]. In this thesis, we use the dataset for the largest system, Grizzly [77], a mid-range TOP 500 supercomputer with 1490 nodes, each with 128 GB DRAM. The complete Grizzly trace consists of 53.4 GB of (uncompressed) data, which comprises over $70,000$ jobs and 560 million records.

Table 2.1 % of maximum memory usage per node for all jobs in Grizzly trace. (Small: ≤ 32 nodes; Large: > 32 nodes)

Max memory use		Usage	
(GB/node)	All	Small	Large
(0,12)	73.29%	63.5%	77.77%
[12,24)	12.43%	20.24%	8.86%
[24,48)	8.17%	8.45%	8.04%
[48,96)	5.65%	6.98%	5.04%
[96,128)	0.46%	0.83%	0.29%

The memory usage for this dataset is collected using the Lightweight Distributed Metric Service (LDMS) [78], a framework for collecting metrics data from computational systems without a significant negative impact on the application's performance running on the system. The data is a periodic snapshot, relatively fine-grained (once every ten seconds) of the free and active memory statistics for every node. It identifies which job used that memory. Since the job identifications are unique, it is possible to identify the parallel jobs running between multiple nodes and therefore deduce the job's number of nodes and duration [56].

Table 2.1 presents the Grizzly trace memory distribution considering the maximum memory used by every job in any node. Even though the system utilization is reported to be *78%* [56], it is clear from its distribution that memory wise the system is underutilized and provisioned to run the worst cases. We can notice that the majority of jobs use less than 24 GB. According to [56], 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. We can also see similar numbers in Peng *et al.* [10] whose results show that for a different HPC system, more than *90%* of jobs utilize less than *15%* of the node memory capacity, and for *90%* of the time, memory utilization is less than *35%*.

CHAPTER 3

Related Work

In this Chapter, we discuss the work related to this dissertation. We begin the Chapter by discussing the topic of estimating performance degradation in Section 3.1. Given the importance of the disaggregated approach and all the challenges in its development, it is also important to pay special attention to the problem of characterizing the performance degradation of an application when sharing resources in this architecture. The rest of the section will present prior efforts to identify, quantify or model the applications' performance due to shared resource contention.

Section 3.2 presents previous works toward realizing memory disaggregation. Some of the listed works propose page based solutions that are either implemented on the kernel or as an extension of the hypervisor memory management. Section 3.3 presents some works that address utilization and the dynamic characteristics of resource usage. We also discuss the topic of accessing and pricing schemes for disaggregated systems, since the current models (often based on core-hours) should be adapted to take into account the disaggregated resources in the newly developed architectures. And finally, Section 3.4 summarizes the considerations of disaggregated works and the analyzes carried out in this thesis.

3.1 Performance Prediction Modelling

Slowdown Based Methods — De Blanche *et al.* [28, 79] propose a slowdown based characterization method to estimate the applications' slowdown when sharing the memory bus. Their sensitivity curve is obtained using synthesized memory traffic and the contentiousness is found using performance counters. De Blanche *et al.* use a fixed read or write ratio to create the sensitivity curve, while, for better accuracy, we calculate the performance degradation using the sensitivity curve that best represents the interfering application.

Bubble-Up proposes a generalisable methodology for predicting performance degradation from contention for shared resources in the memory subsystem [27]. For Bubble-Up, sensitivity, and contentiousness quantify occupancy of the Last Level Cache (LLC), but it is pointed out that the methodology can be applied to other metrics. Since Bubble-Up only considers cache occupancy, it cannot be applied in its original form to our problem, which has separate caches. The work is further improved in [80] to dynamically measure the application's sensitivity for latency-sensitive applications.

Bandwidth Bandit [29] 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. For performance prediction it uses a bandwidth graph, which is a quantitative description of the application's sensitivity to bandwidth contention, then it finds the throughput of a given number of co-running instances. Rather than stealing cache capacity as it is done in the author's previous work [81] (see below), Bandwidth Bandit executes the applications in the same socket using n instances of a single-threaded application.

Cache Contention — Several works have been proposed to model how applications' performance is affected by changes in the amount of available shared resources, especially cache capacity. Our work does not use cache interference, as it is misleading when coscheduling applications using separate cache hierarchies [28]. Eklov *et al.* [81] propose a methodology to measure application performance and bandwidth as a function of the cache capacity occupied by an interfering application. Zhao *et al.* [82, 83] capture the aggregate cache and bandwidth utilization of all cores and characterize the performance degradation using a regression approach, which admits different sub-functions depending on which contention factors are dominant.

Casas *et al.* [84] present a methodology that quantifies an application's utilization of the memory hierarchy, specifically, the capacity and bandwidth of shared caches to predict the application's performance when the required memory resources are not available. Casas *et al.* qualitatively demonstrate that their validation methodology has higher accuracy than prior work [81, 29].

Online Mechanisms — The following works provide online methods to estimate the performance of applications already running in contention and using simulation. Subramanian *et al.* [85, 86] and Xiong *et al.* [87] use, respectively, the memory request rate, cache access rate, and Instructions per Cycle (IPC) to estimate the slowdown for individual applications. Their method involves giving high priority to the target application for a short time to estimate the value of its respective metric if the application were running alone, and then using this value to calculate the experienced interference. Barve *et al.* [88] present an open-source framework that covers a wide range of application classes to guide providers in building performance models. The framework can be used to predict interference levels and make effective resource management decisions. In its offline phase, it defines and clusters a collection of resource utilization metrics and specifies a resource stressor prediction model. The model is applied in the online phase to stress a target application across different resource utilization regions to train a model, later this model is used to predict the application performance at runtime.

Interference-Related Benchmarks — The subsequent works provide a set of benchmarking tools to identify or create contention in shared resources rather than predicting the performance of applications under contention. Delimitrou *et al.* [89] present a collection of microbenchmarks to apply contention or to identify shared resources an application creates contention to, and similarly, the type and amount of contention the application is sensitive to. Xu *et al.* [90] propose a profiler based on supervised machine learning to identify bandwidth contention in NUMA architectures. Molka *et al.* [91] use multiple micro benchmarks to stress different resources in the memory system to identify hardware performance counters that can be used to detect problems caused by memory access.

In contrast to the aforementioned works, our modeling approach provides improvements for a singular reason: Our approach targets performance prediction due to sharing of disaggregated memory, while the prior works use application working set size or local bandwidth as their measure of pressure to create the sensitivity curve. As noted in this thesis, the cache contention characterization method is misleading for predicting the performance of applications using separated cache hierarchies. For this reason, we proposed using a family of smoothed sensitivity curves to account for varying ratios between read and write memory accesses to increase accuracy and decrease the effect of outliers. Our results in Chapter 4 show that our Slowdown methodology is a good approximation for predicting the performance of applications under remote memory access interference delivering higher prediction accuracy than the state–of–the–art with an average error of 1.19% and max error of 14.6%.

3.2 Disaggregated Solutions

Since disaggregated architectures are being developed and full scale prototypes are not ready yet, researches have been mainly focused on the requirements needed to support memory disaggregation [92]. Moreover, despite the promising benefits of resource disaggregation it is still unclear how to properly manage it [22], especially at a large scale.

Memory Disaggregation Systems — Gu *et al.* [14] 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. Its architecture comprises a block device and a daemon that executes in every machine without central coordination. The block device exposes an I/O interface to the virtual memory manager which treats the address space as a fixed size partition, while the daemon runs in user space and responds to memory requests. Their solution is implemented as a loadable kernel for Linux using RDMA for communications.

Jo *et al.* [55] present better performance than [14] by introducing a novel RDMA-backed caching layer for memory disaggregation. They develop a custom paging module that automatically adapts to the memory access patterns of individual processes to minimize the inherent overhead of remote memory accesses. Yoon *et al.* [93] develop a paging based memory disaggregation system on top of unikernels. It provides fast remote memory access improving page fault latency by overlapping the page fault handling with asynchronous network requests and leveraging unikernels features to remove inter-process security checks and locks latency. Their solution achieves up to 2.2× higher performance in real-world workload compared to the paging based system presented in [94].

Lim *et al.* [26, 18] propose a remote memory blade that can be used for memory capacity expansion to improve performance and for sharing memory across servers. They extended the Xen hypervisor to emulate a disaggregated memory design where remote pages are swapped into local memory. Shan *et al.* [22] rely on the concept of splitkernels architecture to break the OS functionalities into loosely-coupled monitors to manage the hardware components. They use a two-level resource management mechanism to support their disaggregated architecture. The global managers perform coarse-grained global resource allocation and load balancing, and they can run on one normal Linux machine, while each monitor works at the lower level, manages a hardware component, virtualizes and protects its physical resources. They emulate the disaggregated hardware components using commodity servers and assert the advantages of disaggregation in resource packing, failure isolation, and elasticity.

Peng *et al.* [10] 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. They studied performance characteristics such as access patterns, local/remote memory ratios, and network connectivity.

Cao *et al.* [95] present a shared-memory based memory paging service to improve Virtual Machine (VM) swapping system. Their solution creates a compressed shared memory swap area between host and VMs, intercepts and redirects the swapping traffic to this area. They leverage the idle memory present in the host or other VMs on the host to implement the proposed disaggregated memory system. Buragohain *et al.* [21] present a performance emulator for disaggregated memory architectures. It works by injecting delays to protected portions of the virtual address space of the process under emulation that correspond to the remote disaggregated memory. The user can specify the amount of local and remote memory, the interconnect bandwidth, and the remote access latency, then the delay will be calculated. Their work showed good accuracy and low overhead in remote memory emulation.

Hardware-level Disaggregation — Pinto *et al.* [31] present *ThymesisFlow*, a fully-functional software-defined disaggregated memory prototype using commercially available hardware components. The architecture introduces the concepts of a borrower, which uses the remote memory, and a lender that donates it. Its design leverages the latest cache-coherent attachment technology for off-chip peripherals to intercept memory transactions and realize the endpoint functionality. They analyze the impact of disaggregated memory by measuring the performance of cloud applications and demonstrating acceptable performance.

The system presented in Guo *et al.* [96] improve some aspects presented in [22] by proposing a new co-designed hardware-software for memory disaggregation. Its architecture includes computing nodes using a user-space library, which is in charge of handling request ordering, retry, congestion, and incast control of the application's memory requests. It also contemplates a memory node device that runs the logic for all data accesses and handles the metadata and control operations. Bielski *et al.* [63] propose for the orchestration of its introduced disaggregated architecture a software component called Software Defined Memory (SDM) Controller integrated into the OpenStack framework. It interacts with agents running on the OS of the memory, compute, and accelerator modules. Its task is to receive allocation requests from the modified OpenStack compute service scheduler, then select and reserve resources with power awareness. After the selection, a subsequent component called synthesizer will forward all necessary configurations to all involved devices. Currently, the authors are evaluating their architectural proposal through simulations. Another approach is UNIMEM [97], which was developed by the EUROSERVER [16], ExaNoDe [23] and EuroEXA [64] projects. UNIMEM implements a global physical address space for Arm architecture accessible by ordinary load–store instructions and RDMA.

Considering Scheduling — In a disaggregated architecture, the coordination of all the hardware and software falls under the control of the management software [1]. It is a key component for the distribution of computing power within cluster infrastructures. Its goal is to satisfy users' demands for computation and achieve a good performance in the overall system utilization by efficiently matching requests to resources.

Papaioannou *et al.* [1] propose a resource scheduling and network management algorithm designed for a disaggregated data center. The scheduler allows the administrator to define policies, which are enforced through a set of weights. Then taking into account the weights, the work assumes that each VM request comes with a set of requirements (cores, ram size, and network bandwidth) that need to be connected, so it does an iteration of filtering, prioritizing and sorting the selected computing and network resources to take the best decision. The proposed scheduling is evaluated using simulation, which demonstrated the scheduling algorithm can improve resource utilization compared to state-of-the-art algorithms and thus reduce energy consumption.

Zervas *et al.* [8] propose a set of algorithms to allocate and maximize resource utilization for a disaggregated data center based on the dRedBox architecture. The algorithms are first fit, best fit, and two network locality based algorithms. To evaluate the proposal they developed a simulator that performs orchestration and allocation of resources with reservation of their network bandwidth and interconnection to serve VM requests. Their simulated results show the importance of network aware algorithms and resource locality between compute and memory in terms of bandwidth and latency to maximize resource utilization.

Farias *et al.* [92] use traces of a representative production system to simulate a scheduling considering disaggregated architectures. In this work, they focus on investigating the efficiency gains when the scheduler can create new logical servers or increase the capacity of those existing, concurrently with the scheduling process. Their results show that using a disaggregated approach the fragmentation decreases compared to the server-based architecture, as more important jobs are allocated simultaneously, and power management features may switch off the unused resources to yield substantial energy savings.

Considering HPC — Although the data center is a dominant target for disaggregation, we can cite a few works towards HPC applications and environments. Allcock *et al.* [98] present an analysis of a system architecture using a dynamically allocatable Random-Access Memory (RAM) pool over the network. The system consists of an external memory appliance connected to the clusters' infiniband switch, as well as a set of software interfaces to access them through its kernel driver. The authors present four different use cases, including in situ analysis, machine learning, quantum chemistry simulations, and virtualization to evaluate the performance of the applications using this model. They demonstrate that there are applications that take advantage of this model with an acceptable performance impact.

Uta *et al.* [99] implement an in-memory distributed file system to manipulate unused memory and bandwidth of cluster nodes running other applications. In their system, a set of nodes can be reserved to have its memory capacity augmented using remote memory. The user can register their nodes as remote memory nodes to a secondary queue along with the amount of available memory, or the system administrator can enforce all nodes to be registered as remote nodes. They use scientific workflows, HPC, and big data applications to evaluate their design, which presents overhead below 10%.

Kommareddy *et al.* [2] use a simulation model to investigate allocation policies for disaggregated memory architectures using non-volatile memory. They implemented a centralized memory management entity on top of Structural Simulation Toolkit (SST) simulator and evaluate four memory allocation policies designs. In its architecture, the memory manager and external memory are kept remotely and each node is connected to them through external links. Since their focus is for HPC systems, their designs are validated using HPC applications and calculating the performance in terms of IPC.

Kwon *et al.* [100] propose a memory centric disaggregated system architecture to execute deep learning algorithms. In their system, the memory nodes are interconnected to the accelerators through a high-bandwidth and low-latency signaling link within the accelerator and serve as a transparent memory capacity expansion. They present three systems interconnect design points that integrate the memory nodes and discuss their trade-offs in terms of bandwidth utilization and overall performance. Through simulations, they show better results than a conventional approach and also increasing in system-wide memory capacity.

Considering Cloud Computing — In order to increase resource utilization, energy, and operational cost efficiency in data centers, few works address disaggregated resources by modifying their virtualized environment or using advanced implementations. Zhang *et al.* [101] investigate opportunities to improve memory efficiency on virtualized systems. They developed a shared memory based system solution that utilizes host idle memory to improve memory efficiency and VM execution performance for memory intensive applications. To achieve this objective they implemented an optimization layer between the host kernel and VM. Its architecture has two components, a coordinator responsible for establishing the shared memory channel between the host and VMs, and a module responsible for providing dynamic allocation and deallocation of shared memory based on the workload demands. They also improve their memory management by developing a shared memory based swap facility and a shared pipes mechanism to inter-VM communication.

Koh *et al.* [102] propose a hypervisor that transparently provides scalable memory for VMs, without requiring modification of the applications or guest OS. Their disaggregated system is backed by RDMA-supported high bandwidth networks, in which multiple connected nodes donate their free memory to VMs. It is possible through a kernel module linked to the hypervisor that runs on the node requiring memory extension. Also, a donor application runs in each donor node to grant their memory through the network. As the disaggregated support is directly integrated into the page management in the hypervisor, the module handle page faults that occur during the execution of VM context to provide remote memory access. Their results show that 6% of performance degradation on average compared to the ideal large memory node can be supported using disaggregated memory.

Garrido *et al.* [103] extend the hypervisor to share the memory capacity of the nodes across the computing infrastructure. It leverages the hypervisor Transcendent memory mechanism that pools the idle or unassigned physical pages of nodes through a key-value store abstraction. They implement a memory manager that controls the system and distributes the global memory capacity. It enforces constraints on memory allocation on the hypervisor as well as the management of physical pages. To evaluate the solution, the shared global address space was emulated using the node's local memory and delays to remote access.

Chen *et al.* [104] investigate the performance of using an in-memory big data processing system on disaggregated memory. Their system is based on an in-memory distributed file system to accommodate the big data processing while increasing the memory capacity of their data processing cluster with a large volume of DIMM-based persistent memory. Their solution consists of a client and a storage backend. The client integrates with the user application through a set of interfaces to perform data and metadata operations on storage. The storage backend is a remote cluster of persistent memory servers. Empirical results show 3.5× improvements compared to the default big data framework setup.

3.3 Dynamic Memory Provisioning

Resource allocation/usage — Amaro *et al.* [94] examine the scenarios in which remote memory can increase job throughput by presenting a swapping mechanism that uses remote memory through RDMA. Furthermore, they develop a remote memory-aware cluster scheduler to split each job's memory demand between local and remote memory, therefore increasing the number of jobs running simultaneously. Amaral *et al.* [105] develop a dynamic loop-based controller to manage resources and a flow-network algorithm to determine the optimal placement of workloads on virtualized data centers. Their approach uses a middleware that intercepts Graphics Processing Unit (GPU) calls and offloads Application Programming Interface (API) related data via the network. Li *et al.* [68] propose a full-stack memory disaggregation design using a CXL-based approach. The design proposed is composed of a multi-ported external memory controller directly connected to CPU sockets via CXL. A system software layer that exposes the pool of memory as a zero-core virtual NUMA node. And a control plane that relies on predictions of memory latency sensitivity and usage for scheduling and asynchronous management of the pooled memory. Their simulation show that the memory capacity of a cloud system can be reduced up to 10%. Michelogiannakis *et al.* [35] perform a detailed analysis of sampled metrics in a production HPC system to quantify what level of disaggregation is appropriated for HPC workloads. They demonstrate that key resources are consistently underutilized, therefore a resource reduction would satisfy the worst-case average rack utilization.

Dynamic resource assignment — Koutsovasilis *et al.* [32] integrate disaggregated memory into the Linux NUMA memory policy. The developed memory balancing policy autonomously migrates memory pages across local memory to the disaggregated, therefore increasing the performance compared to the swap system. In addition, they introduce a memory orchestration stack that monitors the state of each node and scales the amount of the attached disaggregated memory according to the current memory usage of the node. In spite of dealing with disaggregated memory and integrating it into the Linux memory policies. D'Amico *et al.* [33] present a dynamic job scheduling policy integrated into the Slurm resource manager. They implement a variant of the backfill algorithm to leverage minimizing the system slowdown and co-scheduling malleable jobs with jobs that will execute with a reduced set of resources. Using their policy they show a considerable decrease in makespan, response time, slowdown, and energy consumption. Iserte *et al.* [34, 106] also provide an approach to dynamically reconfigure the size of a job to improve the system's throughput and resource utilization. It enhances the collaboration between OmpSs runtime and the Slurm resource manager by designing an API to instruct the runtime to communicate with the resource manager in order to determine the resizing action to execute. Doudali *et al.* [107] introduce *Kleio*, an approach for memory page scheduling in hybrid memory systems. Their approach leverages machine learning to predict future memory access patterns and dynamically adapt the page placement policy. It demonstrates significant improvements in memory access compared to existing solutions.

Accessing/pricing schemes — Cloud computing operates with a different set of assumptions, compared to HPC facilities, regarding pricing schemes [108]. Access to large-scale HPC infrastructures usually requires submission of proposals to undergo a peer-review process describing computational resources as in [109–111]. Their operators usually charge resource usage based on core-hours [112, 108], cloud service providers apply an on-demand resource provisioning around the concept of reducing investment and usually fixed billing models [113–117].

Mazrekaj *et al.* [113] present an overview of some basic concepts for pricing schemes and models which varies depending on the cloud service provider. Lu *et al.* [117] also present a summary of pricing mechanisms and divide them into auction and non-auction strategies. Then, they propose an auction approach to efficiently allocate resources according to the user's Quality of Service (QoS) preference. They conclude that QoS based approximate revenue auction can generate more revenue than a fixed-price strategy. Mahloo *et al.* [4] compare the cost in terms of capital and operational expenditures of a disaggregated architecture and one based on traditional servers. Their framework results show that disaggregation brings high savings in the presence of heterogeneous workloads.

Borghesi *et al.* [108] present a model to analyze the impact of frequency scaling on energy. To assess the cost benefits for the facility and user, they propose four different pricing schemes and conclude that is possible to save energy while not penalizing users from the economic point of view. Malla *et al.* [116] use an embarrassingly parallel application to provide to HPC cloud users a detailed comparison of cost and performance between two different cloud paradigms, Function as a Service (FaaS) and Infrastructure as a Service (IaaS). Their results show that FaaS can cost 14% to 40% less than IaaS for similar performance.

Ferretti *et al.* [118] introduce a model to help researches to understand whether is convenient to use Cloud infrastructures as an alternative to HPC systems for running scientific applications. Their model takes into account performance, cost, waiting time, and user preference. They concluded that the best infrastructure may be that which optimizes the user's expectations. Breslow *et al.* [112] present a runtime system to enable fair pricing for HPC clusters that run co-located applications and a new pricing model to fairly price applications when co-locations are present. The pricing model provides the user discounts at a rate proportional to the degradation that each of their jobs experiences due to contention. While the runtime system uses a cyclic, fine-grain interference sampling mechanism that for brief periods of execution pauses all applications but one and measures how the selected application's performance changes versus running co-located. This mechanism allows the system to accurately deduce the interference between the applications and use these measurements to drive fair pricing for all users' jobs.

3.4 Summary of Disaggregated Related Works

A common feature shared between the aforementioned related works is that the majority of the listed works intend to solve the disaggregation problem for the cloud/data-center domain, which is not the domain of this thesis. Resource management for the cloud has focused on virtualization techniques to serve applications on a reduced amount of hardware, therefore reaching economies of scale [119]. Then, their approach for attaining disaggregation is frequently through the modification of the hypervisor. On the other hand, HPC systems are designed to maximize the interconnect performance and run all nodes at peak performance simultaneously. Furthermore, its applications tend to require a higher amount of computing resources than cloud services [119].

There has been significant evolution in the sophistication of resource management and scheduling as resources and jobs have become more diverse. A number of schedulers have been developed over the years to address various supercomputing and parallel data analysis as well as computer, network, and software architectures [46]. Compared to the state–of–the–art, to fill the gap between HPC platforms and resource management and job scheduling solutions for disaggregated architectures, our work provides an analysis of resource management and job scheduling decisions for disaggregated architectures considering a popular RJMS used in HPC environments.

In order to reach this objective, we used the Slurm resource manager to adopt memory as a disaggregated resource for job scheduling. We emphasize the Slurm resource manager because it is open-source, popular in research, and widely used in HPC systems, installed on several supercomputers of the TOP500 list. In addition, it has a general-purpose plugin mechanism that supports a wide range of extensions, which makes it suitable to analyze scheduling and resource allocation decisions for our target scenarios. We also employ a simulated environment to evaluate the gains that can be attained with such deployment at scale.

The study is further enhanced with an analysis of the effects of users' ability to estimate their jobs demands and the job's resource utilization on system performance. We also discern from previously mentioned works as we are interested in the dynamic assignment of disaggregated memory to jobs, by dealing with large scale HPC system and adapting the job to the infrastructure through a resource manager capable of modifying the allocated memory assigned to the job. We believe it is an important analysis in a way that helps to understand the dynamics of resource provisioning for disaggregated systems. The recommendations drawn from the analysis would help to support procurement decisions when building large HPC systems.

In comparison, it is also worth mentioning the memory disaggregation model architecture adopted in our work. Previous approaches have employed remote memory access as an I/O block device, treating it as a swap partition within the system. However, this approach introduces overheads associated with handling page faults through the swap mechanism, even when utilizing RDMA for remote I/O operations. Additionally, previous efforts relying on software based remote memory access require significant customization of core system components and the development of custom software, demanding application redesign to access the infrastructure through specifically developed libraries. In contrast, our adopted model architecture utilizes load/store semantics to seamlessly expose disaggregated remote memory as byte-addressable and mapped to a NUMA node. This approach enables applications to leverage the extended memory capacity without requiring further modifications or the need for custom middleware in the system.

Part II:

Modelling Contention-Aware Performance Prediction Technique

Se avexe não
Toda caminhada começa no primeiro passo
Jose Accioly

CHAPTER 4

Slowdown Based Method

THIS part dives into the topic of a generic approach to predict performance degradation due to sharing of resources. The emerging proposal of a novel disaggregated memory architecture to allow a flexible and fine–grained allocation of memory capacity to compute nodes shifts the focus to sharing capacity, rather than coherent sharing of data as in the traditional shared memory processors. In this context, jobs request a number of CPU cores and a given memory capacity per core, and memory capacity is then allocated as a common resource. Since applications running on different nodes can share memory devices and interfaces, performance may be affected by contention [22]. For this reason, co-scheduling and resource allocation decisions for disaggregated memory require contention awareness in order to optimize application performance and overall system throughput. In our work, a proper contention model is an important step, as we need a reliable way of assessing the penalty the applications suffer when sharing remote memory in our simulated environment.

We organize the Chapter by first detailing the hardware used to run our experiments and listing the set of single and multi node applications we profiled to create the contention model in Section 4.1. The hardware detailed in Section 4.1 is also used to run all simulations for our disaggregated environment. The profile collected from all applications will be applied to the evaluation of our implemented disaggregated approaches and to create the input data. In Section 4.2, we describe the concept of the disaggregated approach employed in this work. In the following Section 4.3, we detail the methodology applied in this work to achieve the proposed contention model. And finally, the results of the developed model are presented in Sections 4.4 and 4.5. The content in this Chapter can be found in our papers [39, 40].

4.1 Environment Setup

Hardware resource — We carried out the experiments on BSC Nord III supercomputer which has 756 compute nodes 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/Gs. It implements the home snoop cache coherence with Modified, Exclusive, Shared, Invalid and Forward (MESIF) protocol [120]. The node has 64 GB of Double Data Rate type three (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 socket memory access latency is 81 ns and 133 ns for local and remote accesses respectively [121].

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) [122], Rodinia (5) [123], NAS Parallel Benchmarks (NPB) (8) [124], Splash (5) [125] and another 15 diverse publicly available applications. We selected applications to cover a variety of computational patterns found in multithreaded and high performance codes. The single node applications were compiled with GNU/Linux GNU Compiler Collection (GCC) 7.2 and multithreading enabled, while the distributed applications were compiled using OpenMPI 1.8.1. We used *numactl* [126] to apply affinity settings for both threads and memory placement and the *Perf* [127] tool to collect the performance counters.

4.2 Problem Definition: Global Memory Emulation

Since the concept of remote memory decoupled from the processor is not easy to implement in real prototypes and due to the absence of an available prototype [21], we emulate a disaggregated shared memory architecture (following the concept presented in Section 2.2), without the need for real hardware, using a conventional multi-socket server. This approach takes advantage of a two-socket server and its separated LLC to create pressure only in the desired shared resource, i.e. memory bandwidth. According to Molka *et al.* [121], 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 the cross-socket memory access for our experimental platform detailed in Section 4.1, is similar to those presented in disaggregated works such as [8, 9, 69].

Consequently, from a software perspective, in our emulation the memory appears to the remote compute node as a CPU-less NUMA node where its memory characteristics are independent of the memory directly attached to it [69]. However, the access is subject of contention as multiple compute units can share the same memory resource.

Emulating Disaggregated Memory Single Node — To mimic a disaggregated shared memory architecture for a single node application, the approach developed in this work applies the scheme depicted in Figure 4.1. As shown in the Figure, all threads of the interfering application (B) execute on the socket 2 while issuing memory requests exclusively to the memory bank attached to socket 1 (remote access). On the other hand, all threads of the target application (A) use only its memory bank (local access), thus contending for the memory controller and memory bandwidth. Applying this scheme, the impact of application B on application A can be modeled based on B's bandwidth interference. This model predicts the slowdown experienced by application A in the face of diverse remote bandwidth demands.

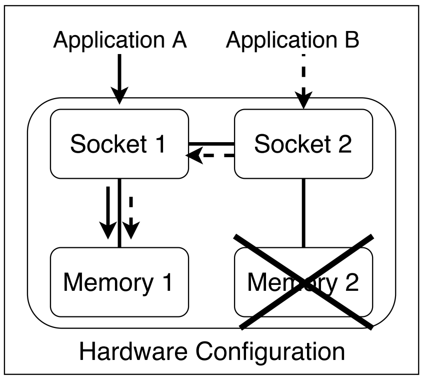


Figure 4.1 Emulated disaggregated scenario applied in this work.

Emulating Disaggregated Memory Multi Node — To mimic a disaggregated shared memory architecture for a multi node application, we extended the previously detailed scheme and depicted its flow in Figure 4.2. All threads/processes of an interfering application (B/C) execute completely on the socket 2 in each node while issuing memory requests exclusively to the memory bank attached to socket 1 (remote access) on the same node. On the other hand, all threads/processes of a target application (A) use only its memory bank (local access), thus contending for the memory controller and memory bandwidth. Thus, the impact of application B and/or C on application A can be modeled based on the overall bandwidth interference. For distributed applications running across different nodes, the main difference compared with the single node scenario is that 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 the 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 demands.

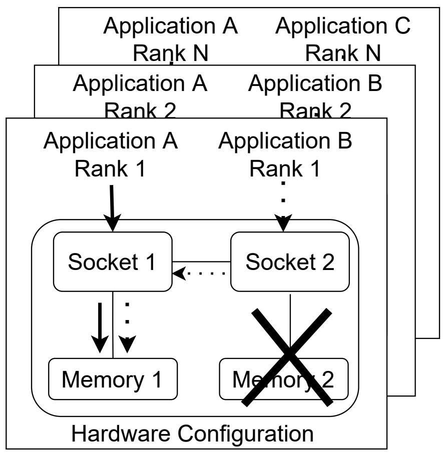


Figure 4.2 Emulated multi node disaggregated scenario.

4.3 Slowdown Method

The most straightforward way to approach the problem of characterizing the performance degradation of an application in contention is to perform a brute-force sweep. According to De Blanche *et al.* [79], from all methods to characterize the performance degradation of an application in contention the brute-force method is the most precise. However, due to the $O(N^2)$ cost, it is not possible to be employed in a production environment. A suitable alternative is to apply a method that analyzes each program once and scales linearly with the number of applications. Prior works [28, 79, 27, 80, 29] predicted performance degradation when two applications run together using the concepts of *sensitivity* and *contentiousness*. Their goal is to predict, for any pair of applications, the slowdown that results from contention in the memory subsystem, without the need to run all combinations of applications against each other.

These Slowdown based methods are decoupled into two steps: (1) to measure the sensitivity curve, which quantifies the impact of different levels of shared resource contention on the application's performance; (2) to measure the contentiousness value, which quantifies how much shared resource contention is generated by the application. Then, to predict the application's performance the contention is applied to the sensitivity curve. In both cases, pressure may be quantified in various ways, for example, in terms of cache capacity and/or memory bandwidth. Slowdown based methods have been successful for single node coscheduling [27–29], but they have not been applied to disaggregated memory.

In this regard, we must take special care when using the Slowdown models. Since in our envisaged disaggregated scenario (Figure 2.7 and Section 4.2), the applications do not share cache capacity, the contentiousness must be solely by using memory bandwidth. In this sense, our Slowdown based method for disaggregated memory creates a family of sensitivity curves to relate computing demands for memory bandwidth to application degradation. These sensitivity curves are built using a carefully curated set of performance counters that correlate well with the memory bandwidth of the application. Since HPC applications are generally batch applications (rather than interactive services), performance is inversely proportional to runtime, which for a given application is itself proportional to memory bandwidth.

4.3.1 Single Node Approach

Figure 4.3 shows a high-level view of our Slowdown methodology for a single node, which is comprised of three components:

① **Interference phase** — In the interference phase, a set of interfering applications execute concurrently with the target Application A, using only the remote bandwidth as the measure of pressure. The interfering applications differ as they account for different read/write ratios to create a family of distinct sensitivity curves (see Section 4.3.1.1). In the disaggregated memory architecture model, remote memory accesses do not create cache contention in the local node as their cache hierarchies are separate (see Figure 2.7). Contention induced by cache capacity can misrepresent the degradation experienced by the applications in such a scenario, so the metric of interest is bandwidth usage and contention on the memory controller.

② **Bandwidth calculator** — The bandwidth calculator is responsible for collecting hardware counters to calculate Application B's remote bandwidth (bw_b) and its read/write ratio (R/W Ratio). The calculated values will be used to select the appropriate target's sensitivity curve to predict the performance degradation (see Section 4.3.1.2).

③ **Prediction methodology** — In the prediction methodology, we start by selecting application A's sensitivity curve taking into account Application's B read/write ratio. Then, to mitigate the measurement noise from the benchmark process and improve the prediction accuracy, a smoothing function is applied to the sensitivity curve. Linear and polynomial smoothing functions are considered, as they are straightforward and commonly employed. This gives the function (f_a) that relates Application A's slowdown to the total interfering remote bandwidth. Finally, the Application's B bandwidth (bw_b) is applied to the function to predict Application A's performance (y) when Application B contends for remote memory bandwidth (see Section 4.3.1.3). The prediction methodology has $O(cN)$ complexity (N as the number of applications and c the number of distinct read/write ratios), as it requires the application to be characterized only once instead of every pairwise execution that would be required in a brute force characterization scenario.

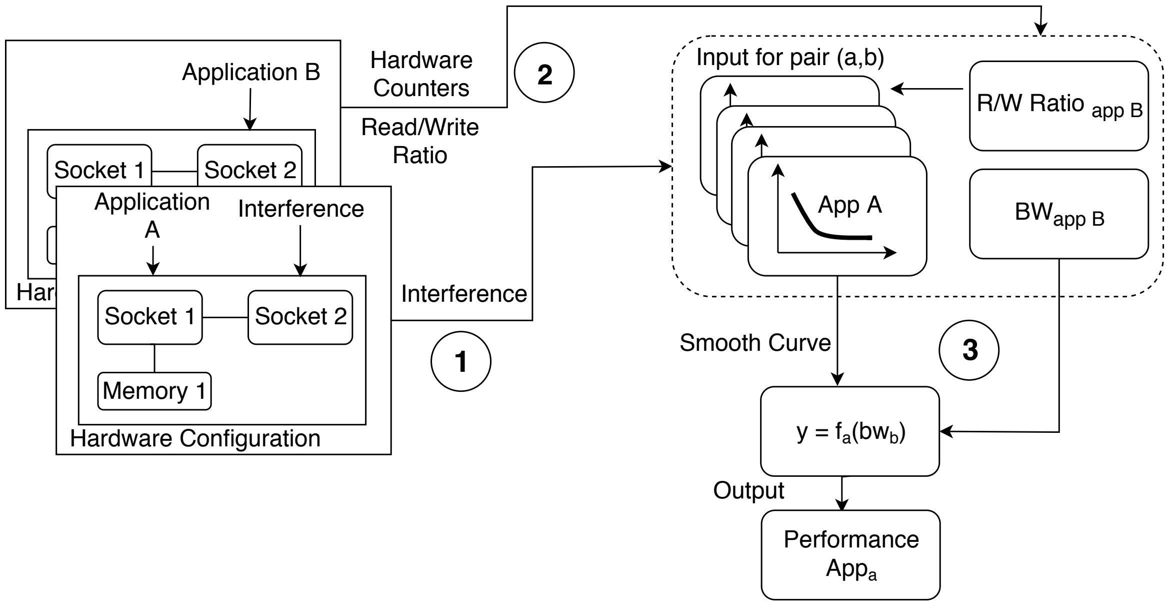


Figure 4.3 High-level view of the Slowdown methodology.

The proposed Slowdown method differs from previous works [29, 28] as the first work can only estimate the performance of a number of identical applications (with the same input data). Besides, the work carried out in [28] does not estimate the performance for remote memory interference and it uses a static read/write ratio interference. As can be seen in the Figure 4.3, our approach relies on three distinct aspects, which diverge from the initial common approach.

The first aspect is the interfering application executing concurrently and using only the remote bandwidth as the measure of pressure. The interfering applications differ as they account for different read/write ratios to create a family of distinct sensitivity curves for each target application. In the upcoming global shared memory address architectures, remote memory access does not create cache contention in the local node as their cache hierarchy will be separate. Contention induced by cache capacity can misrepresent the degradation experienced by the applications in such a scenario, so the metric of interest is bandwidth usage and contention on the memory controller.

The second aspect is using a suitable hardware counter to correctly calculate the application's remote bandwidth and its read/write ratio to select the appropriate curve to predict the degradation. And the third aspect is smoothing the curve points by applying a linear function for better accuracy. Furthermore, the approach maintains the $O(cN)$ complexity (N as the number of applications and c the number of distinct read/write ratios) as it requires the application to be characterized only once instead of every pairwise execution that would be required in a brute force characterization scenario. The problem is modeled in such a way because in the adopted architecture and for recent disaggregated memory models (see Section 2.2) accessing remote memory does not create cache contention in the local node, so the metric of interest is bandwidth and contention on the memory controller.

4.3.1.1 Creating Sensitivity Curves

To create the sensitivity curves, we quantify the memory pressure using the total remote bandwidth rate and calculate the performance degradation the application suffers when executing with an interfering application. For this purpose, we use a synthetic benchmark, which is an adaptation of the STREAM benchmark [128], modified to generate a variable requested bandwidth for the remote memory. Then, we create a curve of the target application's performance, normalized to its performance running alone, on the y-axis, versus the remote bandwidth generated by the modified STREAM benchmark (interfering application), on the x-axis. An example of the sensitivity curve for the Triad workload from the STREAM benchmark is presented in Figure 4.4.

Radulovic *et al.* [129] argue that the total bandwidth is misleading because it combines into a single metric the aggregate bandwidth of reads and writes, even though they are fundamentally different operations. Distinguishing read and write bandwidths when creating the sensitivity curve not only accounts for the particularities of the memory subsystem but also more accurately represents the behavior of real applications. Variable read/write ratios are supported by the synthetic interfering benchmark [130].

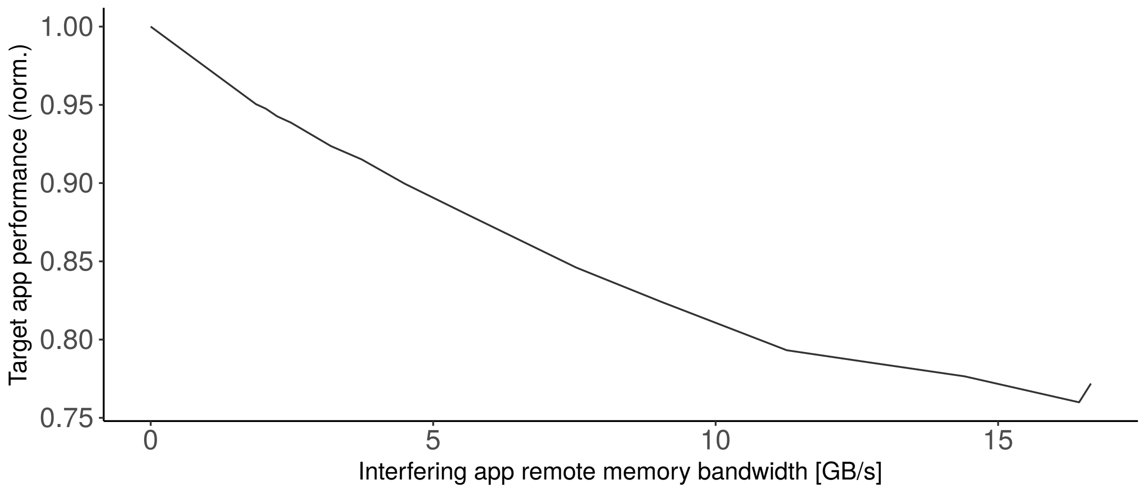


Figure 4.4 Sensitivity curve for Triad workload.

We applied the Slowdown based methodology by creating a family of sensitivity curves that depend on the interfering application's remote memory bandwidth and read/write ratio. Previous works rely on one single sensitivity curve, however in Figure 4.5 we show the measured sensitivity curves for the Triad workload on our experimental platform (more detail in Section 4.1) while the proportion of reads varies between 50% and 100%. In all our experiments, curves created using a higher percentage of reads generally achieved a higher sustainable bandwidth. As pointed out in [129], writes have additional delays caused by the write recovery time, which is the delay between a write and the next precharge command, and the write–to–read delay time, which is the interval between a memory write and the consecutive read. Increasing the proportion of write requests, therefore, reduces the sustainable bandwidth and increases the effective (loaded) latency [129].

An important aspect when creating a sensitivity curve is the representation of pressure in its x-axis. When executing the synthetic interfering benchmark and the target application concurrently, both applications will interfere with each other. Intuitively, the reported interfering bandwidth will be lower than the expected bandwidth of its execution alone.

In our methodology, we use all curves to display the results in Sections 4.4 and 4.5 with the x-axis representing the bandwidth that the interfering application would have had if it were executing alone instead of under contention. This is similar to the Figure 4.5. We chose this methodology because it is generic to any interference study, and it is appropriate for the scenario where the known value is the actual bandwidth usage that the interference application applies.

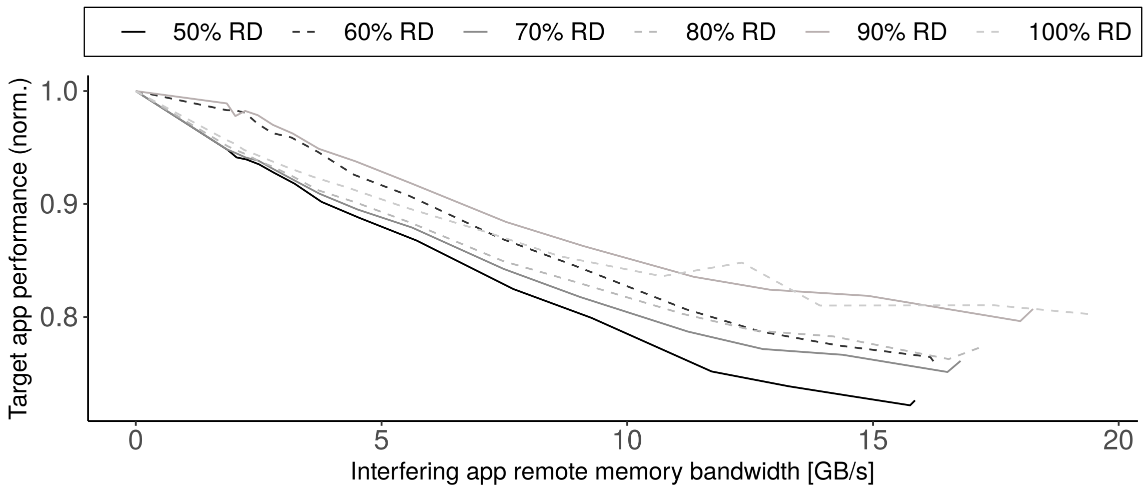


Figure 4.5 Family of sensitivity curves for Triad workload. Each line corresponds to a specific read/write ratio going from 50% Read to 100% Read, with the lighter lines indicating a higher percentage of read.

4.3.1.2 Measuring Contentiousness

In the second part, we characterize the application in terms of how much remote bandwidth it requires throughout its execution by running it solo. The value of remote memory bandwidth will be applied as the contention pressure. For this step, we use performance counters to measure the read and write bandwidths.

Memory bandwidth is typically measured using one of the three approaches: (a) LLC miss [84, 131], (b) Off-core response [132] and (c) uncore Integrated Memory Controller (IMC) counters [133]. While the LLC miss counter accounts for neither the prefetcher read memory traffic nor write memory traffic, off-core response counters cannot detect write memory traffic due to cache line evictions, which is the major portion of the overall write traffic.

The uncore IMC counters, on the other hand, measure events in the memory controller, including read and write Column Access Strobe (CAS) commands [120]. These commands are sent from the memory controller to the Dynamic Random-Access Memory (DRAM) at every memory column access and they include prefetching and eviction events. Each memory access consists of a cache line of 64 B. As the counter measures memory traffic at the memory channel, only the total read/write traffic per channel is measured, which prevents further segmentation of requests. Its broad range makes it suitable to compute application bandwidth with higher accuracy than LLC misses, providing a complete bandwidth profile. Following the measurements, the per-channel read and write application bandwidths are:

$$\begin{aligned} BW_{\text{read}} &= CAS\_COUNT_{\text{read}} \times 64\,\text{B}/elapsed\_time \\ BW_{\text{write}} &= CAS\_COUNT_{\text{write}} \times 64\,\text{B}/elapsed\_time \end{aligned} \tag{4.1}$$

and the total bandwidth is, therefore:

$$BW_{\text{tot}} = BW_{\text{read}} + BW_{\text{write}} \tag{4.2}$$

4.3.1.3 Prediction Methodology

The penultimate step in predicting the application's performance is to calculate the read/write ratio of the interfering application. It is done by dividing Equation 4.2 by BW_{read} to obtain the percentage of memory operations that are reads. As the last step, we select the target application's sensitivity curve corresponding to the read/write ratio of the interfering application. The selected curve is smoothed using a linear function so as to predict the degradation in the case of missing points with higher accuracy and also to decrease the influence of outliers in the collected data. Finally, we apply the interfering remote memory bandwidth to the target interference curve to obtain the expected target performance when running under interference.

4.3.2 Multi Node Approach

Our multi node contention model is an extension of the single node model presented in Section 4.3.1. It was extended to support contention among applications that use multiple nodes. The overall approach is shown in Figure 4.6 and the model input and output are given in Table 4.1.

Sensitivity Curves — 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. 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. Then, moving from single 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.

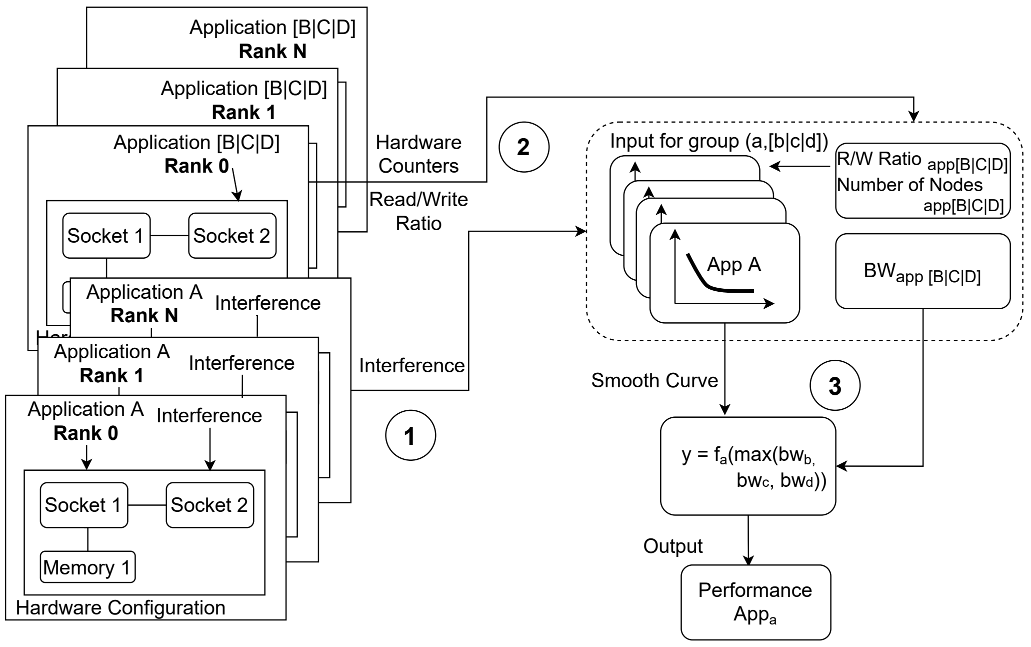


Figure 4.6 Multi node Slowdown methodology.

Table 4.1 Contention model inputs and output

Value	Description
Application inputs:	
$f(bw, N)$	Slowdown curve: normalized performance as a function of interfering bandwidth and number of nodes
R/W Ratio	Percentage of memory operations that are reads
bw_{app}	Total memory bandwidth (bytes/s)
Execution inputs:	
bw	Interfering bandwidth
N	Number of interfering applications
Output:	
P_{est}	Estimated normalized performance, typically $0 \leq P_{\text{est}} \leq 1$

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 (see Figure 4.7). 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.

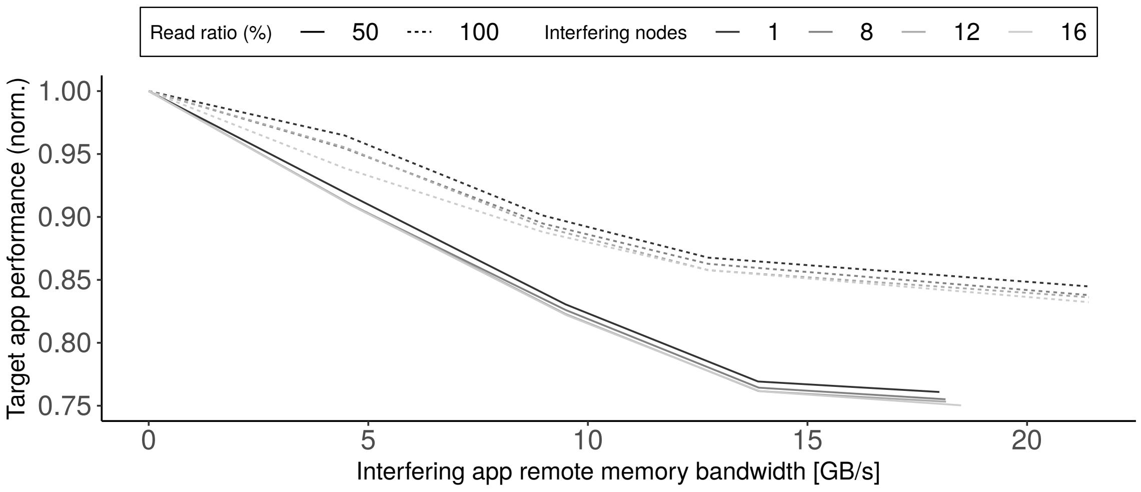


Figure 4.7 Distinct sensitivity curves for stream benchmark.

This introduces a parameter, which is the tolerance within which a node's estimated interference is counted as being close to the maximum. We explore the effect of this parameter in Figure 4.8, which shows the prediction error as a function of this tolerance, with "single node" meaning that the highest interference is assumed to affect a single node. In contrast, the 0% threshold considers all nodes that have the exact same numerical interference value as the highest interfering bandwidth. On the other hand, the range from 25% to 100% includes only nodes whose interfering bandwidth falls within that percentage range compared to the maximum interfering bandwidth.. We see that there is a large improvement moving from the single node case, across all benchmarks, but the precise value of the tolerance parameter was not significant in our experiments. There are two likely reasons for this behavior. Firstly, since estimated interference is calculated from the model rather than measured on a system, all nodes that share the memory with processes of the same applications will see precisely the same estimated interference, and no tolerance is needed for random noise. Secondly, the impact of contention on performance stabilises quite quickly with the number of nodes experiencing a similar level of contention, so precisely determining this number is generally not critical. We, therefore, choose a moderate tolerance of 25% for the rest of the experiments.

To extend the contention model, we first execute the synthetic benchmark [130] to create the sensitivity curve in parallel across a configurable number of interfering nodes (Step 1 of Figure 4.6). Then, we measure the sensitivity curve for 50% reads and 100% reads and use linear interpolation for intermediate read/write ratios. The single node model has shown that linear interpolation exhibits better performance than additional interfering data points (see Section 4.4.4) and that the accuracy is 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.

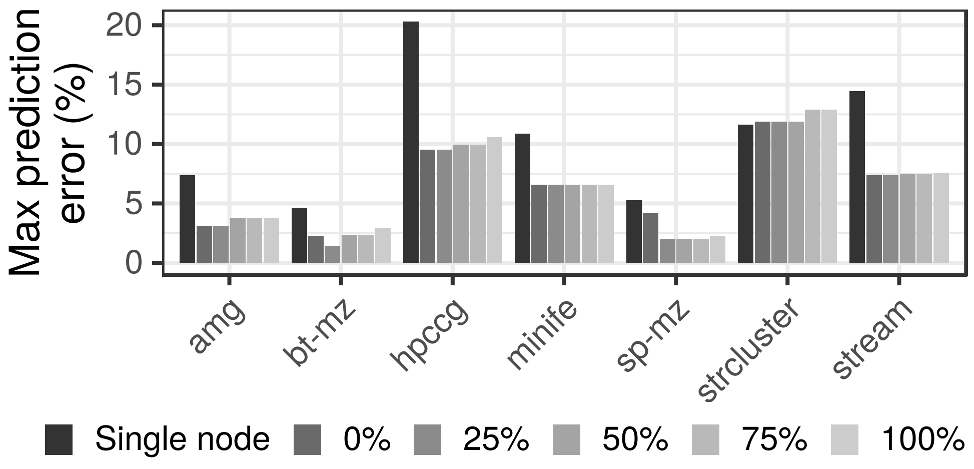


Figure 4.8 Maximum prediction error for multi node applications running on 31 nodes.

Contentiousness — 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 CAS commands [120], averaged over all nodes on which the application is executed.

Predict Methodology — 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.

4.3.3 Key Differences Compared with State-of-the-art and Sources of Error or Simplification

Our work is differentiated from prior work [27–29], and provides improvements for a singular reason: Our approach targets performance prediction due to sharing of disaggregated memory, while the prior works use application working set size or local bandwidth as their measure of pressure to create the sensitivity curve. As noted in Section 4.3, the cache contention characterization method is misleading for predicting the performance of applications using separated cache hierarchies. For this reason, we proposed using a family of smoothed sensitivity curves to account for varying ratios of read and write memory accesses to increase accuracy and decrease the effect of outliers. Beyond that, our model deals with shared memory contention for multi node applications.

We can list two main sources of simplification in our model. The first one is that we model contention and remote access penalty using a fixed latency, which is a result of our disaggregated emulation approach. We are aware that the applications may be affected by distinct latencies accessing remote memory, therefore in Chapter 6 we discuss the effect of this assumption in our results. The other drawback of our approach is that it does not model contention in shared network access when multiple applications are communication intensive. We prioritize the memory access contention model to address the most critical problem since memory access is one of the main sources of contention. Furthermore, aggregating multiple sources would increase the complexity of the simulation, making simulation more complex and slower. Nevertheless, the model can be extended to account for networking, multiple latencies, or any other source of contention to improve the simulation.

4.4 Experimental Evaluation Single Node

For this Section, the resources used to evaluate the accuracy of the proposed approach were presented in Section 4.1 and the simulated global shared memory architecture using a conventional node in Section 4.2. We normalize our results against a brute-force sweep performed on the pair of applications as demonstrated in [79].

4.4.1 Performance Counters

To compare the accuracy for representing the bandwidth of an application, we collected the performance counters listed in Section 4.3.1.2 during the execution of the Triad workload. Performance counters can track the occurrence of events with negligible overhead, and there are commonly employed in many related works [91, 84, 134] to measure resource utilization and demonstrate the effectiveness of any proposed method. Figure 4.9 summarizes the calculated bandwidth derived from the collected performance counters. In the Figure, the dark columns represent the bandwidth for the read traffic while the light columns represent write traffic. As can be seen, cache miss counter represents only a fraction of the sustainable memory bandwidth (5 GB/s), confirming that not only write traffic but part of the read traffic is neglected by this performance counter. The off-core response counter displays a bandwidth of about 27 GB/s which is roughly equal to the read traffic calculated using CAS counter. However, the difference between both counters emerges in the write traffic captured by the latter, which differs by 30% of the overall bandwidth.

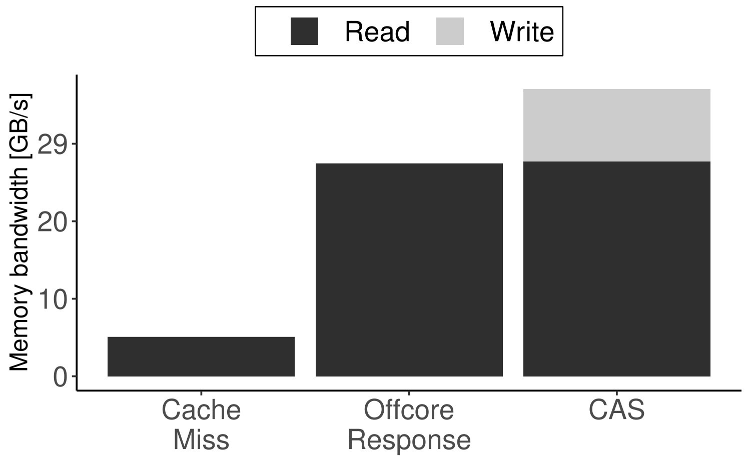


Figure 4.9 Calculated memory bandwidth for Triad workload using three different hardware performance counters.

The bandwidth calculated by Triad during the tests was 30 GB/s on average. According to McCalpin [135], the bandwidth values assumed by the Triad workload is based on the minimum data traffic that each iteration will perform. For the Triad workload, this number is 24 B (two floats read and one float write), however, it does not account for bytes transferred during write-allocate operations. As the Triad kernel requires 4/3 as much bandwidth as the benchmark generates when write-allocate is included, the result must be multiplied by a factor of 1.33. After applying the correction factor, the sustainable bandwidth is approximate 39 GB/s. The total bandwidth calculated for Triad workload using CAS counter deviates by only about 5% of the scaled bandwidth, which confirms its accuracy to be sufficient for the proposed methodology.

4.4.2 Application Characteristics

Figure 4.10 and 4.11 present the characteristics of bandwidth usage for each application considered in our experiments. The applications used in this study show a wide range of memory bandwidth utilization. At least 40% of the applications use 20% or more of the sustainable bandwidth. The sustainable bandwidth for this work is calculated using the highest value achieved among 5 executions.

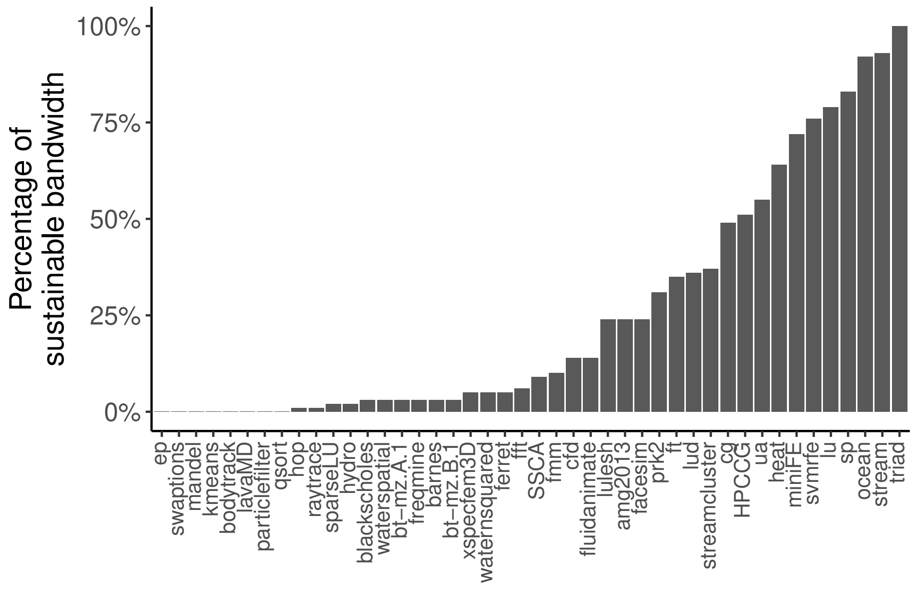


Figure 4.10 Percentage of sustained memory bandwidth utilization for each application in our study. Memory bandwidth usage is calculated using CAS performance counter.

Figure 4.11 shows that the benchmark suite covers a wide range of read/write memory traffic, with reads accounting for between 50% and 100%. The Figure also shows that the family of curves (see Figure 4.5) corresponds to the range that arises in practice.

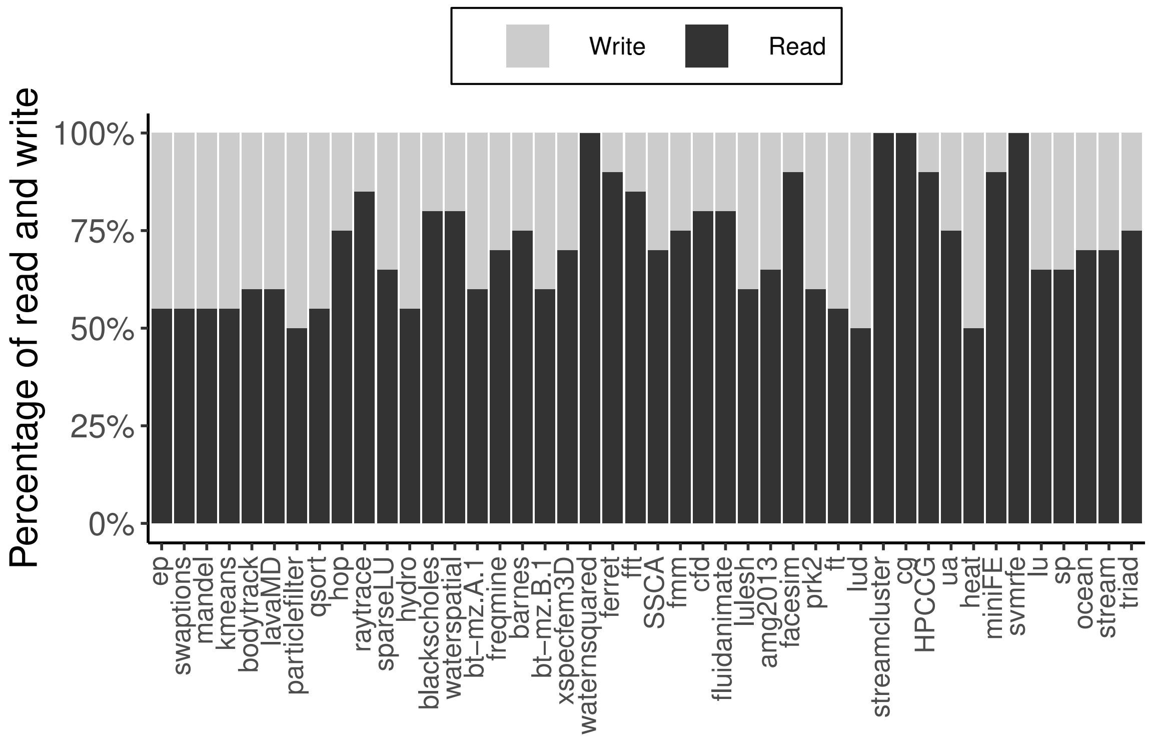


Figure 4.11 Percentage of read/write ratio for each application considering their sustained bandwidth utilization.

4.4.3 Sensitivity Curves

For our tests, the interfering application generates the sensitivity curve for remote memory bandwidth traffic between 10% and the maximum sustainable bandwidth achieved in the node. We then apply a linear function to smooth the curve before predicting the application's performance. We tried different smoothing functions, and the linear function attained satisfactory results. As applications with low bandwidth usage do not present a high drop in performance under remote memory access interference, in Figure 4.12 we only present those applications with a significant drop in performance using the data recorded during the interfering benchmark without any transformations. The sensitivity curves for all applications can be found in Appendix A, Figure 1.1.

As represented in previous figures, lighter lines imply a higher read percentage. Due to high memory bandwidth usage, in Figure 4.12 we notice that the applications are highly affected by remote interference. The Figure also shows the raw data recorded during the interfering benchmark. We can observe the presence of noise in the results for some applications (e.g., *lu*, *sp*, and *heat*).

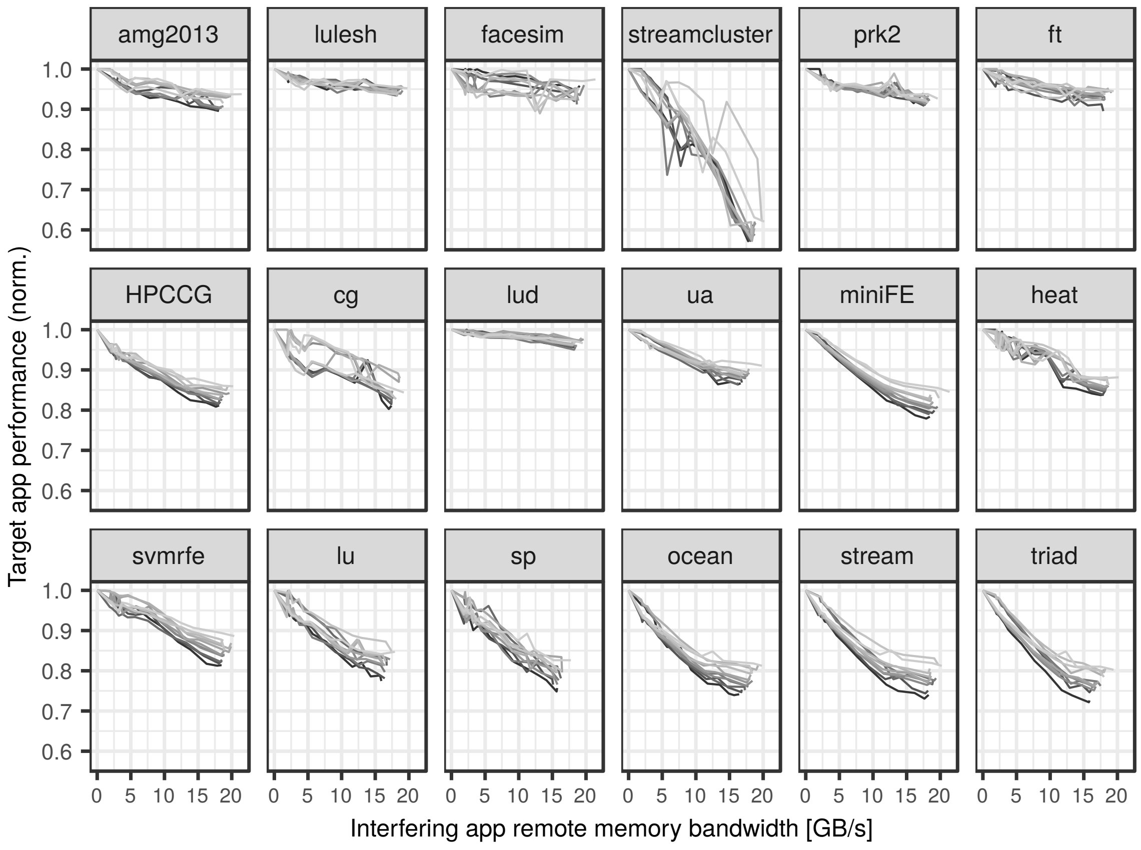


Figure 4.12 Sensitivity curve for applications with high bandwidth usage. Each line corresponds to a specific read/write ratio going from 50% Read to 100% Read, with the lighter lines indicating a higher percentage of read.

When applying pressure equal to the maximum sustainable remote bandwidth, we noticed that the performance of the applications suffers considerable levels of degradation in the face of an interfering application issuing memory requests from a remote node. This indicates that the resource allocation for new architectures using global memory addresses must account for the degradation of applications executing concurrently and sharing the memory subsystem. The requirement and effectiveness of the interfering application are endorsed, as increasing the amount of interference causes a decrease in the target application's performance. However, applications are not affected in the same amount, even though *lud* and *ft* have similar bandwidth requirements (see Figure 4.10), the former is penalized more than the latter, the same happens to *HPCCG* and *ua*. This behavior highlights that some applications are also even more penalized for high latency when handling remote memory requests from the interfering application.

Also as presented in the Figure 4.5, we can notice the difference in the performance when the read/write ratio of the interference application varies. Increasing read ratio, less interference the benchmark causes to the target application, then more sustainable remote bandwidth is used by the remote application. This pattern is emphasized with high remote memory access when we can clearly see the separation between curves.

4.4.4 Prediction Error

We evaluate the effectiveness of our contention model by predicting the expected performance degradation of the applications in a pairwise fashion. We consider the prediction error to be the absolute value of the difference between the predicted performance and the real normalized application performance under contention. The methodology applied in our tests is the following: we start both target and interfering applications at the same time on different sockets. We restart the interfering application whenever it finishes keeping the target application in contention during its entire execution. All pairwise combinations were executed in such a way that the target application completed at least 10×. We recorded the actual performance degradation (increasing in runtime) when the applications executed together, in order to compare with the predicted performance. Degradation for the target application is calculated using its normalized performance alone in the system without interference and its performance under contention.

Table 4.2 summarizes the predicted error for all applications using two different functions to smooth the interference curve. The smoothing functions applied to create the curves are linear and polynomial functions with degree two. It also shows the prediction error using the smoothed specific curve for the interfering application read/write ratio (called *right curve*) and the smoothed sensitivity curves produced with the static values of 50%, 75% and 100% read/write ratio. The column *Cost* indicates the profiling cost needed by the method. It is compared to the methodology applied by previous works that rely only on 4 different levels of pressure (listed 25%, 50%, 75% and 100% contentiousness) and a static read/write ratio to create their sensitivity curve.

Table 4.2 Prediction error for linear and polynomial smoothing

Method	% of Reads	Mean (%)	SD	Max (%)	Cost (×)
Polynomial	**right curve**	**1.15**	**1.49**	**14.5**	**44**
	50	1.29	1.86	20.6	4
	75	1.34	1.86	17.7	4
	100	1.39	2.20	21.0	4
Linear	**right curve**	**1.19**	**1.59**	**14.0**	**44**
	50	1.27	1.86	22.1	4
	75	1.38	1.94	18.9	4
	100	1.49	2.32	19.5	4

We achieve the lowest mean and variance prediction errors using the curve corresponding to the actual read/write ratio of the interfering application. In the best case, the right curve using polynomial smoothing attains a mean error of 1.15%, and a relative improvement of 18% compared with predictions using a static read/write ratio. In addition, it also has the lowest worst case with around 14% max error and relative improvement between 19% and 31%. The linear smoothing method achieves a mean error of 1.19%, which is similar to the polynomial smoothing results, and relative improvements between 26% and 37% for the worst case. Figure 4.13 presents the distribution of prediction error using polynomial smooth function (best result so far).

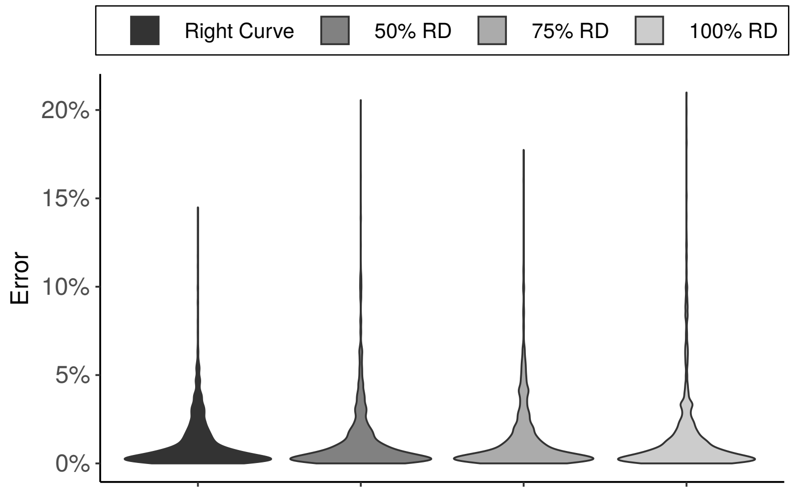


Figure 4.13 Distribution of error using polynomial smoothing function.

In the Figure , we can note that the density of the prediction error using the correct read/write ratio is more concentrated in the base of our chart and the majority of the values is below 5%. Using static curves we note predictions with high variance and the presence of a longer tail, denoting the occurrence of higher values for errors. Using fixed read/write ratio curves the max error is higher than 15%, however, the max error using the right curve is lower. The results confirm that the read/write ratio plays an important role to predict performance degradation.

To further evaluate the influence of noise and outliers on our predictions, we compared the smoothing results against the non-smoothed interpolation method used in previous works. We assessed the method using all data points (16 distinct values of interference or contentiousness) collected during the interference benchmark and a sampled version using four distinct interference values (25%, 50%, 75%, and 100%) to create the interference curve. Table 4.3 summarizes the overall result. We do not see an overall improvement compared to the smoothing values when using all data points and right curves. Even though it has a mean error similar to linear smoothing and better results than the static curves, its worst case prediction increases. This can be viewed as a side effect of the noise intrinsic to the collected data for all sensitivity curves calculated.

Table 4.3 Prediction error for smoothing functions and sampled data

Method	**# of Points**	**% of Reads**	**Mean (%)**	**SD**	**Max (%)**	**Cost (×)**
Polynomial	All points	**right curve**	**1.15**	**1.49**	**14.5**	**44**
Linear	All points	**right curve**	**1.19**	**1.59**	**14.0**	**44**
Interpolation	All points	**right curve**	**1.19**	**1.63**	**24.2**	**44**
		50	1.34	1.89	18.3	4
		75	1.34	1.82	19.4	4
		100	1.43	2.19	21.0	4
Interpolation	Sampled	**right curve**	**1.35**	**1.62**	**18.1**	**11**
		50	1.47	1.91	18.1	1
		(Memgen) 75	**1.56**	**1.94**	**21.6**	**1**
		100	1.58	2.18	24.7	1

By sampling the data points and using the right curve, the interpolation maintains the lowest mean error compared to its static counterparts. However, it increases the worst case prediction error compared to the smoothing methods and decreases it compared to using all data points. A positive side effect of sampling the data for the right curve results is that part of the noise is removed and the interpolated values provide a more stable result, which explains its results compared with the other methods. This aspect illustrates that the overall accuracy is affected by the data points and smoothing function, which decreases the effects of outliers and improves the worst case predictions.

Another important point to be considered is the effective cost to achieve the results. In Table 4.3 for our initial approach the high cost of using the right curve can be broken down into two components: the number of read/write curves and the number of data points collected. In our tests, we analyzed 11 different read/write ratio curves (from 50% to 100% increasing 5% each step) and we also sampled 12 supplementary points (interference or contentiousness values) in addition to the default 4 points used in previous works. Even though polynomial smoothing has the best results so far, it also has the highest cost to compute. The cost is 44× higher than using a single sampled static curve due to the additional curves collected and the number of points.

To investigate the effects of the trade–off between accuracy and cost, we reduced the cost of our methodology by estimating the values for the right curve based on two sampled curves. For this test, we used the values of 50% and 100% sampled sensitivity curves collected for the previous test, and we estimated the performance of the target application based on the read/write ratio proximity of the interfering application to both curves. The process is shown in equation 4.3. First, we calculate the proximity (*scale*) of the interfering application read/write ratio to 100% ratio. Then, we predict the performance of the target application for 50% (*P 50%*) and 100% (*P 100%*). In the end, we apply a weighted mean estimation to determine the estimated performance (*P_est*).

$$\begin{aligned} scale &= (interf\_ratio - 50)/(100 - 50) \\ P\_est &= ((P50\%) \times (1 - scale) + (P100\%) \times (scale)) \end{aligned} \tag{4.3}$$

Table 4.4 summarizes the overall result of reducing the cost of the methodology. Sampling the data points decreases to 11× the cost of using the right curve for the smoothing methods, as we keep the same number of read/write ratio curves we collected. A smoothed sampled curve also keeps the mean and maximum prediction errors lower than the sampled static curve. However, we achieve the best trade-off between cost and error in estimating the values for the right curve. We reduce the cost to only 2× and the results are similar to those using a higher number of curves and data points.

The linear method is better than the polynomial one with 1% of difference for max error, thus we chose it to compare with the state–of–the–art.

Table 4.4 Prediction error for estimated curves

Method	**# of Points**	**Mean (%)**	**SD**	**Max (%)**	**Cost (×)**
	All data	1.15	1.49	14.5	44
Polynomial	Sampled	1.21	1.43	15.4	11
	Estimated	**1.19**	**1.48**	**15.6**	**2**
	All data	1.19	1.59	14.0	44
Linear	Sampled	1.21	1.48	13.7	11
	Estimated	**1.19**	**1.50**	**14.6**	**2**
Interpolation	(**Memgen**) **75**	**1.56**	**1.94**	**21.6**	**1**

4.4.5 Comparison with Memgen

In this section, we compare our methodology to the state–of–the–art Slowdown based methodology, Memgen [28]. Memgen uses a modified version of the Triad workload to generate a specific amount of traffic, then creates contention for the target application. Memgen creates four reference points which are 25, 50, 75, and 100% of the sustainable bandwidth as its interference pressure. To create the sensitivity curve, Memgen relies on linear interpolation between the reference points in order to estimate the performance, and defines the contentious pressure as the memory bandwidth usage of an application. To evaluate the applications' memory bandwidth usage, Memgen uses the bus_trans_mem.self performance counter.

To apply the Memgen methodology to our environment, we adapted some unavailable features that will be discussed hereafter. Their triad version code is not available online, so we could not use the same interference application to create the sensitivity curve. However, as their version is based on the triad algorithm from STREAM, we applied our interfering application that generates a 75% read/write ratio. Using this specific configuration to create the sensitivity curve, we maintain the same static read/write ratio and computing characteristics used by the authors in their work. Another point regarding the sensitivity curve is that [28] and [79] do not specify whether the values of the x-axis are the maximum bandwidth or the interfering bandwidth in contention. We assumed the same approach applied in our work, using the maximum bandwidth achieved during the interfering test.

Another adapted feature concerns using the bus_trans_mem.self performance counter [136] to compute the application's memory bandwidth. This counter is unavailable in the Sandy Bridge processor architecture [120] which is used in our experiments. As a consequence, to calculate the contention pressure we applied the CAS performance counters to measure the application's memory bandwidth. These performance counters calculate with high precision the actual bandwidth for applications keeping the comparison fair. The result of the comparison with Memgen is shown in Figure 4.14.

In the Figure we can see that the error for the Memgen methodology is higher than using our methodology. For our approach, the distribution of errors is more concentrated in the base, which means that it has the majority of its predictions close to the true value. The Memgen methodology has a relative mean error increase of almost 24% (see Table 4.4 Interpolation method). This is also true for the worst case, where the max error is 21.6% using the Memgen approach and 14.6% using our approach. These results highlight that by distinguishing the interference caused by different ratios of read/write and increasing the number of points collected, the Slowdown method can be improved.

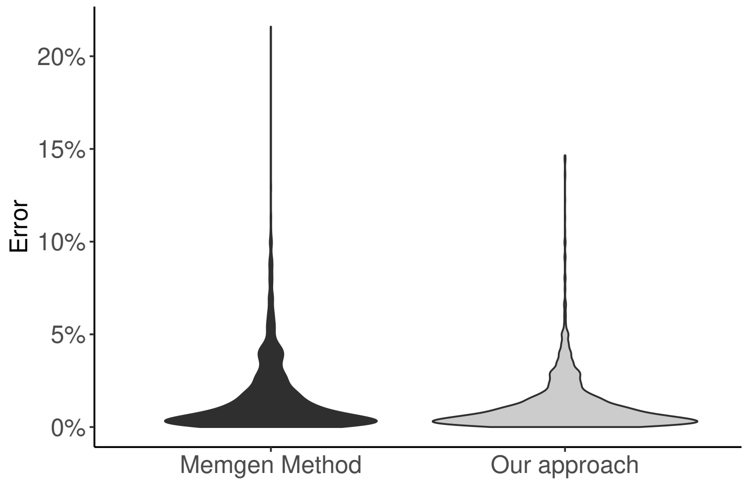


Figure 4.14 Distribution of error for Memgen methodology and our approach.

Another important point to be considered is the effective cost to achieve the results. In Table 4.4 the column *Factor* indicates the cost to achieve the results compared to the Memgen approach. The cost of using the right curve can be broken down into two components: the number of read/write curves and the number of points. In our tests, we analysed 11 different read/write ratio curves while the Mengem approach uses only one read/write ratio curve. Comparing any right curve results with its static counterpart, we can see that all mean and max errors are lower, except using all data points where we can confirm that outliers have a considerable impact on the results. Our estimated method was able to decrease our overall prediction cost and still achieve lower mean error than the static read/write approach.

4.5 Experimental Evaluation Multi Node

We evaluate the effectiveness of our multi node contention model that predicts the degradation of a target application when it experiences different levels of interference while running on different nodes. For the multi node approach, we followed the same methodology applied to the single node approach. 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 7×. 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. To visualize all sensitivity curves for the multi node applications see Appendix A.

In our analysis we use at most 31 nodes due to the limitations of our access to the infrastructure. We are only allowed to reserve up to 32 nodes on its resource manager. To correctly profile the applications, in this reservation, we have to use one head node to run the scripts that control the experiment, otherwise, it would interfere with the results. The script would take away one core from the local or remote application leaving fewer resources for the computation and thrashing the performance counters. In this regard, it is sensible to separate one node to exclusively run the script.

Figures 4.15 and 4.16 present the results for a mix of interfering applications that vary in contentiousness (see Section 4.3.2), nodes, and read/write ratio. Figure 4.15 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 (~8× increase), the maximum error for any application increased at most 3×. Demonstrating that we do not compromise the accuracy of the model when increasing the number of nodes up to 31 nodes.

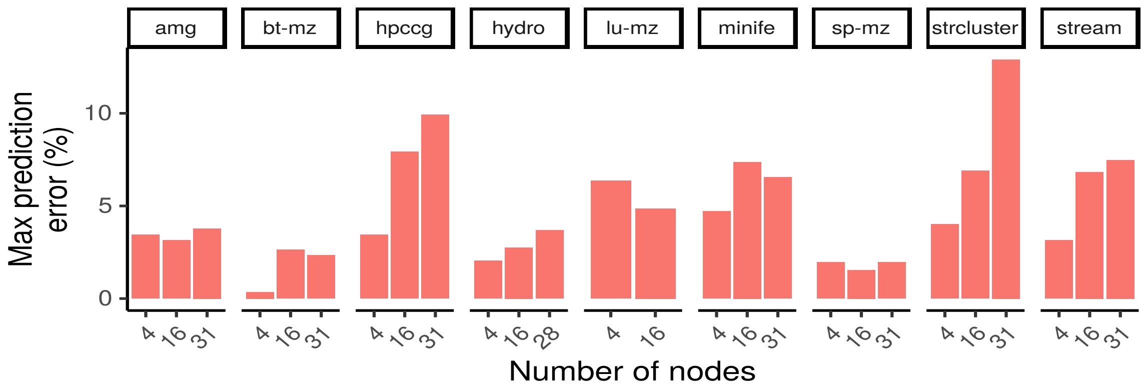


Figure 4.15 Slowdown model prediction maximum error using multi node approach.

As we intend to use the methodology for our analysis of resource management and job scheduling decisions, we cover some special cases regarding possible job placement and the kind of degradation it might suffer. We underline one special case: when the access to memory is fully remote. To predict the performance of the applications when memory access is fully remote, we created the sensitivity curve for remote access. In this case, our target application runs completely remotely, while the interfering runs locally. To simplify the costs of benchmarking, we collected the performance under 100% local interference and interpolated the points between the collected point and the remote performance alone (0%, without interference).

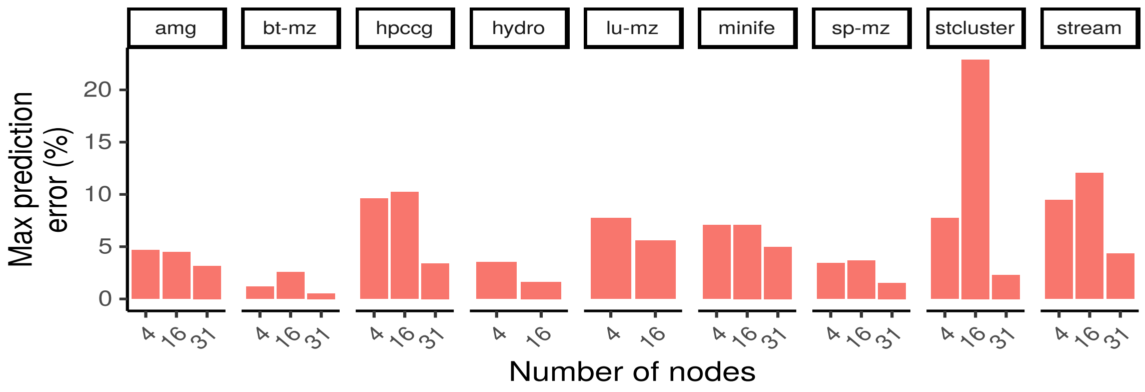


Figure 4.16 Slowdown model prediction maximum error for remote execution.

Figure 4.16 presents the maximum prediction error for the remote execution. We observe that the maximum errors are kept below 10% and for some applications, increasing the number of nodes does not generally increase the maximum prediction error. In Figures 4.15 and 4.16, the *streamcluster* application experiences a high prediction error. This application has high variance in its runtime even without contention. Consequently, the prediction model for this application is more challenging to build.

4.6 Conclusion

In this Chapter, we presented our Slowdown based methodology to build a contention model to predict the performance degradation that results from contention in remote memory access. We added to the methodology the concepts of smoothing and read/write memory access ratio to create the correct sensitivity curve, in order to increase the accuracy and similarity with real executions. Using the characterization of an application's sensitivity to contentious pressure from remote access to the memory subsystem, we were able to predict an application's performance in a pairwise execution with 1.19% prediction error on average and 14.6% in the worst case. Compared with the state–of–the–art, the relative improvements are almost 24% on average and 33% for the worst case.

Interesting avenues to build on this work in the future are detailed in Section 9.1. We believe that the approach presented in this Chapter is of particular importance for novel and upcoming global shared memory architectures and to analyze future resource allocation decisions for such platforms. We will therefore use the proposed model in an at-scale evaluation of the proposed allocation and scheduling policies for disaggregated memories using the Slurm simulator.

Part III:

Allocation and Scheduling in Disaggregated Memory Systems

Boi com sede bebe lama
Barriga seca não dá sono
Eu não sou dono do mundo
Mas tenho culpa, porque sou
Filho do dono

Petrucio Amorim

CHAPTER 5

Infrastructure and Experimental Methodology

In this Chapter, we introduce the methodology used in this work to evaluate the implemented disaggregated approach and its at-scale evaluations analyzed in this work. The content of this Chapter is used in our papers [40–42]. In the following sections we will detail the methodology used to generate the workload applied in our simulations. All simulations were carried out using the environment setup and benchmarks detailed in Section 4.1 and the developed contention model presented in Chapter 4. In Section 5.1 we detail the simulated systems applied in this work. Section 5.2.1 describes the methodology to generate the synthetic workload used during the evaluation of the disaggregated approach presented in Chapters 6 and 7. Sections 5.2.2 and 5.2.3 detail the methodology used to generate the synthetic and real workloads used to analyze the dynamic nature of memory usage in Chapter 8.

The methodology applied to Chapters 6 and 7 uses synthetic workloads in which the memory requested is the peak usage throughout the execution of the job. The metric is required as the workload model generator employed to create the input data does not model the dynamic aspect of the memory usage of the jobs. Therefore, it is sensible to assume the requested memory of a job is an estimation of its peak usage. Then in Chapter 7, we analyze the implications of accurate estimation of peak memory usage to the system's performance. Additionally, enhancing the finds of Chapter 7, the methodology applied to the Chapter 8 uses two different datasets for which we have the profile of memory usage. As a result, we can contrast the system's performance considering static and dynamic memory allocation.

5.1 Simulated System Configurations

We set up different configurations to explore heterogeneity in job demands and node capacities. The details of these experiments are given in Table 5.1. In Chapters 6 and 7, our HPC system has *normal* nodes, which have typical memory capacity, and *large* nodes with twice the memory capacity of the *normal* nodes. To evaluate different scenarios, we experiment with multiple ratios between large and normal nodes, varying from 0% (all normal nodes) to 100% (all large nodes). We similarly define a job to be *large* if it requires a large capacity node to run with the *Baseline* policy (without disaggregation). A job is *normal* if it can execute on a normal capacity node. The systems have a total of 1024 nodes each having 32 cores and 32 GB or 64 GB of memory, Slurm is configured to use its *Baseline* or our *Disaggregated* select resource policy. The parameters for job scheduling are the same for all experiments. The input data is generated using the methodology detailed in Section 5.2.1.

Table 5.1 Slurm configurations used for our simulations in Chapters 6 and 7

Configuration parameter	**Value(s)**
System size	1024 nodes
Number cores per node	32
Memory per node	32 GB, 64 GB
Allocation policy	Baseline, Disaggregated
Scheduling policy	Backfill
Queue and Backfill size	100
Backfill and Scheduling interval	30 s
Heterogeneous system ratio: % Large nodes	0, 15, 25, 50, 75, 100

For the simulations and analysis carried out in Chapter 8, we changed the system configuration to a more realistic setup based on the infrastructure in which the memory usage trace was collected. Table 5.2 present the details of the HPC systems considered in our simulations. For both the synthetic workload detailed in Section 5.2.2 and the real workload detailed in Section 5.2.3, we use systems with a total of 1024 and 1490 nodes respectively. The latter matches the system from which the dataset was collected. We use *normal* nodes with 64 GB and *large* nodes with 128 GB. We further underprovisioned the systems by adding *extra small* nodes with 32 GB. Thus, we experiment with multiple ratios between large, normal, and extra small nodes. However, for each system, the largest memory node has either 128 GB or 64 GB memory. Slurm is configured to use its *Baseline* or our *Disaggregated* select resource policy, with the

Disaggregated approach allocating memory statically or dynamically (see Chapter 8 for more details).

Table 5.2 Simulated system configurations for Chapter 8

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† [137]	
Cost per 128 GB	\$1280 [137]	

† 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. For the cost–benefit analysis, detailed in Section 8.4.3, we use the costs given in Table 5.2, which were taken from a recent analysis of a small-scale HPC cloud platform [137]. It is known that the network topology connecting the compute nodes also affects price and performance, even though it is not mentioned in the analysis, we adopted a torus topology, sized as recommended by prior work [138, 139]. Torus networks are commonly used in large scale supercomputers and they have a lower cost compared to other popular alternatives such as fat-tree typologies [139].

5.2 Workload Methodology

None of the traces described in Section 2.3 provide all of the information that is required for our analysis. We used three sources of job traces, which are summarized in Table 5.3. The first is based on the CIRNE model. The second uses the Google trace shaped by HPC job statistics from CIRNE and Archer [140]. The third is based on LANL's Grizzly trace, augmented by the job submission times from the CIRNE model. All traces are augmented to use the Slowdown model detailed in Chapter 4.

Table 5.3 Summary of information provided by the job traces

Trace	Domain	Submission times	Memory request	Number of nodes	Job duration	Slowdown model	Memory trace
CIRNE [70]	HPC	✓	×	✓	✓	✓	×
Google [74]	Cloud	×	×1	✓	✓	×	✓2
Grizzly [56, 76]	HPC	×	×	✓	✓	×	✓

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).

5.2.1 Synthetic CIRNE Model

We generated synthetic workloads using the CIRNE Comprehensive Model [70] (details see Section 2.3.2). However, the workload trace generated using the model does not provide memory information about the memory capacity requested by each job. As an important aspect necessary to our evaluation, we augmented the set of generated traces with scaled memory information from the applications profiled in our real environment. The step–by–step process to augment the synthetic trace is depicted in Figure 5.1.

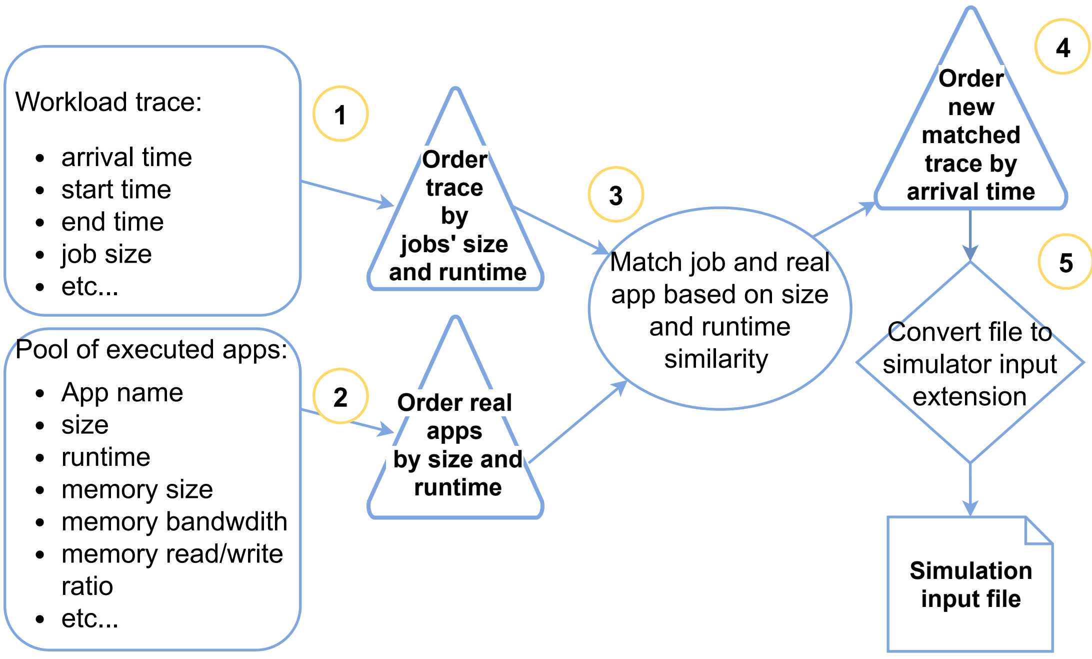

Workload trace:
• arrival time
• start time
• end time
• job size
• etc...
Pool of executed apps:
• App name
• size
• runtime
• memory size
• memory bandwdith
• memory read/write ratio
• etc...
1
Order trace by jobs' size and runtime
2
Order real apps by size and runtime
3
Match job and real app based on size and runtime similarity
4
Order new matched trace by arrival time
5
Convert file to simulator input extension
Simulation input file

Figure 5.1 Augmenting the workload trace with real application data. Methodology adapted from [25].

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 the Euclidean distances to map each real application to a similar synthetic job based on its size and runtime (Step 3). In the case of having different applications with the same size and runtime, several strategies can be applied to select the mapped pair of jobs and applications. In our methodology, we pick the first pair of the mapped job–application from our output to represent the job in our trace and ensure the process is reproducible when recreating the traces.

Finally, we generate the new augmented trace by the assigned arrival time (Step 4) and convert it to a binary readable by the simulator (Step 5). In 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 contention model (Section 4.3.2) 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 listed in Table 5.1. This way we match the trace profile in the number of jobs issued to the large capacity nodes to the ratio of nodes in the system, thereby allowing a more comprehensive analysis of the system when it runs balanced or an imbalanced workload mix. We ensure that all traces have total *node–hours* (#nodes × runtime) of large and normal jobs in the indicated ratio. The characteristics of the large and normal jobs are given in Table 5.4.

Table 5.4 Large and normal job characteristics for CIRNE generated trace

	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

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. This allows us to expose the effects of the strong scaling in the system since HPC users are often driven by time to solution, therefore memory underutilization is common as jobs of good scalability are distributed over various nodes to accelerate its execution.

5.2.2 Synthetic Model plus Google Trace

In order to use the Google trace (detailed in Section 2.3.3) and draw meaningful conclusions, we performed some adaptations before adding it to our methodology.

Adapting Google trace — In this work, we are only interested in the job request type, as allocation requests would not provide the exact starting and finishing times of the applications running on it. Since it is composed of several traces for each cluster, we chose the trace having the largest proportion of best-effort batch jobs (trace data from Cell *b* according to [74]), which are low priority jobs handled by the batch scheduler. To use the job's records that resemble HPC jobs the most, we filtered the trace using the job's priority and scheduling class parameters that define them as latency insensitive batch jobs. These kinds of jobs approximate from batch jobs usually submitted to HPC clusters as they aim to finish as quickly as possible.

However, batch jobs in their cluster typically have no strict completion deadlines. They are evicted and killed to free resources for high-priority jobs. We filtered the jobs that finished normally at least once, and we denormalized the memory usage per node to match our trace. Since the Google trace does not have the value for the highest memory capacity in the system, we used the max value of 12 TB to denormalize the data as it was reportedly used in their data centers in the same year of the trace's release [141].

As mentioned in Section 2.3.3, the trace reports usage as the average and maximum usage during a series of 5 min measurements. We use the maximum used memory to define the usage for the period between two measurements. We correlate the simulated job's progress and the usage record, calculating which progress each measurement should represent. We divide the number of records by the task runtime. Then, we use the cumulative sum of the result to define which progress the record represents.

Generating the input files — In order to use the Google usage trace we also generated the input job trace files using the CIRNE Comprehensive Model [70]. However, differently from the previous methodology presented in Section 5.2.1, we augmented it to cover two main aspects for our dynamic memory evaluation. The first one is to correlate the generated file with the Google usage trace, and the second aspect is the specification of the job's memory request, which is not modeled by the CIRNE model. The complete methodology is presented in the Figure 5.2.

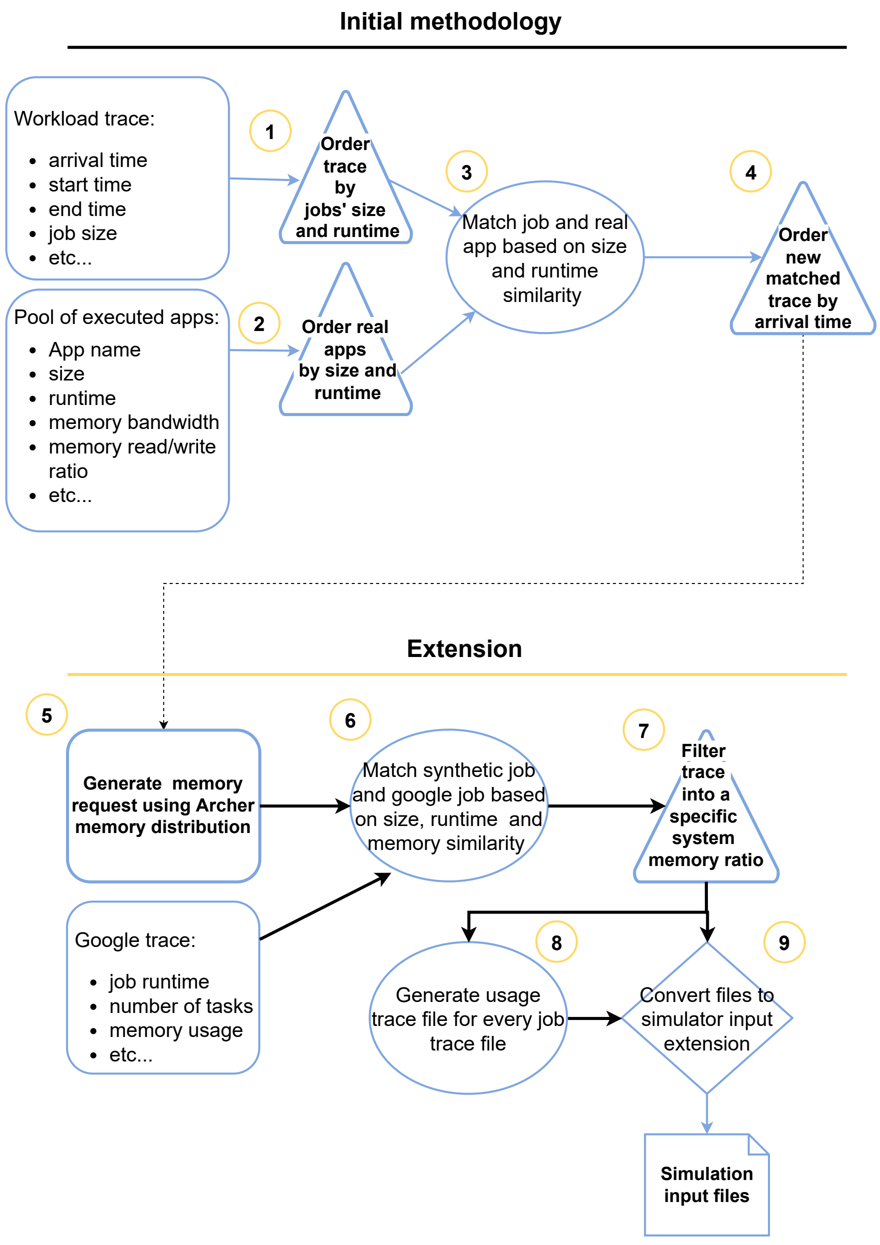

Initial methodology
Workload trace:
arrival time
start time
end time
job size
etc...
Pool of executed apps:
App name
size
runtime
memory bandwidth
memory read/write ratio
etc...
1
Order trace by jobs' size and runtime
2
Order real apps by size and runtime
3
Match job and real app based on size and runtime similarity
4
Order new matched trace by arrival time
Extension
5
Generate memory request using Archer memory distribution
Google trace:
job runtime
number of tasks
memory usage
etc...
6
Match synthetic job and google job based on size, runtime and memory similarity
7
Filter trace into a specific system memory ratio
8
Generate usage trace file for every job trace file
9
Convert files to simulator input extension
Simulation input files

Figure 5.2 Extended methodology to augment workload trace with real application data and per-job memory usage from the Google trace. Methodology adapted from [25].

We followed the initial steps of the previous methodology by first, generating the synthetic trace using the CIRNE Model (Step 1) for the simulated system size. We then use a pool of previously executed applications for which we have a profile regarding size, runtime, memory bandwidth, read/write ratio, and local/remote access memory ratio (Step 2). To correlate both data, we calculate the Euclidean distance to map each real application to a similar synthetic job based on its size and runtime using the trace and app list (Step 3). Following, we order the generated file by its arrival time (Step 4). These four steps comprise the main initial methodology applied to generate the trace files used in our initial evaluation.

The initial methodology is augmented with the steps detailed in the *Extension* presented in the Figure 5.2. Previously, we scaled the memory used by the correlated application in our contention model as a proxy for its job memory request. For this evaluation, we generate the memory request following the memory distribution presented in [140]. It presents the actual memory per node demands of the most used applications on a contemporary HPC supercomputer. The distribution is displayed in Table 5.5. For each job, we use its size to generate the memory request (Step 5).

Table 5.5 % of maximum memory usage per node for all jobs. Table adapted from [140]. (Small: ≤ 32 nodes; Large: > 32 nodes)

Max memory use		Usage	
(GB/node)	All	Small	Large
(0,12)	61.0%	69.5%	53.0%
[12,24)	18.6%	19.4%	16.9%
[24,48)	11.5%	7.7%	14.8%
[48,96)	6.9%	3.0%	11.2%
[96,128)	2.0%	0.4%	4.2%

Once we have the memory demand for each job, we calculate the Euclidean distance once more, but this time to map each job and its new memory capacity to a Google job (see the beginning of the Section), to create its usage trace profile (Step 6). The usage trace profile is an additional trace file with several records having the *jobid*, *node*, *max memory usage* for the period, and the *application's progress*, which represents the starting point of the period. To keep the simulation time under control, for each job and node, we filter the number of records using the Ramer–Douglas–Peucker (RDP) [142, 143] algorithm. It is a method that resamples a curve to a similar curve with fewer points.

Table 5.6 presents the characteristics of the defined large and normal jobs. The memory demand of normal jobs is less than the capacity of a normal node (see Section 5.1 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 are presented in Figure 5.3. 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, we take a conservative approach in our study having similar distribution for the maximum usage and requested memory. As mentioned in [36, 37, 2], users are implicitly encouraged to overestimate their resource request, to avoid having the job killed due to insufficient requested resources.

Table 5.6 Large and normal job characteristics for Google plus synthetic trace

	Normal Jobs		**Large 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

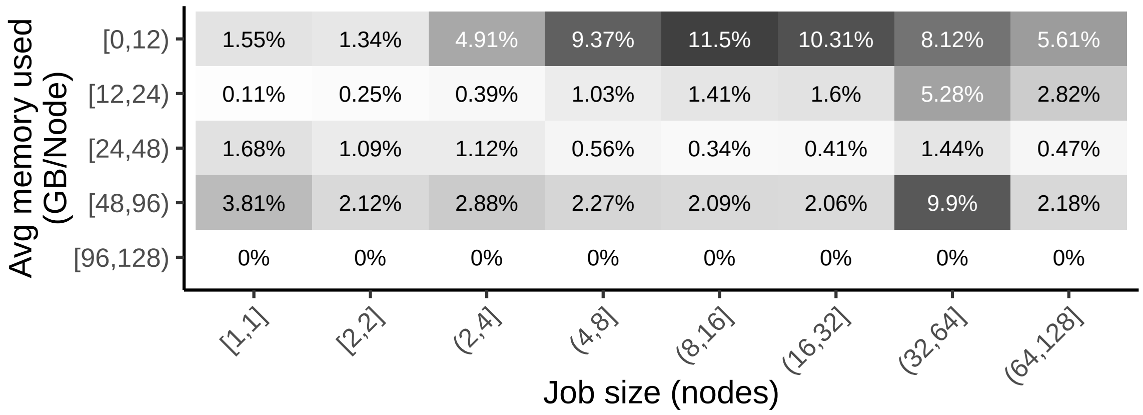
We sample the job trace into additional input files to target each of the specific heterogeneous system ratios listed in Table 5.2 (Step 7). For every trace, we keep the total *node–hours* (#nodes × runtime) of large and normal jobs in the indicated ratio. We also separate the usage trace profile for each additional file created in the previous step (Step 8). Finally, we convert the new augmented traces to a binary readable by the simulator (Step 9).

In the end, we have a new input job trace preserving the synthetic trace info, but including the memory capacity and an identifier for the job application. The memory will be used for resource scheduling, while the application identifier will be used by the contention model to calculate the slowdown suffered by this particular job due to resource sharing.

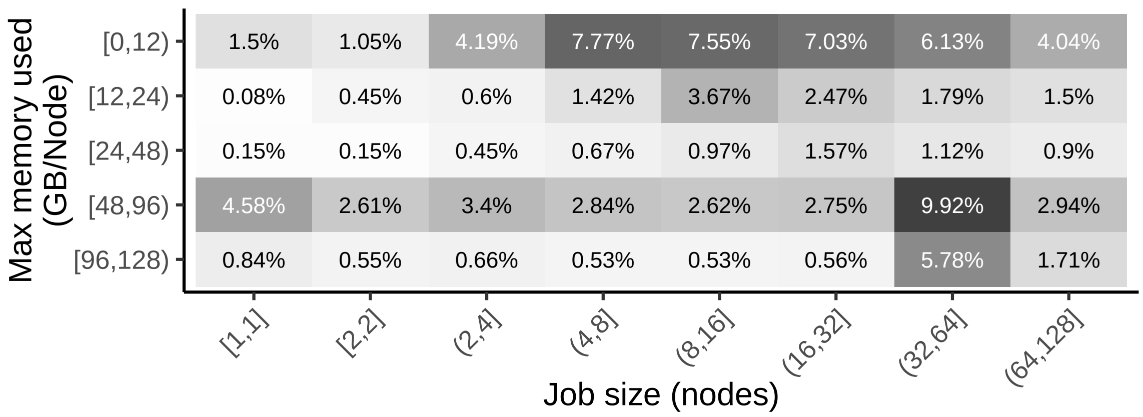
5.2.3 Adapting Grizzly Trace

The methodology applied to the Grizzly dataset is much simpler than the ones presented in the previous Sections. To be able to use and simulate the Grizzly dataset, we extracted some periods from the entire dataset. We separated the dataset into periods (a week) to group jobs into a reasonable set of jobs. Then, we calculated the utilization using the average *node–hours* usage to represent the average cluster node utilization before we start our simulations. To perform this calculation, we summed the *node–*

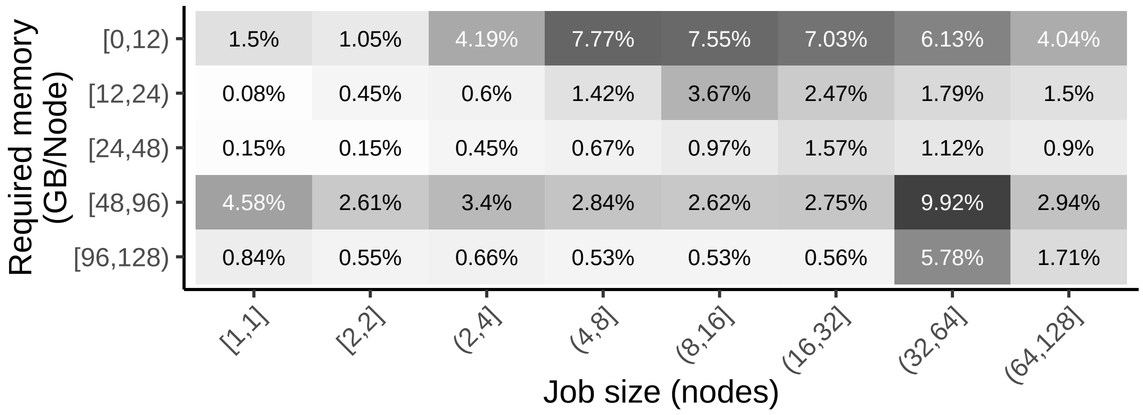
hours of all jobs in the period and divided it by the period's makespan (calculated using the start time of the first job and end time of the last job in that period).



(a) Average memory usage from usage trace profile.



(b) Maximum memory usage from usage trace profile.



(c) Requested memory from job trace.

Figure 5.3 Trace memory heatmap distribution versus job size for the synthetic model plus Google trace.

We sampled the Grizzly dataset (see Section 2.3.4), to obtain a smaller trace that was feasible to simulate. Figure 5.4 shows all the one-week periods in the Grizzly dataset, 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 gray dots. We took a random sampling of the weeks with a utilization of 70% or more, which is representative of HPC [56]. We then randomly chose seven periods to simulate. Figure 5.4 shows that the chosen periods are representative of the important periods during which utilization is relatively high.

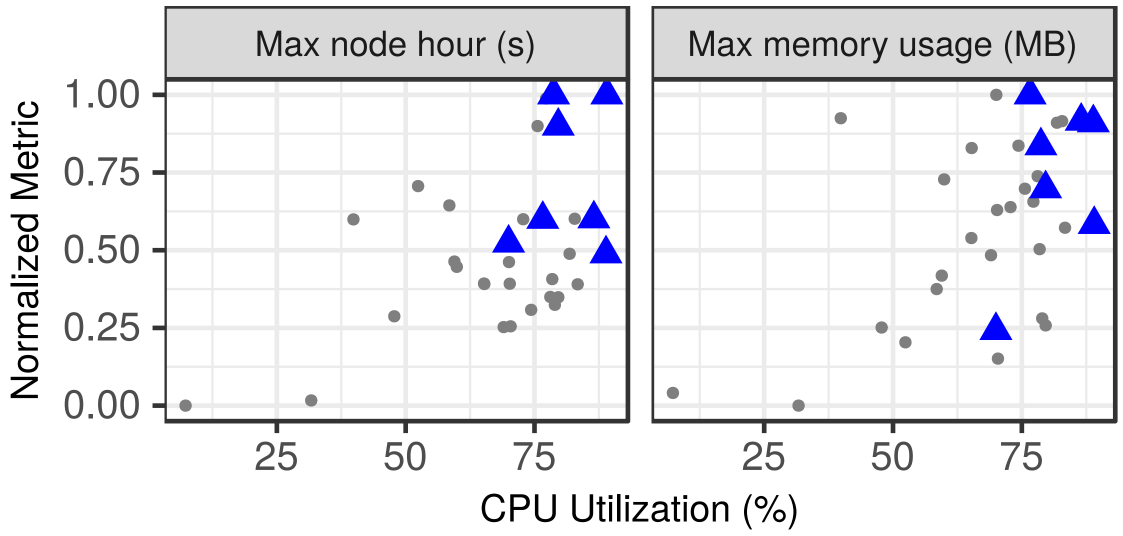


Figure 5.4 Sampling the Grizzly usage trace. Each point represents a 1-week period. Simulated periods, depicted as blue triangles, are representative of weeks with CPU utilization ≥ 70%.

We generated the input job traces using seven periods selected from the dataset. First, we separate from the jobs' specifications the metrics we could extract from them, e.g. runtime and the number of nodes. However, to simulate the traces we must create a SWF like trace style used by the simulator. Consequently, we augmented them to generate the missing fields required by our simulation methodology, e.g. *requested memory*, *submission time*, *application id*. For the requested memory metric, we used the maximum memory consumed by the job in any node. As this information is not present in the initial dataset, we intentionally used the maximum memory usage reported, because by doing so, we would be able to verify the significance of users' resource estimation on the system's performance.

Then, to finalize the setting up of the simulated job traces, we generated a submission distribution using the CIRNE Comprehensive Model [70] and we performed a Euclidean distance to map each job to a real application in our pool of previously executed applications. This step maps each real application (presented in Section 4.1) to a similar job in the selected period based on its size and runtime. This step is necessary in order to use our contention model during the simulation to estimate the effect of shared memory resources.

In the end, since the dataset is composed of several records of memory consumption for every job in each compute node (every ten seconds), we formatted the usage trace to our simulation methodology. We generated our usage trace profile for each selected period. The usage profile is an additional trace file with several records having the *jobid*, *node*, *memory usage*, and the *application's progress*, which defines the time window the recorded memory usage represents. To keep the simulation time under control, we apply the RDP [142, 143] algorithm for each pair of jobs and node to filter the number of records from the usage trace profile. The algorithm will resample it to a similar curve with fewer points.

CHAPTER 6

Extending Slurm Simulator for Disaggregated Memory

In the previous Chapters, we introduced basic concepts and materials used throughout this work. We also introduced a generic approach to estimate the performance degradation due to the sharing of disaggregated memory using a profiling technique. By contrast, this Chapter describes how we extended the Slurm resource manager to support disaggregated memory and how we integrated the developed contention model (presented in Chapter 4) in our simulated environment. The content presented in this Chapter can be found in our paper [40].

Over 90% of the 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 resource manager's goal is to satisfy users' demands for computation and achieve acceptable performance in the overall system utilization by efficiently matching requests to resources.

Nevertheless, the rigid boundaries between compute nodes limits compute and memory resource utilization in existing HPC systems. HPC applications are rarely co-located on a compute node [10], 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 [1] because HPC application memory demands vary dramatically, by orders of magnitude, due to application characteristics and strong scaling [6, 7].

Disaggregated memory has recently been proposed to allow a flexible and fine-grained allocation of memory capacity to compute jobs [14, 8, 10]. In this direction, we propose 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 the research level, and system software is immature. We, therefore, extend an existing simulation approach using Slurm to account for memory bandwidth contention in disaggregated memory leveraging the developed Slowdown based method presented in Chapter 4. We then use the extended Slurm simulator to determine the overall system throughput, job queuing, and execution time of a large-scale disaggregated HPC system.

A positive aspect brought by the disaggregated architectures is that requests from resource-hungry applications can be executed in such architecture consuming resources from other servers, while others can share memory to better exploit memory capacity and improve performance [26]. They can be submitted to a regular queue, avoiding the need to wait for large memory nodes[1] (typically a small number in the cluster), without having to overprovision resources [7].

In summary, the major contributions of this Chapter are:

① A Disaggregated-aware allocation policy implemented in the Slurm resource manager. The allocation policy allows nodes to use the memory capacity of a remote node when the memory demands of submitted jobs exceed the system node's local memory.

② We present an extended job scheduler simulator to support disaggregated memory on top of the Slurm resource and job management system. The simulator environment supports the evaluation of several adaptations to the resource manager and takes into account the heterogeneity among the resources.

③ An at-scale evaluation of our contention model and allocation policy for disaggregated memories using the simulation environment. 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 [144].

[1]On MareNostrum–4 at BSC large memory nodes represents only 6% of the total number of nodes.

6.1 Resource Allocation for Disaggregated Memory Systems

As presented in Section 2.1.1, one of the major characteristics that prevent the use of disaggregation by the Slurm resource manager is that it has a processing and server-based architecture. Memory management is tightly coupled with the 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 the 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.

We notice that modifying the resource selection and using remote memory across the cluster impacts applications. This means that validation through simulating the platform requires some degree of consideration for the application's performance running in such a configuration. To this end, in Section 6.2 we integrated the disaggregated strategies considered in this thesis and the developed contention model described in Chapter 4 to characterize the slowdown experienced by the applications sharing memory resources.

After our initial analysis of the job scheduling process depicted in Figure 2.3 and described in the Section 2.1.1, we modified the verification performed by the job scheduling process to build the list of nodes available to the job. It checks what kind of node configuration has enough memory capacity to satisfy the request before removing it from the selection. In the baseline system, if no nodes can satisfy the request then an error is raised, even though there is enough memory scattered throughout the system.

While the default allocation (Baseline) used by Slurm 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. Some of the strategies are described in the following Sections.

6.1.1 Job submission interface

In order to take advantage of disaggregated memory, the user must indicate the amount of memory required. It may not always be possible to obtain this information, although it must be pointed out that on existing systems users already need to have some idea of the memory demands in order to choose whether to submit the jobs to the normal or large memory queue. In the following Chapters, we will discuss the impact of the user's provided memory demands on the overall system's performance.

It is also worth pointing out that the user does not have to indicate the memory bandwidth to allocate disaggregated resources to the job in our system. During the allocation, the resource manager exclusively uses the requested memory, as we already mentioned that it is the prevalent method on existing systems. Notwithstanding, memory bandwidth is an important parameter used by our contention model during our simulations to correctly estimate the contention on shared resources and consequently the application's performance. The parameter is incorporated into our job traces as indicated in Section 5.2.

6.1.2 Supporting Memory Disaggregation

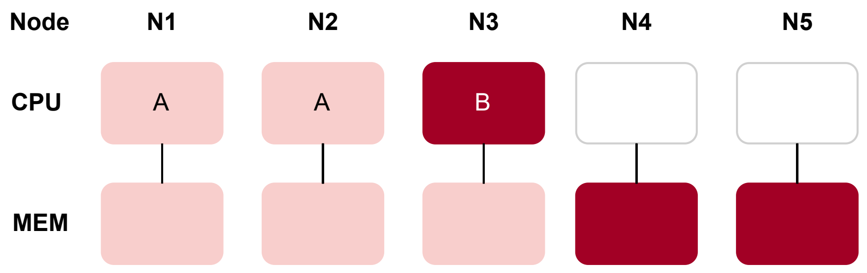
Figure 6.1 details some of the allocation strategies explored in this work. In this Figure the letters represent jobs and their order of arrival. Our basic implementation is the *Strawman* approach (Figure 6.1a), in which we consider every node with cores and memory available. In this approach, we first select the processing units. Later, we iterate over the nodes with memory available and accumulate it until the requested amount of memory for the job is satisfied. The downside of this approach is that it will generally assign mainly remote memory for the jobs, even though there is local memory available for other nodes. As it can be seen in Figure 6.1a, Job B using a single node allocates its whole memory requirement remotely because the local memory associated with the selected compute units in node N3 is already taken by Job A. It happens because in this approach both resources are treated completely independently without trying to allocate memory close to the processing unit. Consequently, it will compromise the jobs' performance, since it will lead to undesirable higher contentions and remote usage.

The following implementation is the *Naive* approach (Figure 6.1b). In this approach the resource allocation starts packing memory locally to the job, otherwise, it uses remote memory nodes. It tries to solve the problems arising from the *Strawman* approach by mainly accumulating memory close to the cores. However, two major issues arise from this approach. The first one is that since it accumulates the memory locally, jobs with low memory requirements and jobs that do not fully utilize a local memory node will have their memory packed on a few nodes which will decrease the local memory ratio. In the Figure, Job C uses two processes (running on node N1 and N2) and its memory requirement can fit on a single node. Consequently, the approach will allocate the memory locally for the process running on node N1 and remotely for the process running on node N2. The second problem using the *Naive* approach is that it might use nodes with a small amount of available local memory that will lead to unnecessary high usage of remote memory. In this case, Job G is allocated to node N5 which has low local memory available, the remaining capacity is allocated remotely on node N2.

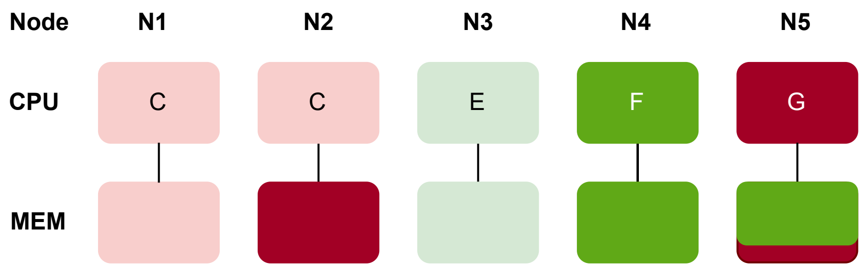
To cope with the problems of the previous approaches we analyzed the *Local* approach (Figure 6.1c). The idea behind this approach is that it only allocates nodes that meet a defined threshold of local memory available. If the local memory is below the defined threshold, the node is used only as a memory node for other jobs, which means that the compute units associated with the node are not allocated to any incoming job. In this case, node N4 has CPU and memory available, however, it effectively becomes a memory node as the memory available is too low to receive new requests (most of its capacity has been taken by another job). As a consequence, the next job to start, Job K, allocates node N5 which satisfies its CPU and memory requirements. An issue with this implementation is that for resource hungry jobs the allocation accumulates the total required remote memory for the whole job. Consequently, we consider the job will have several of its nodes contending for the same memory node due to local/remote access.

We further improved *Local* approach implementing the *Local spread* approach (Figure 6.1d). For this approach every compute node receives a specific memory node based on its remote demand, hence eliminating the contention the job would have had accessing the same memory node. In this case, for Job L running on nodes N1 and N2, it will receive memory capacity from node N3 for the process running on node N1 and memory from node N4 for the process running on node N2. This approach prioritizes the usage of local memory while spreading the remote memory. For resource hungry jobs, remote access will dominate the contention with other jobs when accessing the remote memory nodes.

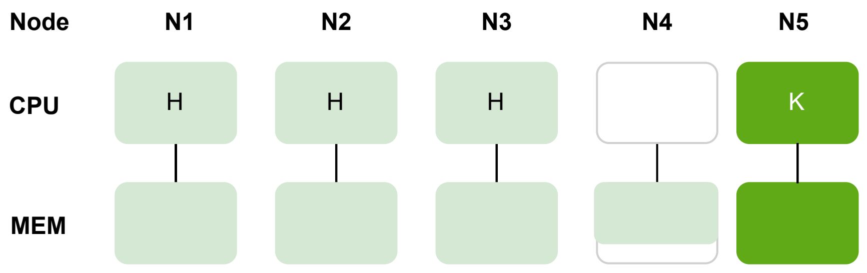
In spite of prioritizing nodes with higher local memory available to decrease the remote memory usage and increase performance, a common issue to the *Local*, *Local spread*, and similar approaches is that some nodes will be used exclusively as memory nodes. Their memory threshold constraint can lead to a waste of CPU across the system, which can decrease overall efficiency and increase the cost. Nevertheless, we argue that this might happen when memory is the bottleneck and the most requested resource in the system. In addition, previous studies have shown that memory is the most underutilized resource, and therefore the overall performance will benefit from better memory usage.



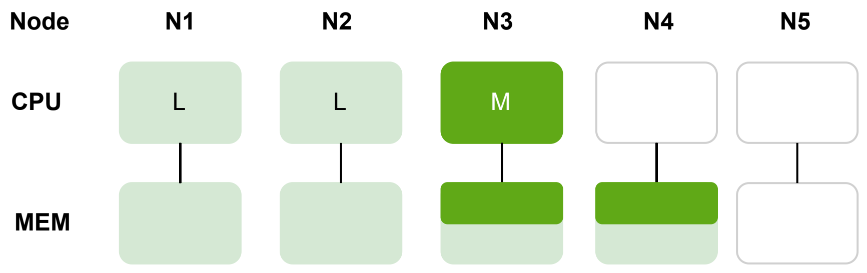
(a) *Strawman.*



(b) *Naive.*



(c) *Local.*



(d) *Local spread.*

Figure 6.1 Graphical schemes of some memory allocations explored in this work for a few simple cases. The letters represent the job's order of arrival and each color identifies different jobs and their respective allocated memory. Red boxes represent allocations that incur unnecessary remote usage or contention. Green boxes depict adequate allocation and white boxes represent resources available.

6.1.3 Disaggregated Allocation Policy

Table 6.1 summarizes the allocation strategies considered in this work and detailed in the previous Section. It also introduces the strategy employed for our results and is detailed in this Section. The majority of the strategies had poor performance compared to the baseline (see Section 6.3.5). They failed because they either generally assigned mainly remote memory for the jobs (even in the presence of available local memory), or because they used nodes with low local memory which contributed to increasing the remote memory usage (decreasing local to remote ratio).

Table 6.1 Summary of disaggregated strategies considered in this work

Approach	Summary	Advantage	Constraint
Strawman	First it selects the processing units. Then, accumulates memory capacity to satisfy the request.	Uses fewer resources.	High usage of remote memory.
Naive	Accumulates memory close to cores.	Increase local bandwidth usage compared to the *Strawman*.	Uses compute units with low local memory available. High usage of remote memory.
Local	Allocates nodes with a minimum amount of local memory defined by a threshold.	Avoid allocating nodes with low memory capacity available.	Nodes of jobs may contend for the same memory node. May waste resources as nodes can be used solely as memory nodes.
Local spread	Every compute node receives an exclusive memory node.	Each node of a job does not share bandwidth from remote memory nodes.	Distinct memory features lead to unnecessary remote memory usage. May waste resources as nodes can be used solely as memory nodes.

(*table continues*)

Table 6.1 Continued: Summary of disaggregated strategies considered in this work

Approach	Summary	Advantage	Constraint
Chosen	Uses baseline allocation before the disaggregation.	Leverages distinct memory configurations to reduce the need for remote memory usage.	May waste resources as nodes can be used solely as memory nodes.

Even though the *Local spread* approach tries to increase the local-to-remote ratio, we notice two main issues when prioritizing the usage of local memory while spreading the remote memory. The first issue is that the allocation will favor nodes that already have memory remotely allocated instead of using nodes with free memory, consequently, the job will have its execution slowed down due to unnecessary use of remote memory. The second issue of this approach is that for resource hungry jobs in a heterogeneous environment, it does not differentiate nodes with distinct memory features. Therefore, the jobs can use nodes with much less total memory, and then to fulfill the requirement it will use more remote memory. In this case, more remote access will slowdown the jobs' execution.

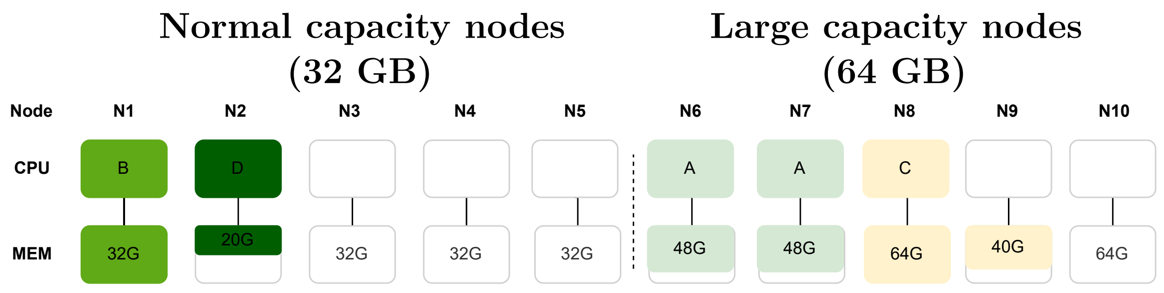


Figure 6.2 Graphical scheme of the memory allocation explored in this work for a simple case considering a system with half of its nodes having 32 GB (5 nodes on the left) and 64 GB of memory (5 nodes on the right).

Figure 6.2 presents a schematic simple case to exemplify the best allocation strategy explored in this work. This Figure shows a 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 above 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 Figure 6.2, jobs *A, B*, and *D* uses 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 is able to provide. To serve this request we use our disaggregated approach employing remote memory.

To improve performance, we do not use the CPU cores of nodes that have already lent memory to another node (Section 6.3.4 evaluates the effect of relaxing this requirement). This means that such a node effectively becomes a memory node for other jobs. 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 available memory. Then, nodes with higher local memory available are selected, consequently decreasing the influence of remote memory access. Instead of using *normal* nodes to satisfy the job's *C* request, the approach uses *large* nodes, thus increasing the local memory usage. For a newly submitted job, node *N9* might be used exclusively as a memory node since only 37,5% of its local memory is available.

6.2 Contention Model and Disaggregated Integration into Slurm Simulator

Figure 6.3 shows our disaggregated memory approach and contention model integration into the BSC Slurm simulator. Our disaggregated memory approach modifications are depicted as red boxes, while the contention model integration is represented as dashed red boxes. As detailed in previous Sections, our disaggregation approach modifies the resource manager scheduling process by expanding the baseline resource allocation with remote memory usage. All the modifications are done on the controller side which performs all allocation and scheduling operations. However, since the simulator *Slurmd* daemon is a simplification of the original source code, the contention model is responsible to support the update of the simulated runtime.

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.

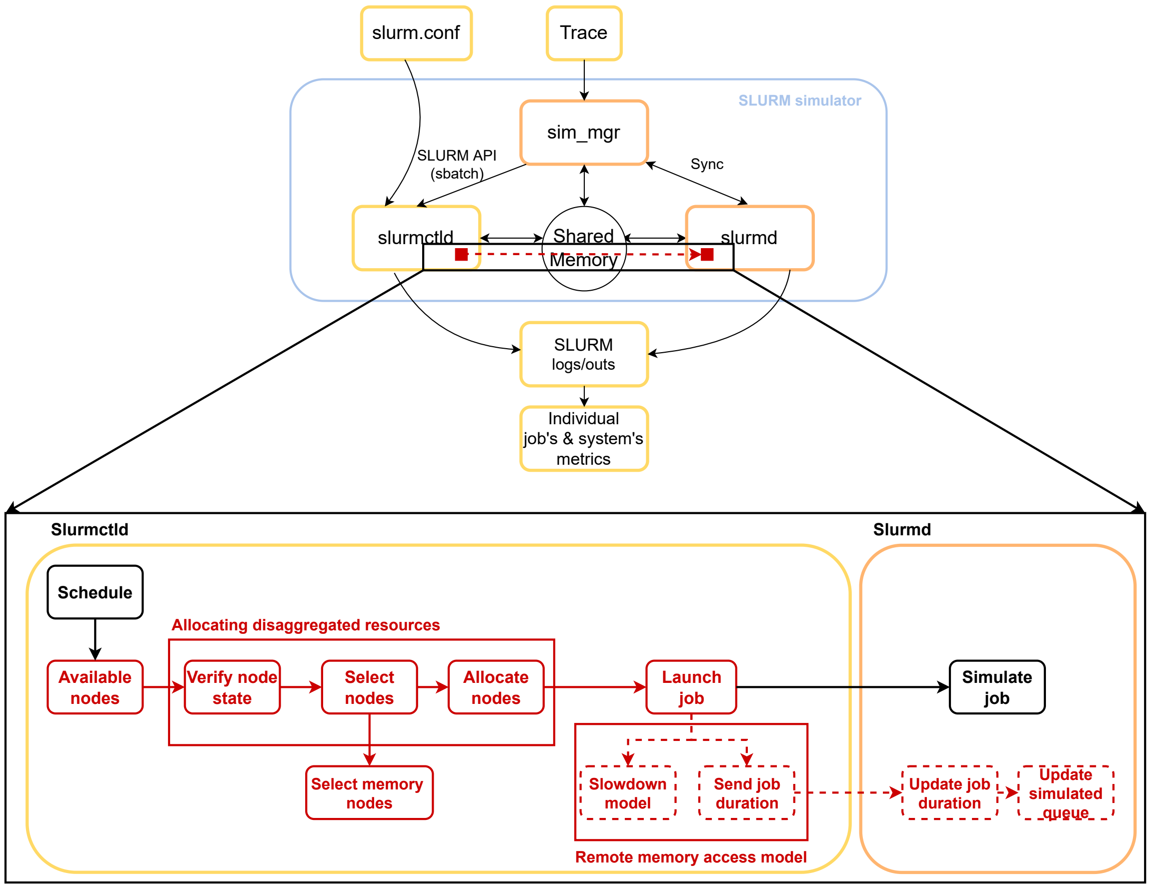


Figure 6.3 Developed disaggregated memory scheduling and contention model integration into BSC Slurm simulator.

We modified the trace format to also provide the information needed by the memory access contention model (see Section 5.2.1 and Chapter 4). 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. Using the application type identifier we are able to access its bandwidth (contentiousness) and the local-to-remote access memory ratio (penalty accessing remote memory). As pointed out in Section 6.1.1, this information is only used by the contention model to correctly estimate the contention on shared resources. The allocation policy does not need this information and therefore no extra profiling is required to allocate resources.

We modified the simulator to invoke the contention model each time any job starts (Launch job function). The contention model calculates the estimated performance of every job that potentially has a contention with the starting job. It will also account for the slowdown the jobs suffer accessing remote memory in case of having no contention with another running 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 or uses entirely local memory. 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 6.1. It is crucial to emphasize that the contention model is also invoked upon the completion of each job. This step is necessary to update the execution speed of any simulated jobs that experience contention with the finishing job. By eliminating the contention caused by the finishing job on the shared resources, we ensure a more accurate representation of the execution of the remaining jobs.

$$runtime\_left' = runtime\_left - delta\_time \times P_{\text{est}} \tag{6.1}$$

As pointed out in Section 4.3.3, a drawback of using the contention model in our experiments is that it considers only a single latency to model the access and contention on remote memory usage across the cluster. We know that the farther the nodes are from each other the higher will be the latency accessing the memory. Even though we assume a single latency in our simulated system, the effect of this assumption on performance in our results is further analyzed in Section 6.3.7.

6.3 Evaluation

The details of our simulated environment, configuration, methodology used for input generation, and its distribution applied in our experiments are listed in Chapter 5.

6.3.1 System Throughput

Leveraging the contention model and the Slurm simulator, we evaluate the implemented disaggregated infrastructure described in Section 6.1 simulating the different heterogeneous scenarios presented in Section 5.2. In this step, we assumed that the scalability of our memory access contention model has the same behavior presented in Section 4.5 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. Figure 6.4 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 this system has enough resources to execute any job across all inputs.

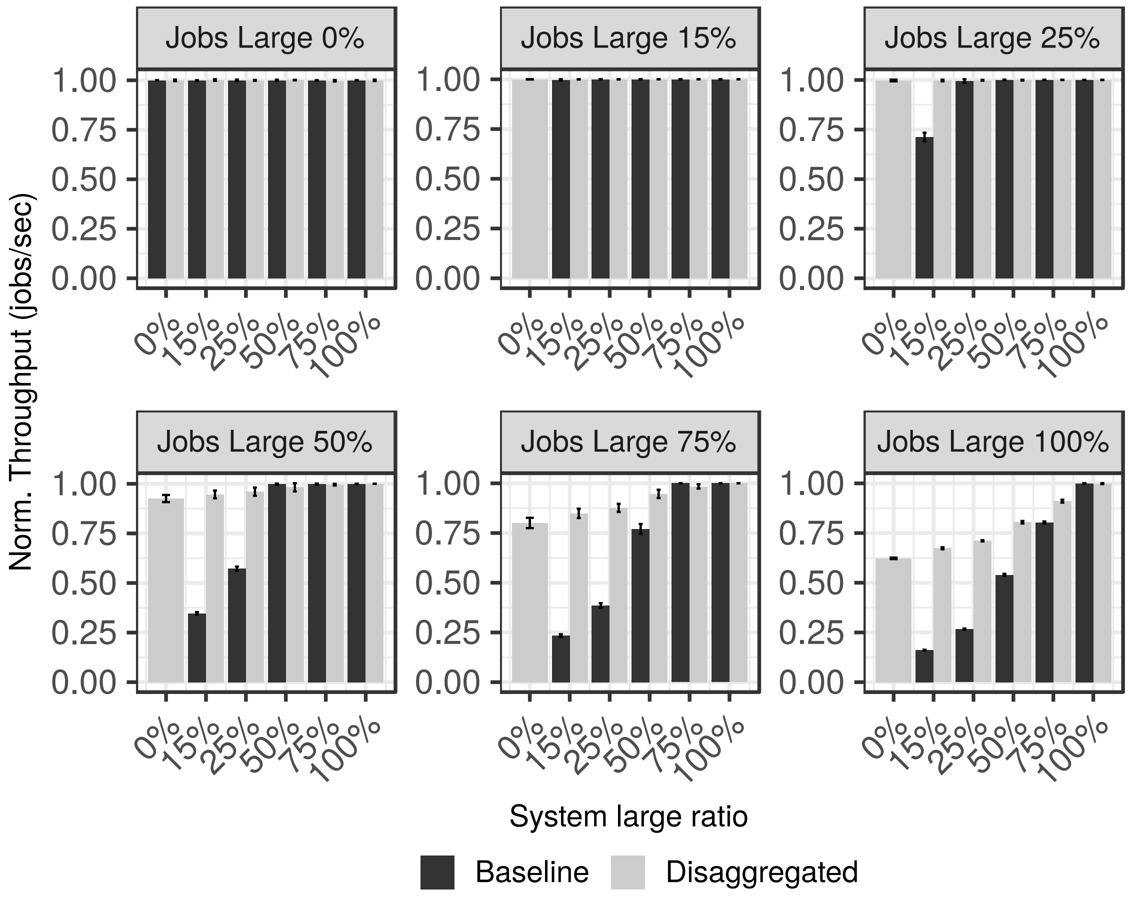


Figure 6.4 Normalized throughput (y-axis) experienced by each simulated system (x-axis) for various job mixes. Missing bars in the plots indicate there are not enough large memory nodes to run all the jobs.

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 Figure 6.4 has double bars showing the comparison between the baseline and disaggregated approaches except for the 0% system for job mixes with large jobs.

Figure 6.4 shows a clear trend in the system's throughput based on the resource 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 a 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 is able to run the large jobs, thereby the jobs wait longer to have access to the resources needed leaving aside other nodes. It contributes to increasing the makespan and therefore to 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 are not possible using the baseline approach.

The memory savings provided by the disaggregated approach are noticeable. For 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 (Figure 6.4). 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 memory savings are 14%, 20%, and 15%.

6.3.2 System Response Time

Figure 6.5 shows the cumulative distribution of the response time (defined by the waiting time plus execution 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 (middle panels) or run on an overprovisioned system (left-hand side panels), the approach's lines overlap showing similar performance. On the other hand, when the job mix stresses more resources on an underprovisioned system (right-hand side panels), 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 their submission. The impact of decreasing resources is less noticeable with our approach as it presents a 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 of some jobs.

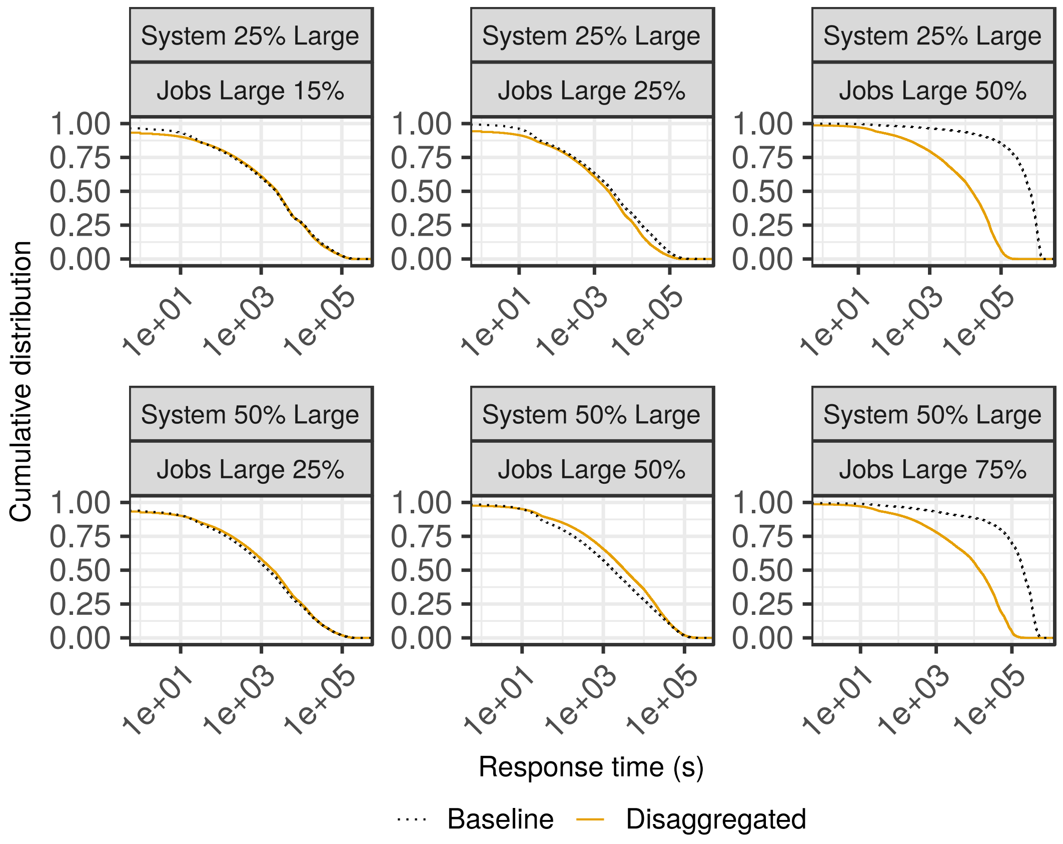


Figure 6.5 Cumulative distribution of response time for two different systems and different job mixes.

Figure 6.6 presents the distribution of response time across the top three scenarios of Figure 6.5 but divides them into large and normal jobs. We only show three scenarios for brevity, as we have seen that the other scenarios presented similar performance. We can notice that for the scenarios in which there was no benefit from disaggregated memory (left-hand side and middle panels), the baseline and disaggregated approaches have similar response time distributions. However, in the scenario for which there is a large reduction in response time (right-hand side panel), the large and normal jobs benefit roughly equally.

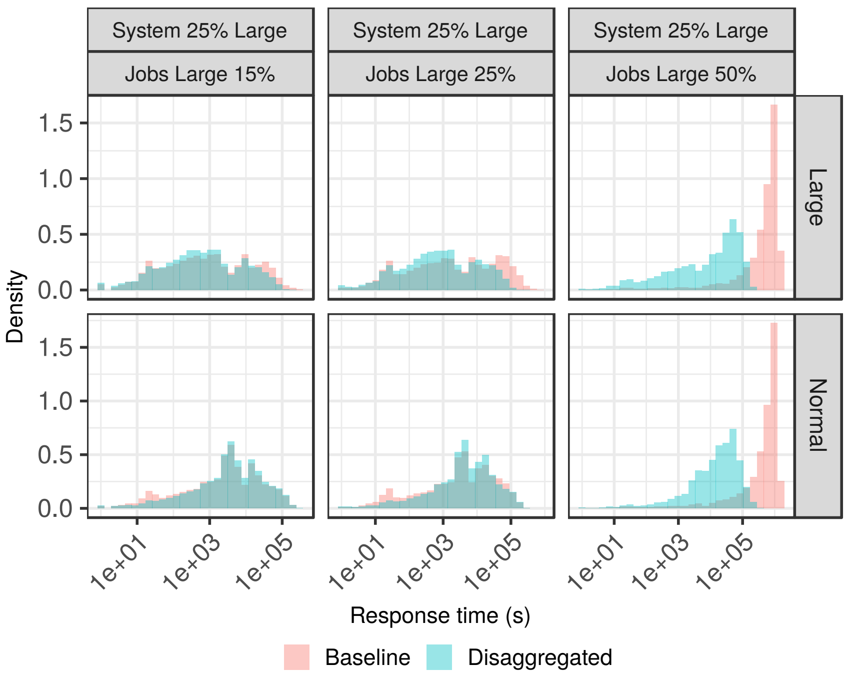


Figure 6.6 Response time distribution for large and normal jobs.

6.3.3 CPU and Memory System Utilization

Figure 6.7 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 fewer large nodes than available), *match* (job mix demand equals the number of large nodes), and *underprovisioned* (demands more). We 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.

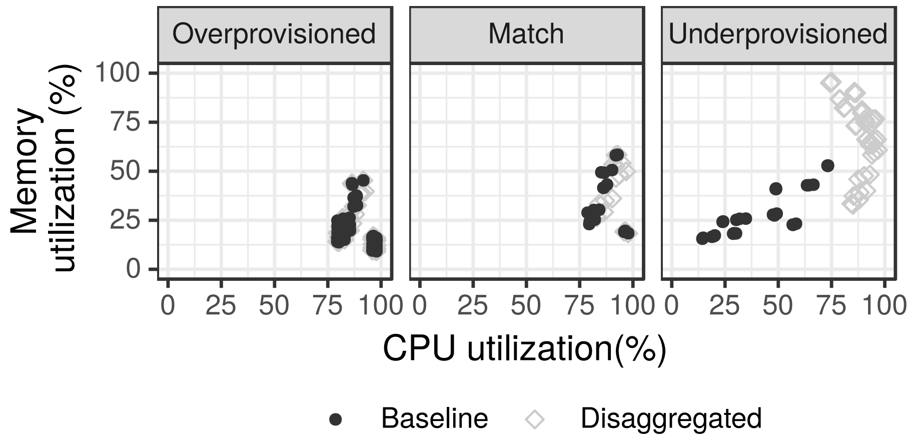


Figure 6.7 Average memory and CPU utilization when using either Disaggregation or Baseline under different job/node demands/capacities.

6.3.4 Varying Local Memory Threshold

To understand the effects of applying a threshold over the node's available memory before considering the selection of the compute nodes, we run several experiments varying this parameter. The Figure 6.8 presents the performance of the disaggregated approach when varying the local memory allocation threshold. We notice a slight decrease in the throughput when we decrease the local memory threshold. Even though we allow nodes with low free local memory capacity to take part in the compute node selection, to improve the allocation process we apply a weight to ranking the nodes based on their free memory capacity available before the actual node selection, which decreases the influence of the threshold parameter.

6.3.5 Different Memory Allocation Designs

In this section we present the results of running the different designs implemented in this work and detailed in Section 6.1.2, namely *Strawman*, *Naive*, *Local*, *Local spread* and the chosen *Disaggregated* approach (defined in Section 6.1.3 and used in all results in this Chapter). Figure 6.9 presents the normalized throughput for every design on a few representative scenarios. Overall, the tested designs presented lower throughput than the chosen disaggregated strategy. *Strawman* and the *Naive* approaches presented

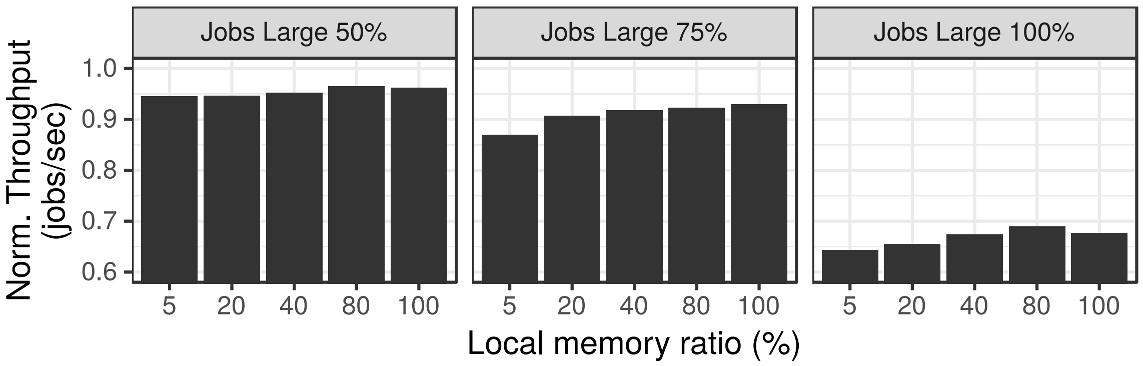


Figure 6.8 Effect of local memory threshold on throughput.

the lowest throughput among them in every scenario. It happens as both approaches allocate more remote memory even though there is local memory available to the compute nodes, thus increasing the remote-to-local ratio and exposing the application to unnecessary slowdown. *Local* and *Local spread* presented an improvement over the other two approaches with the latter having close results to our developed method on the underprovisioned scenario since they share a similar remote memory allocation policy. On the other hand, for overprovisioned scenarios, they decreased the throughput, as they disaggregated resources even though there are nodes with enough local capacity to satisfy the job's memory request. In opposition, our chosen *Disaggregated* approach applied in our simulations only resorts to disaggregation when there are insufficient resources, thus decreasing job degradation.

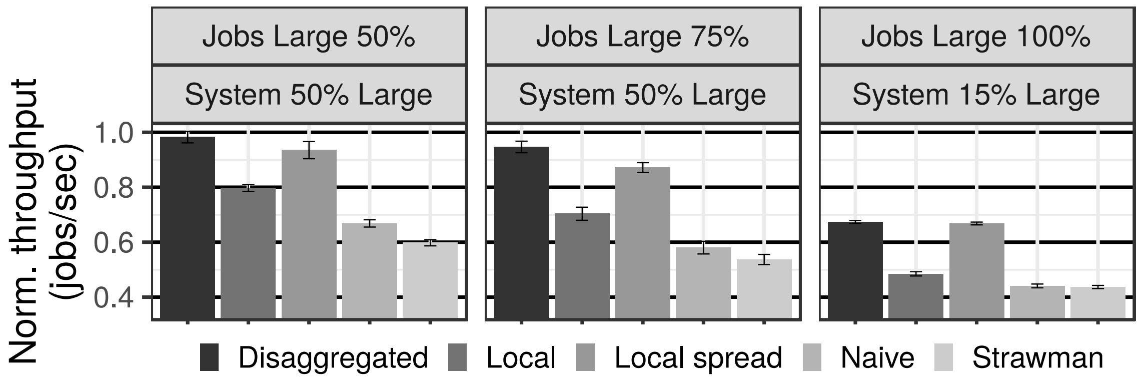


Figure 6.9 Normalized throughput for alternative disaggregated scheduling algorithms.

6.3.6 Scheduling Overhead

Figure 6.10 shows the averaged total scheduling time per job for both approaches in all simulated scenarios. The total time is divided by the number of jobs in each job mix, which varies from 16,000 to 27,000 jobs. To properly compare both approaches, we removed the 0% system bar for the baseline approach, since the baseline is only able to execute one job mix on this system. We can observe that as we add to the system more large capacity nodes, the baseline takes advantage and decreases its total scheduling time. Its large variance for underprovisioned systems is due to the large job mixes that put more pressure on the insufficient resource, consequently generating a bottleneck in the scheduling for the baseline. The increase in scheduling time is almost completely explained by the larger number of attempts to backfill jobs. On the other hand, the disaggregated approach is able to avoid this bottleneck and get more benefits on underprovisioned systems. It presents less overhead than the baseline when memory resources are constrained.

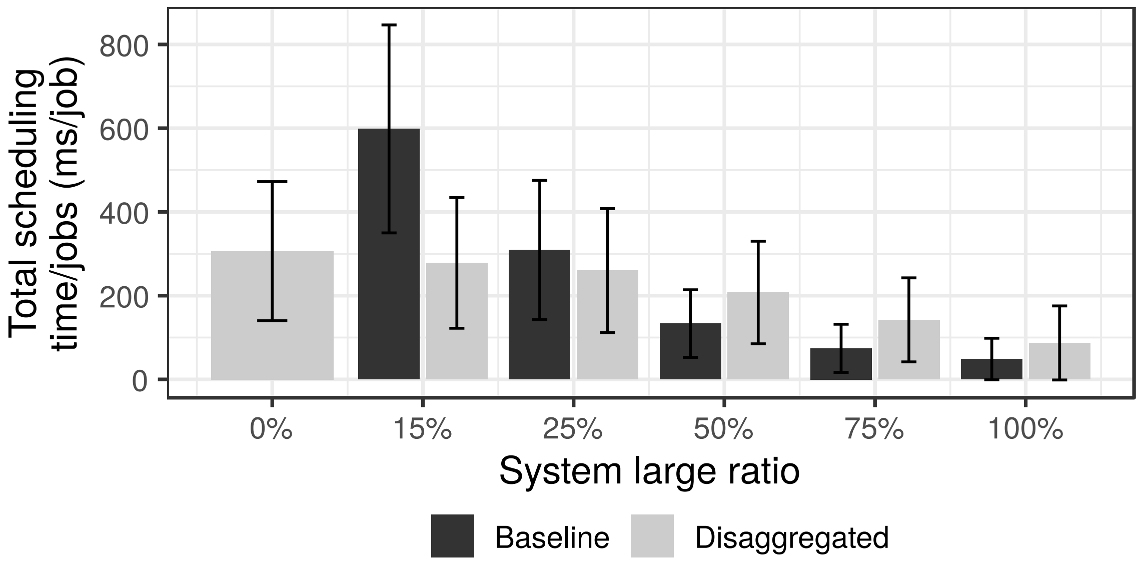


Figure 6.10 Total scheduling time averaged per system.

6.3.7 Constraining Memory Allocation

In order to analyze the effects of the simplification in our contention model (single latency model detailed in Section 4.3.3), we executed an experiment constraining the number of nodes allowed to borrow or lend remote memory. We define a group as the number of nodes that have similar latency among them but would have different latencies accessing other groups, therefore we disable the allocation of memory between groups. By grouping the nodes and limiting the number of remote nodes that can be allocated, we try to quantify the effect that considering only one latency to access any node in the system has on the achieved performance. Figure 6.11 shows the normalized throughput (y-axis) as a function of the number of nodes allowed to share memory (x-axis) in a group. As an example, for a group of 2, a node is only allowed to use its local memory and the memory of a remote node close to it.

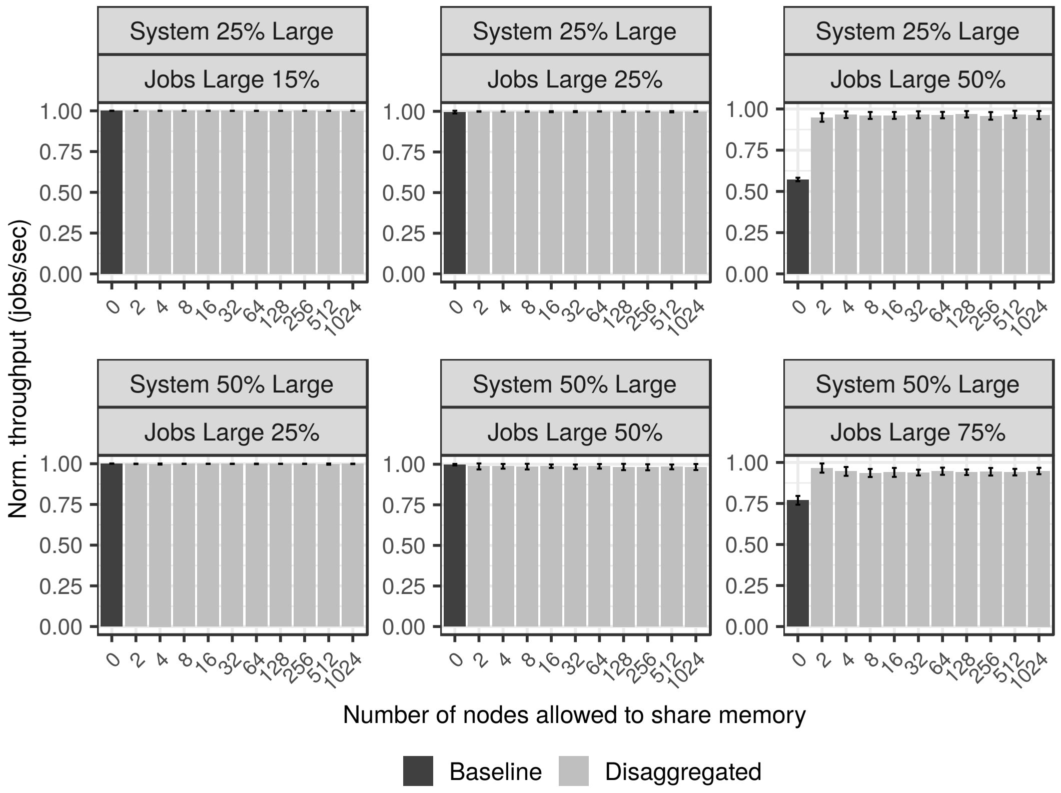


Figure 6.11 Normalized throughput (y-axis) as a function of the number of nodes allowed to share memory (x-axis) considering two systems and different job mixes.

In this Figure we show the results considering the three common scenarios presented throughout this work, which are: the overprovisioned and matched scenarios (leftmost and middle panels), and the underprovisioned scenario (rightmost panels). For matched and overprovisioned scenarios the assumption has no effect on the overall performance achieved as we have seen that in these scenarios there is no need for using disaggregation. These systems have enough resources to run the job mix without requiring remote memory capacity.

On the other hand, for underprovisioned scenarios in which disaggregation is used, most of the gains (comparing the baseline and disaggregated execution) is not due to the simplification of the model using a fixed latency. We can notice that even using a group of 2 it is already enough to outperform the baseline execution. Increasing the number of nodes in the group does not increase dramatically the overall performance as so to impact our results.

In our simulations (up to this point) our setup involves large memory capacity nodes and normal memory capacity nodes (having half of the capacity of a large node). Therefore, the demands of the jobs are no more than twice the memory capacity of the normal nodes, which also helps to explain the disaggregated performance using a small group of nodes. In this sense, an additional remote memory node is already enough to improve the performance compared to the baseline execution, since the baseline is not able to execute high memory demands jobs on lower provisioned systems. This behavior might not hold true for cloud/data-center premises as nodes may support the colocation of jobs in these systems. On the other hand, we target HPC systems in which colocation of applications is rarely done.

6.4 Conclusion

Disaggregated memory addresses the problem of stranded resources inherent in the dominant HPC cluster architecture that causes globally inefficient use of both CPU and memory. In this Chapter, we investigated 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, in this thesis we extended an existing Slurm simulator to support disaggregated memories. For a complete analysis, we developed and integrated a contention model 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 in [144].

It is known that users must express beforehand their resource demands when submitting jobs to HPC systems. The request will be based on users' knowledge or ability to estimate the resources necessary to run the job. Consequently, they will likely overestimate their demands to avoid having the job killed by running out of resources. In this Chapter we assumed that the submitted jobs perfectly estimate their demands. For the following Chapters, we will dive into the effects of the user's ability to predict memory demands and job resource utilization on system performance.

CHAPTER 7

Memory Demands in Disaggregated HPC Systems

DISAGGREGATED memory has recently been proposed as a way to provide a flexible and fine-grained allocation of memory capacity [13, 14, 2, 8, 10, 26]. The addition of disaggregated memory to an HPC cluster architecture would allow applications to share system memory capacity, reducing or eliminating stranded memory resources that would otherwise be unavailable to other HPC jobs, while maintaining the cost-effectiveness and scalability of traditional HPC cluster architectures. Although there is some ongoing work on dynamic resource assignment and malleability [33, 34], most HPC job schedulers statically assign resources to jobs. This means that the user of a disaggregated memory system is required to provide an accurate upper bound on the job's memory demands at submission time.

This Chapter investigates how critical such memory demand bounds are for maximising system throughput and minimising job response time (defined to be waiting time in the queue plus execution time). We analyse to what degree the users would have a natural incentive to provide accurate memory bounds. Our analysis uses the extended BSC's Slurm simulator for disaggregated memory systems (see Chapter 6).

We use a simulation approach for two main reasons. Firstly, there are no large-scale HPC systems with disaggregated memory hardware including a complete software stack. Secondly, and more importantly, simulations allow studies to be performed more quickly without occupying the resources of large-scale production systems. Moreover, the correlation analysis of Section 7.1.2 can only realistically be done using simulation.

In our studies, we show that from the HPC system operator's perspective, overall system throughput is conditional on accurate memory estimations, but, from the perspective of a single user on a large system, there is little incentive to provide an accurate bound on memory consumption. Even when the cost of a 60% increase in memory demands only increases a single job's user response time by 8%, the aggregate result of everybody doing so can be a 25% reduction in system throughput and a 5× increase in average response time. We make some initial recommendations and encourage additional research in this direction. If these results are reproduced more widely, then it will almost certainly be necessary to allocate GB–hour memory capacity explicitly, as part of the peer review process, in addition to the core–hours.

The contributions presented in this Chapter are published in our paper [41]. In summary, we make the following contributions in this Chapter:

① We use the extended BSC's scalable Slurm simulator for disaggregated memory to investigate how the user-specified memory upper bound affects overall system throughput.

② We introduce a simulation-based methodology to correlate the accuracy of the memory upper bound with the job's response time.

③ Assuming these results can be reproduced more widely, we make recommendations that can be applied to production systems.

7.1 Extending the Simulator with Memory Overestimation

We first extended the Slurm simulator with a memory overestimation module that represents the user, by determining the memory consumption bound at submission time, as a function of the actual peak memory consumption and the intended user behavior. The output of this module is the estimated upper bound, which the Slurm simulator passes to the disaggregated-aware Slurm in order to allocate resources. The actual memory consumption, however, is still used by the Slowdown model to determine the effect on performance.

In the baseline experiments, the upper bound equals the actual memory consumption. This is the unrealistic best case, in which the users perfectly estimate each job's memory consumption. Otherwise, the estimated memory bound can be either (1) overestimated by a fixed percentage, which can be used to quantify the general effect of overestimation on overall system throughput and response times (in our tests, the fixed percentage varies from +0% to +60%), or (2) overestimated by an independent, identically distributed (i.i.d.) uniformly-random percentage between +0% and +100%, which is used by our correlation analysis to quantify the effect on single job response time.

7.1.1 Determining the Effect on System Throughput

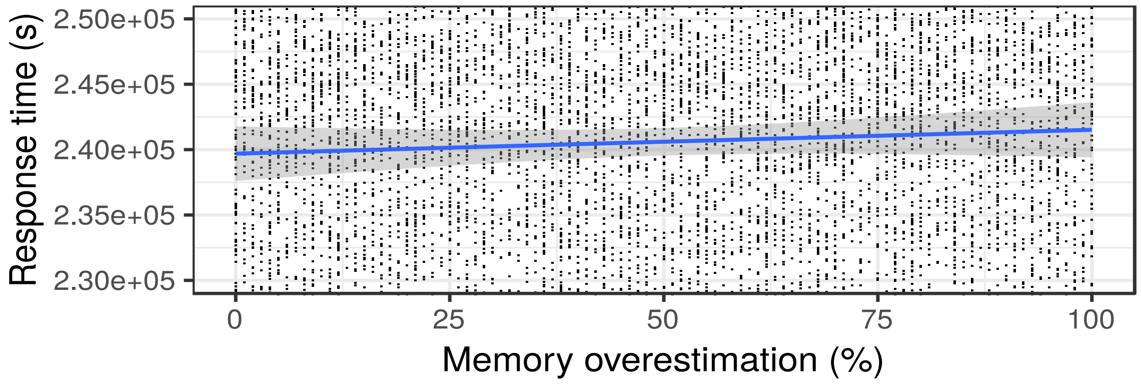
To determine the effect of overestimation on system throughput, we configure the memory overestimation module to uniformly overestimate the memory demands of all jobs, between +0% (the baseline) and +60%. We then plot system throughput as a function of the overestimation. The impact of the overestimation on the system's performance is discussed in Section 7.2.1.

7.1.2 Correlating Memory Overestimation and Response Time

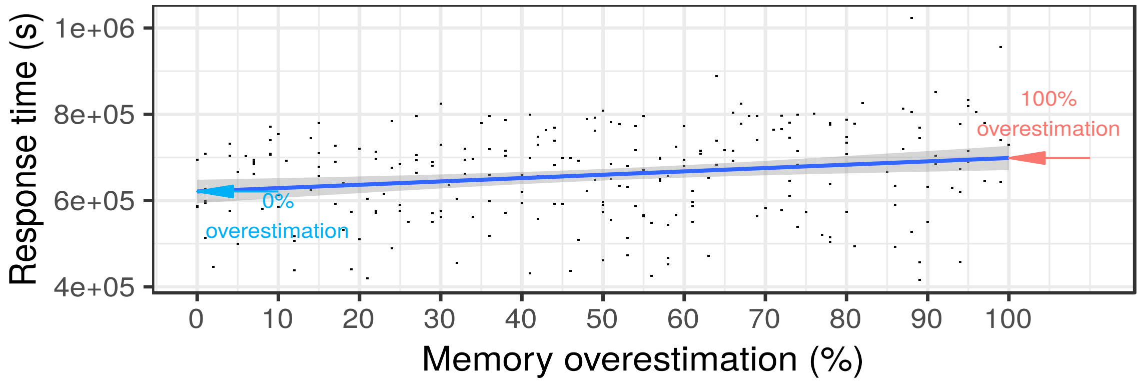
To determine the effect of overestimation on individual job response time, we may plot a typical job's response time as a function of its memory overestimation, holding everything else constant. But it is clearly not practical, in terms of simulation time, to do this one job at a time. Instead, we perform a correlation analysis, by running the original trace several times, applying a uniformly-random overestimation to each job. For this correlation analysis to make sense, it is important that the degree of memory overestimation is independent of other job characteristics. This is one reason why the simulation approach is important, as it allows us to apply an i.i.d. random overestimation. Observational data may be misleading, for instance, if larger jobs systematically had a larger (or smaller) degree of overestimation.

Figure 7.1a shows a direct plot of the response time as a function of the memory overestimation, across all jobs, for the scenario with 50% large jobs and 0% large nodes. The full results, for all scenarios, are in Section 7.2. We add a trend line using linear regression. Since the jobs have widely varying response times, even with no memory overestimation, the points on the y-axis have a very large range, of which Figure 7.1a shows a small part. We, therefore, filter the jobs using the baseline response time, when the overestimation is +0%, to obtain Figure 7.1b, which is for jobs whose baseline response time was between 3×10^5 s and 4.2×10^5 s. In this Figure the y-axis presents the response time when the jobs overestimate their memory consumption. Similar plots were obtained for each interval of baseline response times. There is still significant noise, but it is much less than before. Figure 7.1b also shows the trend line, which allows us to predict the average response, for jobs with the given range of baseline response times, i.e. from +0% to +100%, as a function of the memory overestimation.

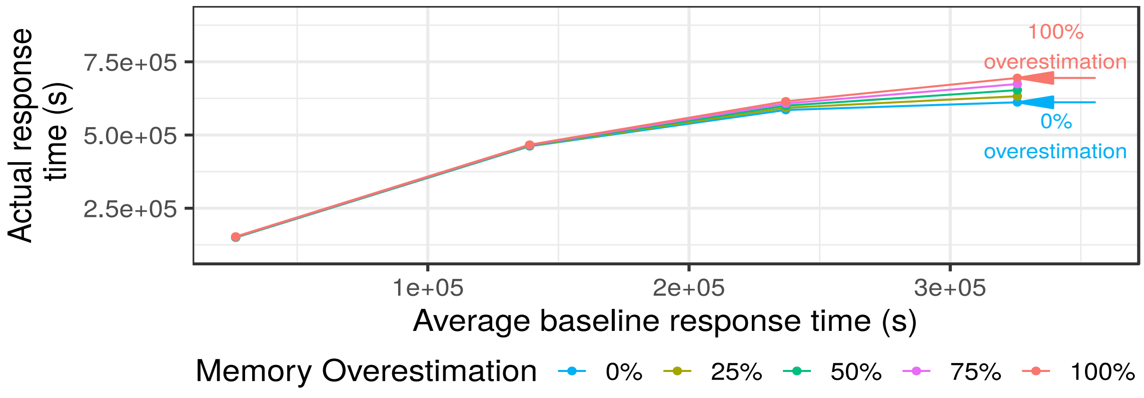
Finally, Figure 7.1c assembles all the information into a single figure. The x-axis is the baseline response time, for +0% overestimation, and the y-axis is the actual response time, depending on the overestimation (five different curves). In this example, we see a large increase in the overall response times, e.g. from 2.0×10^5 s to 7.5×10^5 s for the top-rightmost point, but we see very little difference between the +0% and +100% cases. In Section 7.2.3, we will show the complete results, which follow a similar trend.



(a) Response time for all jobs.



(b) Filtering response time using a fixed interval range.



(c) Using trend line to derive the response time.

Figure 7.1 Correlating memory overestimation to response time (example with 50% large jobs and 0% large nodes).

7.2 Results

Here we evaluate and discuss in more detail the effects of job requests that may overestimate memory on the system performance and user response time. We analyse the difference it makes to the users being perfectly accurate by specifying their requests, and what happens to the requests on the system. The details for the configuration of our simulated environment, methodology used for input generation, and its distribution applied in our experiments are listed in Chapter 5.

7.2.1 System Job Throughput

Figure 7.2 presents the throughput of the system, in jobs per second, as a function of the memory demand overestimation, across all scenarios. We notice that, in all cases, system throughput drops as the overestimation increases. Even though for low degrees of overestimation, the impact on throughput is modest, the degradation increases with the mismatch between the system and the job mix, reaching almost 40%. We, therefore, conclude that, from the system operator's perspective, effective system utilization requires that the jobs generally have accurate estimations of their memory demands.

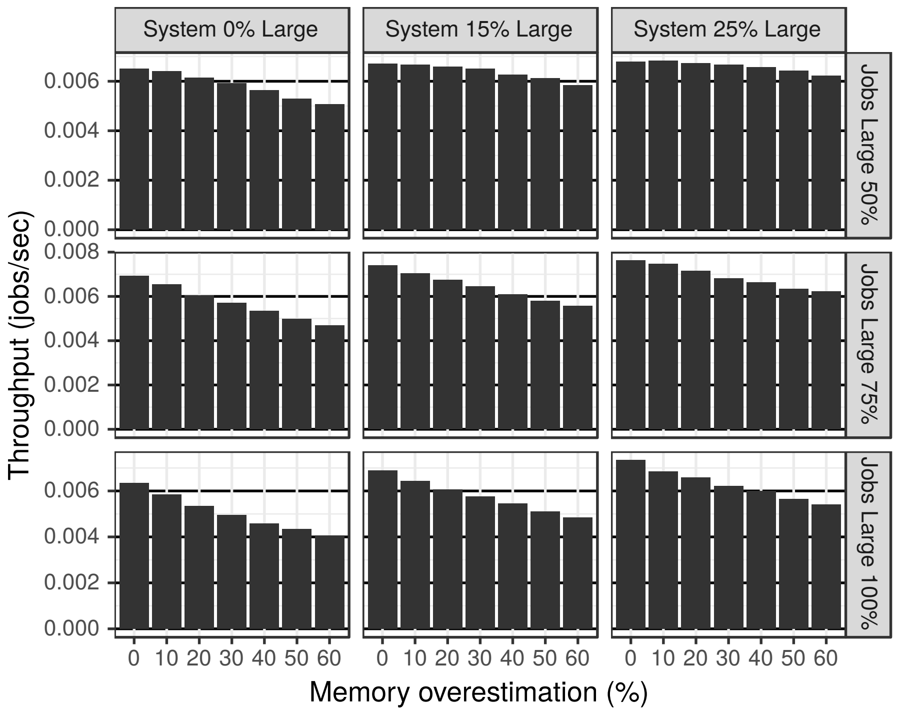


Figure 7.2 System throughput (*y-axis*) when all jobs overestimate memory requirements by the same percentage (*x-axis*).

7.2.2 System Job Response Time

Figure 7.3 shows the average response time when all jobs overestimate the memory demands by the same amount. Following a similar trend as the decrease in throughput, the response time increases, showing that the system's response time is impacted as a whole when all jobs overestimate their requests. For such a scenario, we notice that when all users are accurate in specifying the memory usage, it would benefit the whole system, because it would decrease the load on the system in queuing time due to the lack of resources. Consequently, the system would run more efficiently by decreasing overall response time and increasing overall throughput.

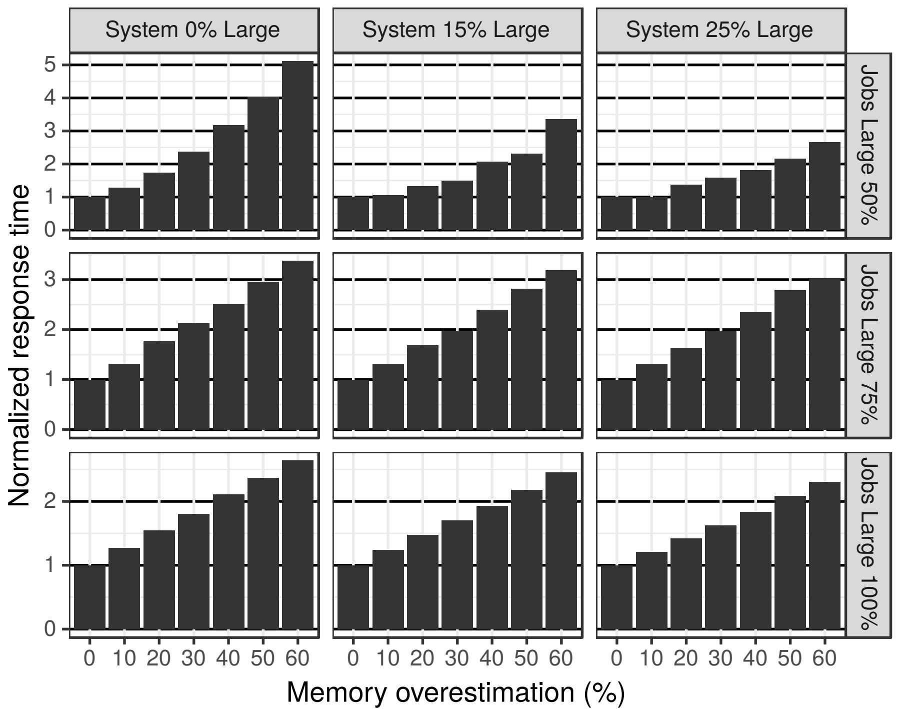


Figure 7.3 Normalized response time (*y-axis*) when all jobs overestimate memory requirement by the same amount (*x-axis*).

7.2.3 User Job Response Time

Each plot in Figure 7.4 shows the average response time (y-axis) as a function of the baseline response time (x-axis) for the jobs of two types of users. The data is obtained following the methodology described in Section 7.1.2. The 0% line is for a "diligent" user who accurately determines the memory consumption whereas the 100% line is for a "careless" user whose prediction is twice the actual consumption.

Results are shown in a 3×3 grid, corresponding to the three different systems and job mixes. Figure 7.4 plots the actual average response time, whereas Figure 7.5 plots the normalized response time.

We see in each scenario a large increase in response time, compared with the baseline (represented by the black line). We notice, as expected, that in all scenarios the response time is impacted even for the diligent user, whose jobs do not overestimate the memory demands. For *0%* large node system, where disaggregation is more often used to accommodate the job mixes, we perceive a slight increase in response time when the job doubles its request. However, there is little or no difference in the response time when we start adding large nodes to the system. We observe that being accurate in a scenario where other users are not, provides a small benefit to a single user, whose jobs will be impacted by the load the other users create on the system due to their overestimation.

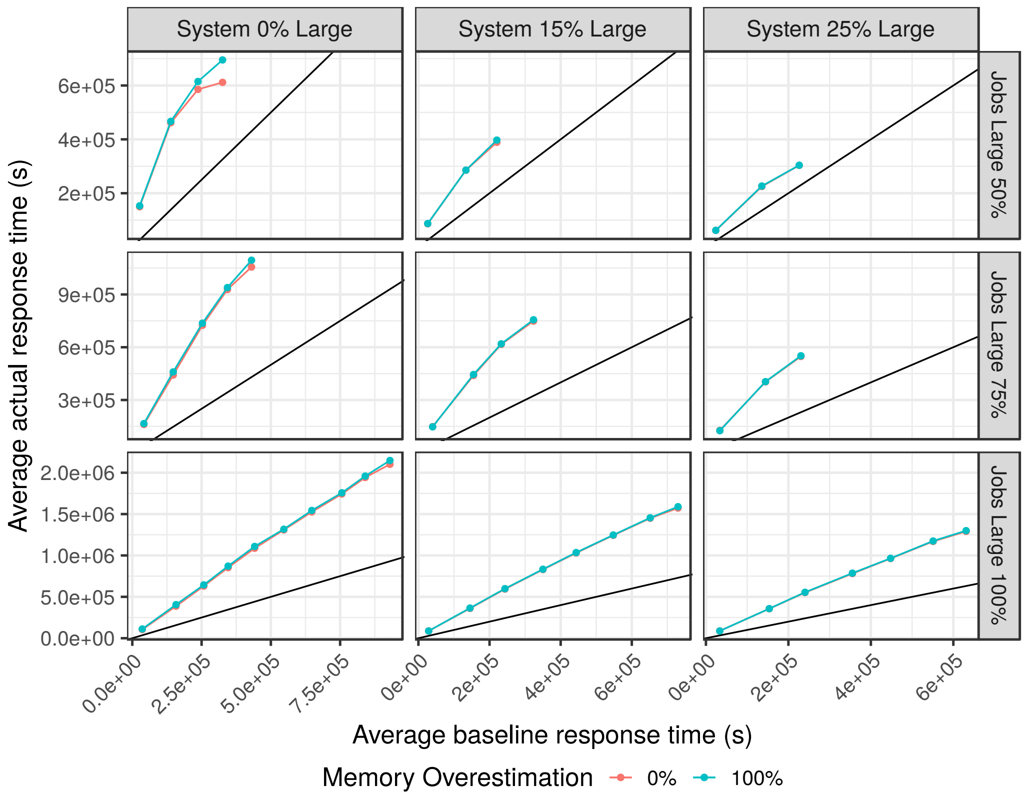


Figure 7.4 Individual job response times increase when the users overestimate job memory demands, but memory overestimation has little effect on response times (+0% curves vs +100% curves).

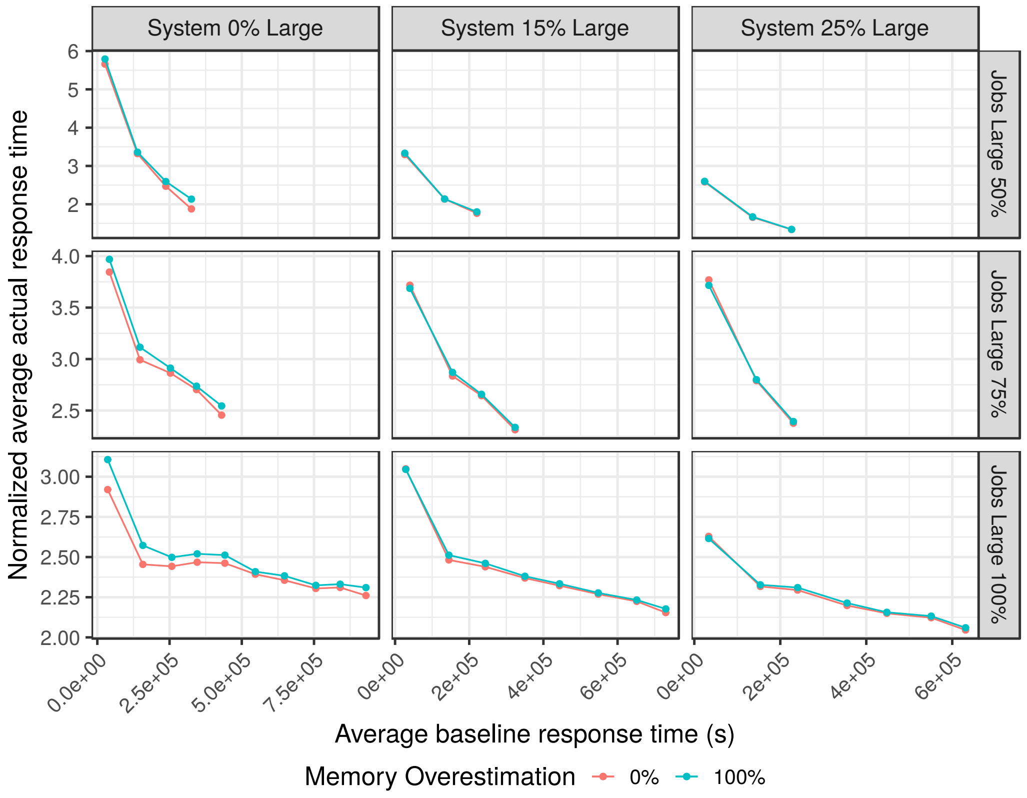


Figure 7.5 Normalized figures for individual job response times.

7.3 Conclusion

Disaggregated memory is under development as a way to provide a flexible fine-grained allocation of physical memory. Users of an HPC system supporting disaggregated memory would likely be expected to estimate their job's memory demands. In this Chapter, we investigated how the system's overall throughput and response time would be affected, according to various assumptions on the user's ability to predict memory consumption. We find that even when there is a large effect on system throughput (-25%) and response time (5× higher), there is a very little direct incentive for the users to be accurate in their estimates, with only an 8% increase in response time. We make a step towards understanding how to bring disaggregated memory to HPC, by demonstrating that users should receive incentives to provide accurate memory usage estimates. These incentives could translate to an increase in priority, the number of simultaneous running jobs, or larger core–hour allocations.

CHAPTER 8

Dynamic Memory Provisioning on Disaggregated HPC Systems

In the previous Chapter we investigated how the system is affected by the users' ability to estimate the peak memory consumption of their jobs. However, HPC applications have widely varying per-node memory footprints due to diverse application characteristics, differing problem sizes, and strong scaling [2, 6, 7]. Imbalanced memory usage can happen across compute nodes and time in a job [11]. The system's resources are severely underutilized when the system is not running its worst case workloads, first because many HPC systems do not allow sharing resources. Second, due to homogeneously configured nodes in the current systems [11, 56], which usually are overprovisioned as a temporary solution to allow most of the workloads to fit within the resources of a single node [32, 55, 35].

In a typical HPC cluster architecture, memory is tightly coupled to the CPUs running the jobs, leading to stranded memory capacity and inefficient use of the memory resources. In fact, 25% to 76% of the total memory capacity typically remains idle [56, 10, 55, 68]. 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 [32]. It enables fine-grained allocation of memory capacity to jobs [13, 14, 2, 8, 10, 26], while maintaining the cost-effectiveness and scalability of a cluster architecture [55].

HPC job schedulers generally allocate resources statically [11], so disaggregated memory resource management systems require the user to specify the peak memory demands at submission time [40, 10, 11]. It is difficult for users to know the precise maximum memory footprint, and they have the incentive to overestimate this figure to avoid an out-of-memory error, which would terminate the job [36–38, 2]. In the previous Chapter, we investigated the incentive for users of a disaggregated memory system to provide accurate memory estimates. We showed a tragedy of the commons effect: even when a 60% overestimation increases a user's total time from submission to completion (the response time) by just 8%, the result of everybody doing so can be a 5 times increase in response time and 25% reduction in throughput.

In this Chapter, we make 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 BSC Slurm simulator [57]. 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 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 12%. When memory demands are overestimated by +60%, the improvement in performance for an underprovisioned system is up to 18%. 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 [33, 34, 106]. 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 [31, 32]. In contrast, our simulation approach has allowed the evaluation to be performed on up to 1490 nodes. A more extensive comparison with related work is in Chapter 3.

The contributions of this Chapter can be found in our paper [42] under submission. In summary, the major contributions of this Chapter are:

① We extend Slurm's memory allocation policy and BSC's Slurm simulator to support disaggregated memory with dynamic memory reallocation.

② 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 the cost–benefit in HPC systems.

③ We demonstrate that dynamic memory assignment delivers improvements up to 18% in throughput, 38% in throughput per dollar, and up to 69% reduction in job response time (median), compared to a static policy, when there are imbalanced memory usage and overestimated demands on underprovisioned systems.

8.1 Dynamic Memory Allocation Policy

We enhanced BSC's Slurm simulator, extended for disaggregated memory presented in Chapter 6, 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). Figure 8.1 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 presented in Chapter 6, 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 works [78, 75] have already demonstrated low overhead for data collection to track job executions, with sampling intervals on the order of seconds. In our work, we update the memory usage on average by every 5 minutes. The usage information is passed to the Slurm controller, which updates the job memory allocations. In Section 9.1, we provide an additional discussion regarding the *Monitor* module.

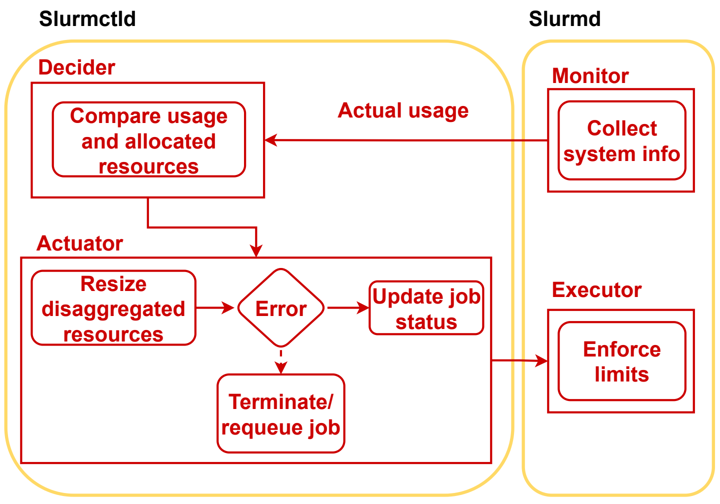


Figure 8.1 Proposed dynamic allocation of disaggregated memory and its integration into Slurm.

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 allocated 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, then remotely if necessary. In practice, when a job dynamically allocates memory (e.g., using the *malloc* or *new* functions), it initially utilizes the memory that has been reserved for it by Slurm. However, if the job exceeds the reserved memory, it will be temporarily blocked, and Slurm will search for available memory capacity within the entire system. Once found, the additional memory will be allocated and assigned to the job, ensuring its continued execution. 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 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 namely: *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, and it is impractical to checkpoint in response to a failed memory allocation and restart from an arbitrary backtrace. Checkpointing, therefore, has to be done periodically at defined points in the execution. 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.

8.2 Dynamic Memory Allocation in the BSC Slurm Simulator

Figure 8.2 shows our approach to evaluate the dynamic allocation of disaggregated memory using BSC's Slurm simulator. Our modifications are depicted as orange boxes. The functions that are partially implemented or adapted to run in a simulated environment rather than a real system are represented as dotted boxes. It worths mention that the scheme depicted in this Figure deals with jobs already running in the system, therefore they are subject to memory usage variation and dynamic memory management. On the other hand, Figure 6.3 demonstrates the initial static scheduling and resource allocation of jobs submitted to our disaggregated approach.

In this simulated environment, the *Decider* module receives the memory usage from the offline memory usage trace (details on Section 5.2), rather than periodically 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 the job execution for

[1] 100% large jobs, 50% system, +100% overestimation (see Chapter 5).

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.

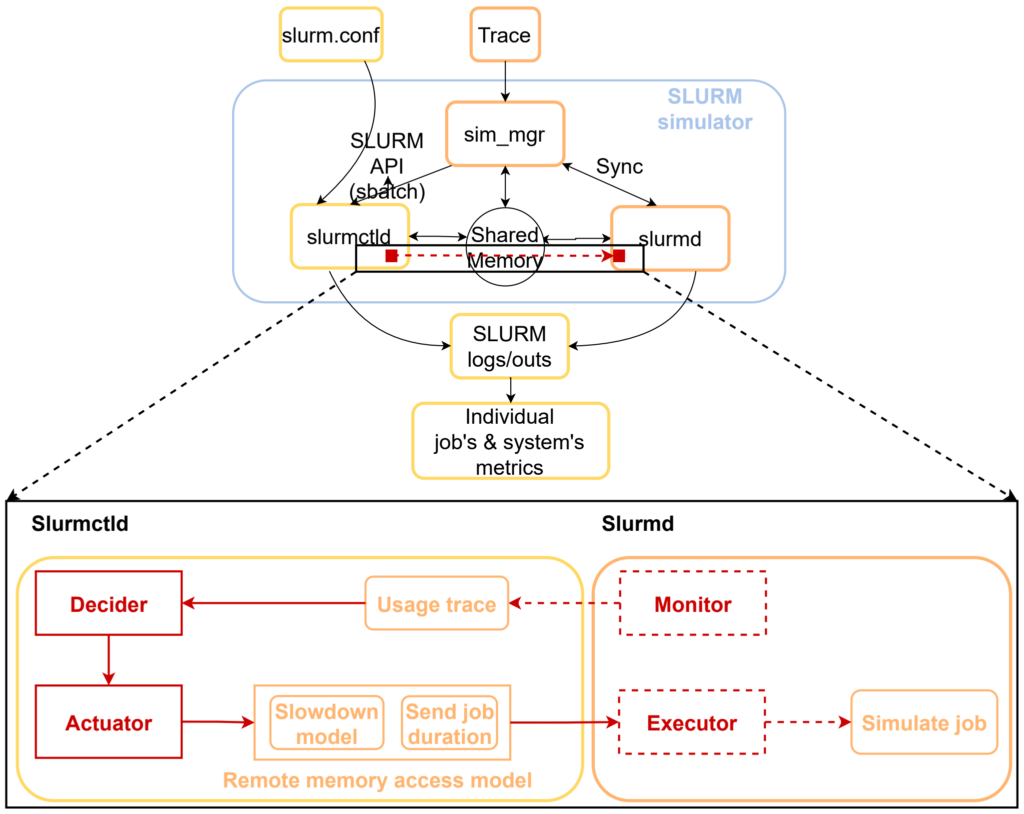


Figure 8.2 Proposed dynamic allocation of disaggregated memory and its integration into BSC Slurm simulator. The scheme considers jobs already running.

8.3 Allocation Policies

We present our results evaluating 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 (Chapter 6).
- **Dynamic**: disaggregated memory with dynamic memory allocation policy (Section 8.1).

8.4 Results

8.4.1 System Throughput (Jobs per Second)

Each plot in Figure 8.3 shows the normalized throughput, in jobs completed 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 left column 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 right column correspond to a more realistic +60% overestimation. The rows show different proportions of large jobs for the synthetic trace, together with the Grizzly trace at the bottom.

When the demand for memory consumption is low, e.g. in the top-left panel corresponding to +0% overestimation (left) and 0% large jobs (top), the system memory can be reduced to 25% (32 GB per node instead of 128 GB per node), without any impact on throughput for the disaggregated approaches. Nevertheless, since the normal jobs require up to 64 GB per node, once the memory provisioning goes below 64 GB per node (50% total system memory), the baseline (no disaggregated memory) approach becomes unable to run some of the jobs, therefore not showing bars below 50% system. As the proportion of large jobs increases, along the left column, memory has an increasing effect on system throughput. We see a large difference between the baseline and static approaches, and up to 12% 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 right column of the Figure, the peak memory footprint is overestimated by a more realistic +60%. In this case, the baseline approach is unable to execute all the jobs, 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 15% large jobs and a system with 25% total memory, the dynamic approach achieves throughput over 95%, which is 18% 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.

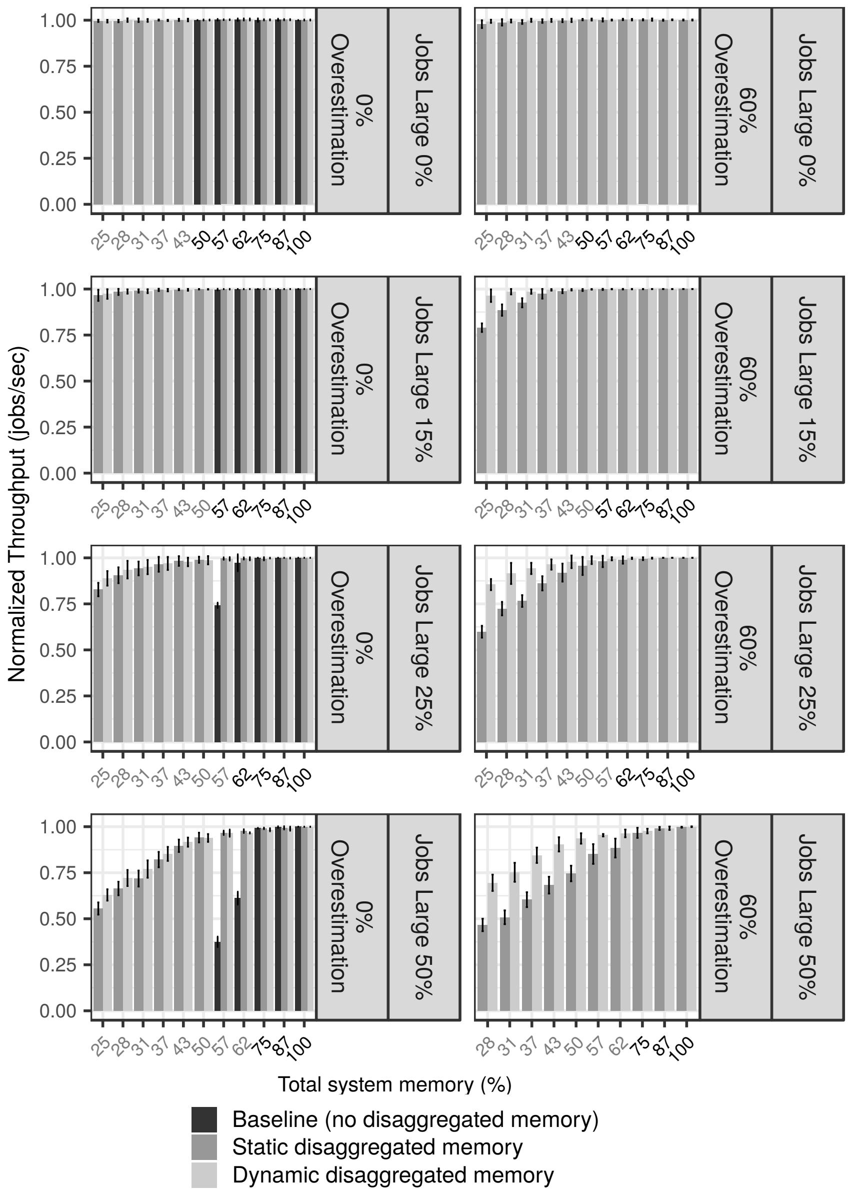


Figure 8.3 Normalized throughput (y-axis) for each memory configuration (x-axis) for various job mixes. The left column has +0% memory overestimation and the right column has +60% overestimation. Bold x-axis labels identify overprovisioned memory systems. Missing bars in the plots indicate there are 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. Continues on the next page.

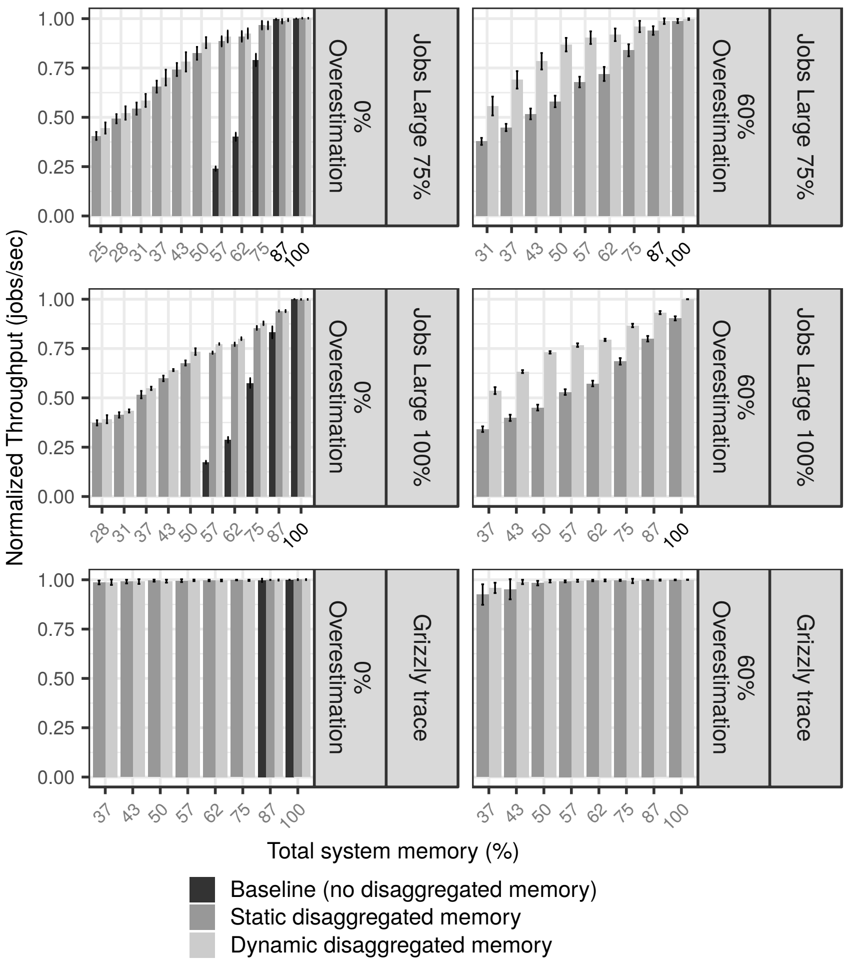


Figure 8.3 Continued: Normalized throughput (y-axis) for each memory configuration (x-axis) for various job mixes. The left column has +0% memory overestimation and the right column has +60% overestimation. Bold x-axis labels identify overprovisioned memory systems. Missing bars in the plots indicate there are not enough large memory nodes to run all jobs.

8.4.2 Job Response Time

Figure 8.4 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 the underprovisioned system, the median response time (y-axis equals 0.5) 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.

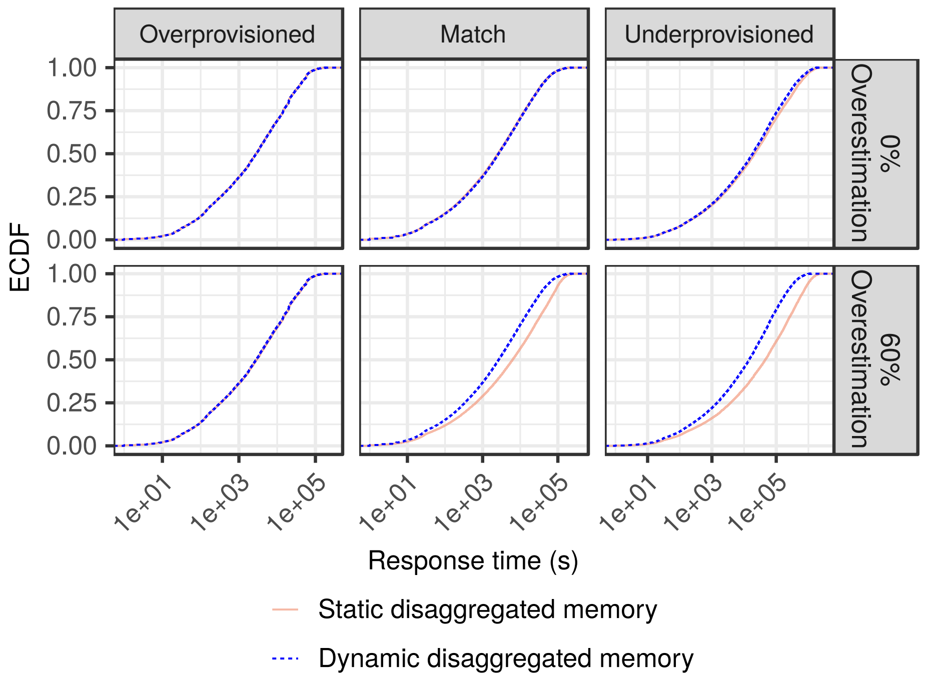


Figure 8.4 Empirical cumulative distribution of response time for different systems and job mixes. For +60% overestimation and underprovisioned systems (bottom right), the dynamic approach has a 69% lower median response time (note: logarithmic x-axis).

8.4.3 Cost–Benefit Analysis

Figure 8.5 shows the results of the cost–benefit analysis, assuming the component costs given in Table 5.2. 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 plots.

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 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%.

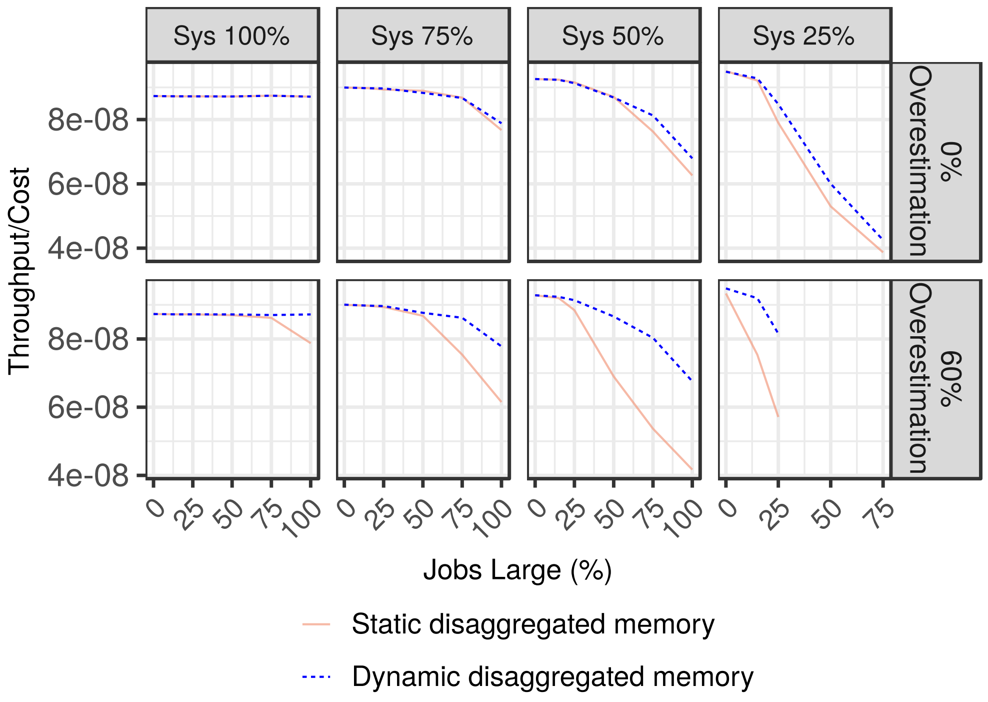


Figure 8.5 Cost–benefit analysis: throughput per cost (y-axis) as a function of the job mix (x-axis). The dynamic approach has a gentler drop in throughput when memory demand is high, reducing the risk of memory underprovisioning.

8.4.4 Minimizing Memory to Achieve Defined Throughput

Figure 8.6 shows the amount of resources 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 when we increase the 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.

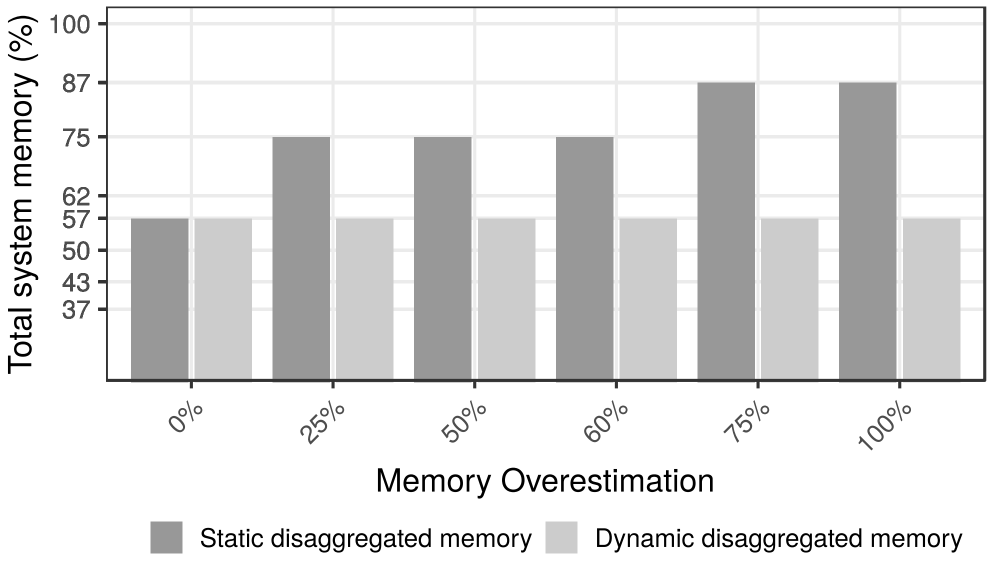


Figure 8.6 System resource provisioning (y-axis) as a function of the memory demand overestimation (x-axis) to achieve 95% of the fully provisioned throughput. Results are shown for synthetic trace with 50% large jobs.

8.4.5 System Utilization of Memory and CPU

Figure 8.7 shows four scatter plots of system memory utilization on the y-axis vs. system CPU utilization on the x-axis. System CPU utilization is the fraction of cores in the system that are allocated to jobs. Both utilization figures are relative to the resources of the system on which the trace was executed. The y-axis is the amount of memory used by the application, rather than the potentially larger amount of memory allocated to it, much of which may be unused. This provides a common basis for comparison between the static and dynamic approaches.

As shown in previous sections, the static and dynamic approaches have similar performance for a match or overprovisioned scenarios with no overestimation. Both are constrained by CPU with moderated memory utilization. For the underprovisioned scenarios dynamic approach have a slightly higher utilization. The difference between the approaches is emphasized even more when the jobs overestimate memory demands. The dynamic approach achieves higher utilization on both overprovisioned and underprovisioned scenarios, as a result of releasing unused resources which are thus used by jobs waiting in the queue. More jobs running in the system increase utilization for both CPU and memory while decreasing the system's waiting time. On the other hand, the static approach will have lower utilization as it will keep unused memory allocated, therefore preventing jobs to start in compute nodes that had their memory exhausted by another job, decreasing CPU and memory utilization.

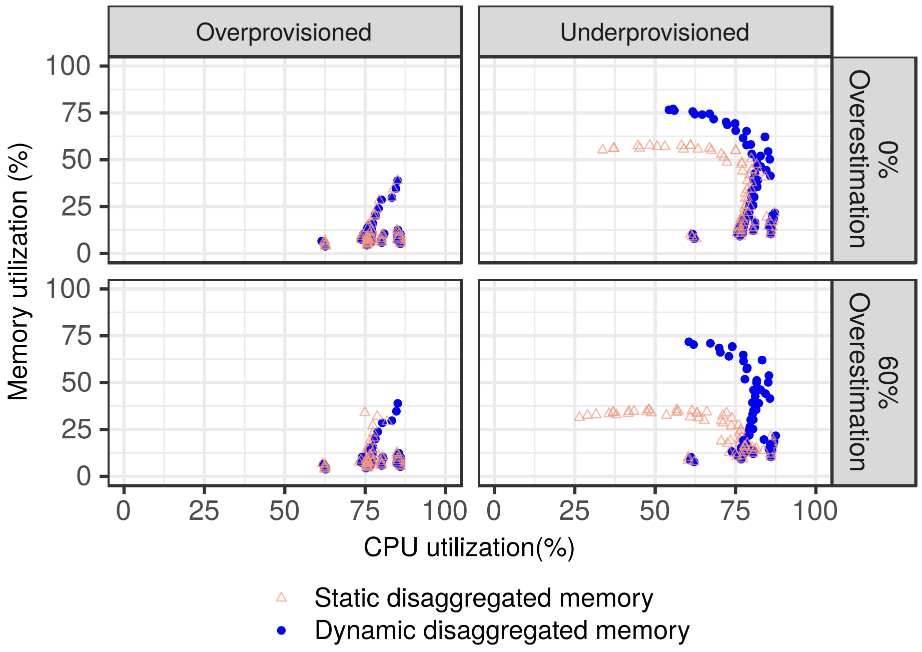


Figure 8.7 Average memory and CPU utilization. The dynamic approach has higher memory utilization when memory is underprovisioned or overestimated.

8.4.6 Effect of Overestimation on Individual Job Response Time

In Chapter 7 we identified a tragedy of the commons situation, which we reconsider in Figure 8.8. The x-axis indicates the job response time in a baseline execution (with +0% overestimation) and the y-axis indicates the average job response time in a modified execution. There are two kinds of users. The solid 0% line is for a "diligent" user who accurately determines the peak memory footprint, whereas the dashed 100% line is for a "careless" user whose memory demands are twice the actual consumption. The red pair of lines is for the static allocation policy. The distance between the red and black lines shows that the increase in response time doubles, and the similarity between the two red lines indicates that both kinds of users suffer equally so there is no incentive to be accurate. In opposition, the blue lines for the dynamic policy demonstrate that overestimation of the jobs' memory demands increases response time by at most 10% since the system takes into account the actual memory consumption rather than the memory demands. The response time of the dynamic approach is only slightly affected by users overestimating their memory consumption.

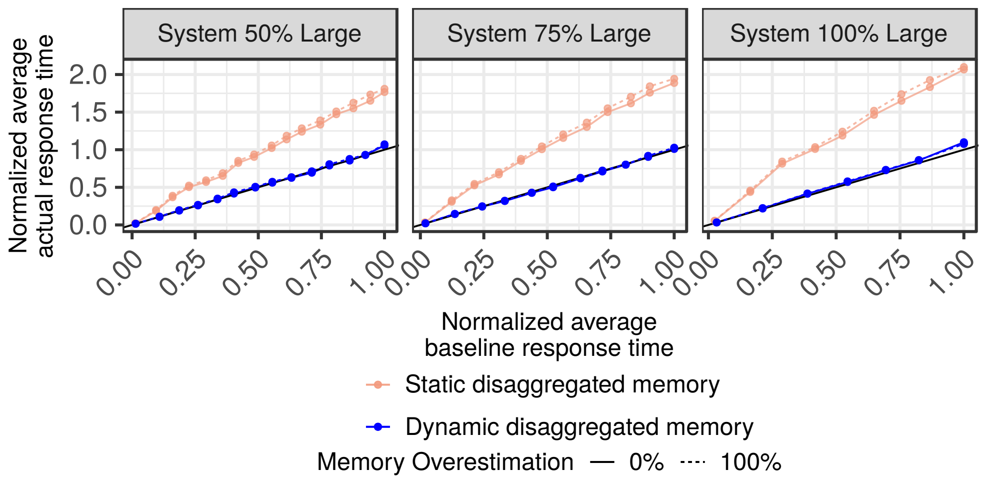


Figure 8.8 Actual job response time (y-axis) as a function of the job baseline response time (x-axis +0% overestimation). The response time of the dynamic approach is only slightly affected by users overestimating the memory consumption, unlike the static approach.

8.4.7 System Throughput vs. Overestimation

Figure 8.9 shows the throughput, on the y-axis, as a function of the system memory capacity, on the x-axis. Running from top to bottom, each panel shows a different amount of overestimation, from +0% to +100%. The left column is for the synthetic trace with 50% large jobs and the right column is for the Grizzly trace.

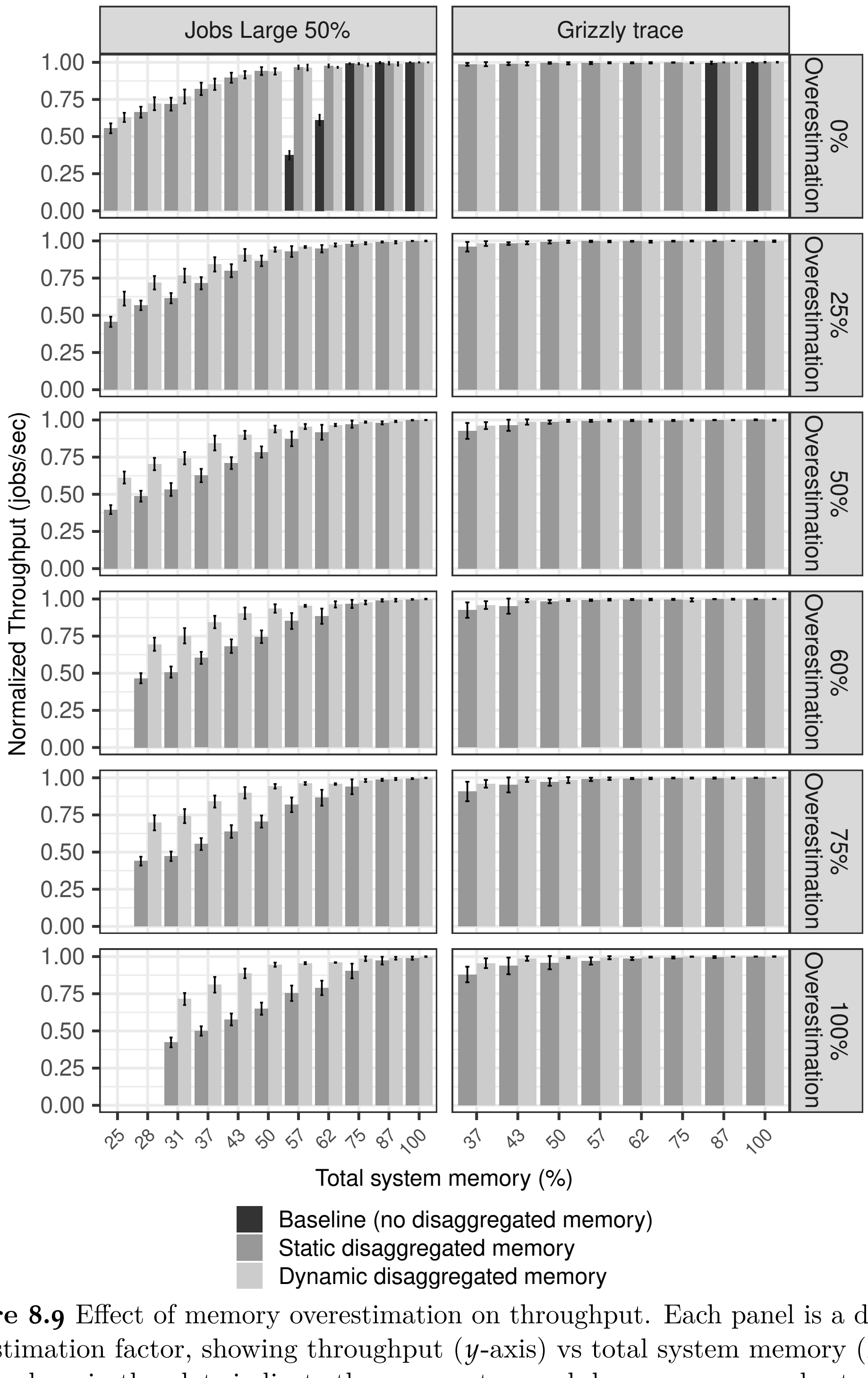


Figure 8.9 Effect of memory overestimation on throughput. Each panel is a different overestimation factor, showing throughput (y-axis) vs total system memory (x-axis). Missing bars in the plots indicate there are not enough large memory nodes to run all the jobs. Compared with the static approach, the dynamic approach is less affected by memory overestimation. Results continue on the next page.

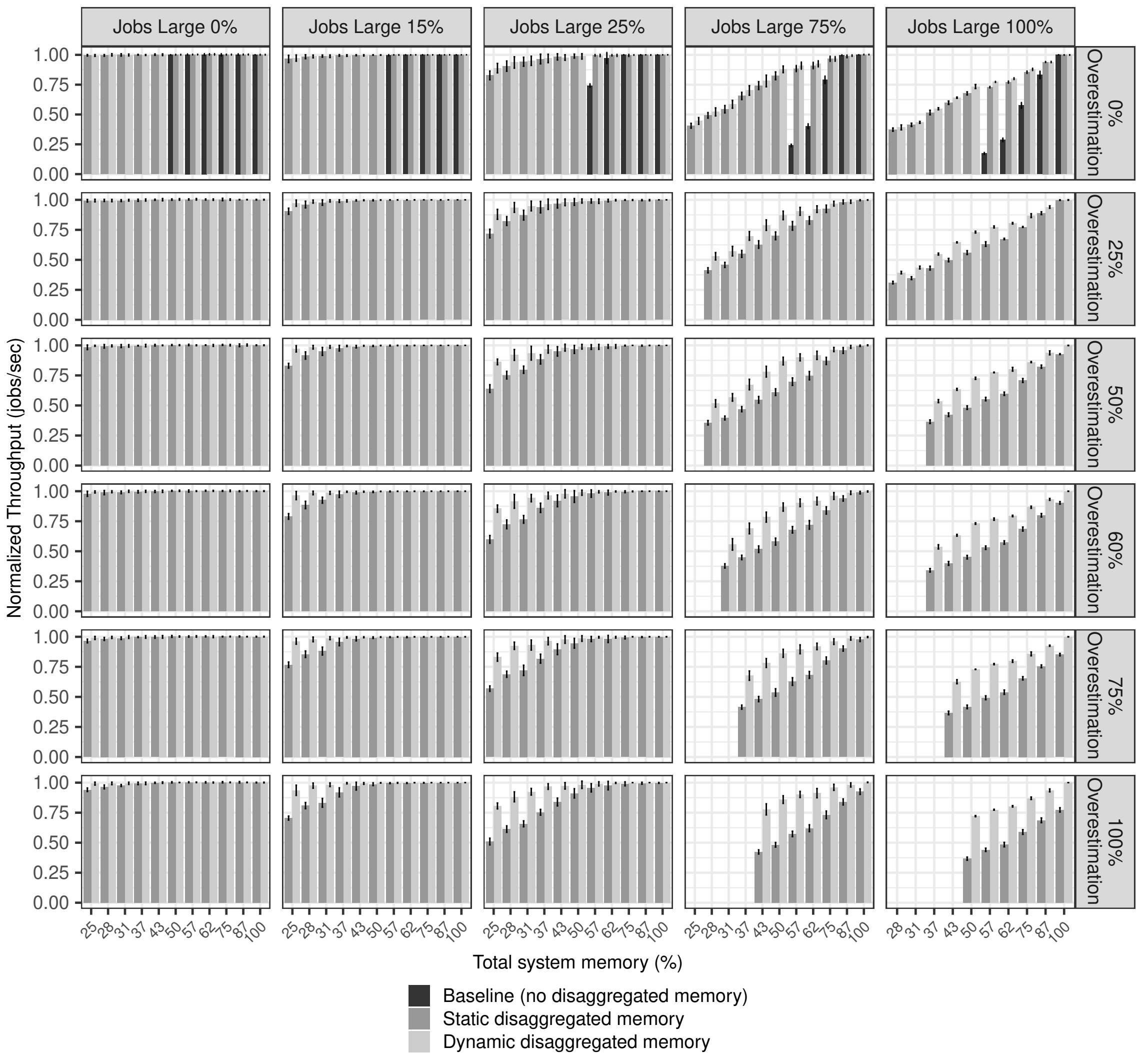


Figure 8.9 Continued: Effect of memory overestimation on throughput. Each panel is a different overestimation factor, showing throughput (y-axis) vs total system memory (x-axis). Missing bars in the plots indicate there are not enough large memory nodes to run all the jobs.

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%. It demonstrates that a dynamic approach is able to run higher load demands on fewer resources with better resource management, therefore reducing the investments in resources.

8.4.8 Initial Memory Provisioning

In this Chapter, as it is common in most of the current HPC systems, we assume that the system requires the user to express their memory demands at submission time, and as a consequence, these demands are likely overestimated. In this section, we analyze the opposite effect. At submission time, the system will proactively decrease the memory request submitted by the user and the resource manager's dynamic memory management will handle the allocation based on the job's usage. Figure 8.10 presents the normalized throughput (y-axis) as a function of the decreased memory requested by all jobs (x-axis) considering different job mixes and systems large node ratio. In this experiment, we use as the starting point the scenario in which the jobs do not overestimate (+0%) their demands, and then we uniformly decrease all jobs memory requests at submission time.

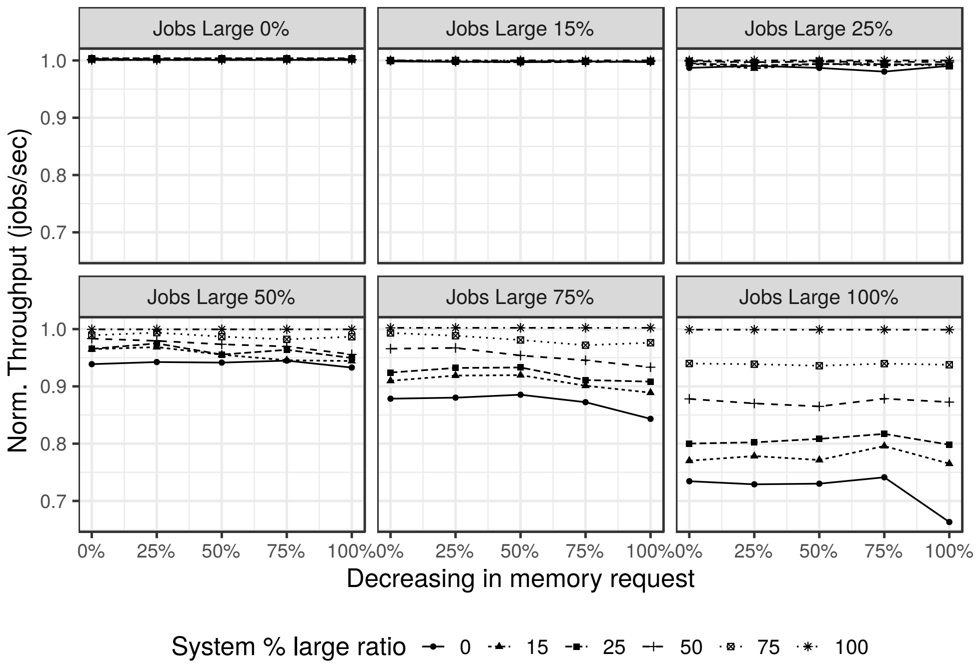


Figure 8.10 Normalized throughput (y-axis) as a function of the decreased memory requested by all jobs (x-axis) considering different ratios of large capacity nodes in the system. Decreasing memory requests does improve performance.

We can notice that there are no cases when decreasing the initial memory request (going to the right on the x-axis) improves the performance. For job large mixes 0%, 15%, and 25% (top row), decreasing the memory request does not have an impact on the overall performance even when the system is underprovisioned. The reason is that the majority of the systems are overprovisioned to run these job mixes. The compute nodes allocated to the jobs have enough memory to satisfy the job demands and its execution, therefore they will not need to contend for remote memory capacity.

On the contrary, the findings indicate that when job demands exceed the system provisioning in underprovisioned scenarios, the throughput decreases for larger job mixes (50%, 75%, and 100% mixes) even if the submitted memory request is not reduced (0% on the x-axis). However, if the system is provisioned to meet the memory requirements of the job mix (matched or overprovisioned scenarios), decreasing the memory request has little impact on overall performance. Nonetheless, we did observe a slight decline in performance when the memory request was reduced, with reductions of 1%, 4%, and 10% for the most extreme case (100% reduction in memory request on a 0% large node system) in the bottom row of the graph.

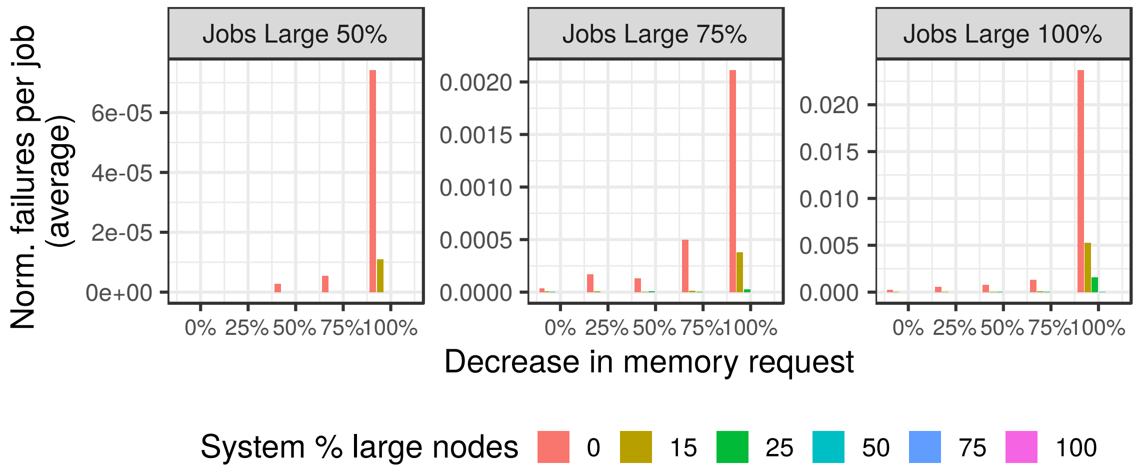


Figure 8.11 Averaged normalized failure per job (y-axis) as a function of decreasing memory requests for all jobs (x-axis) considering different ratios of large capacity nodes in the system. The number of failures rises when there is a disparity between the system and job mix, as well as when the initial memory request is decreased.

Figure 8.11 helps to explain the reason why decreasing the memory request at submission time does not perform well for underprovisioned scenarios. It shows the average normalized failure per job (y-axis) as a function of the decreased memory requested by all jobs (x-axis). Only three scenarios consistently presented failures, as we can see in the Figure. It is evident that the greater the reduction in the memory request, the greater the number of job failures, especially for systems that are underprovisioned to handle the job mix. This effect can be attributed to the system being more aggressive in initiating jobs with low initial memory allocation. Consequently, jobs with varying memory requirements end up competing for remote memory allocation simultaneously, resulting in some of them failing to allocate memory. These jobs are then terminated and requeued for later execution, resulting in a higher rate of failures and a decrease in the overall performance of the system.

8.4.9 Sharing of Memory Capacity in the System

Figure 8.12 summarizes the overall percentage of nodes sharing memory capacity over time in the system (y-axis) as a function of the total system memory capacity (x-axis). Each panel represents different job mixes submitted to the system, considering the baseline execution case of +0% overestimation. We can observe that increasing the memory capacity in the system (going to the right on the x-axis) will decrease the percentage of nodes sharing memory, as the system will have more provisioned nodes with enough resources to satisfy the job's demands. Jobs will not need to go remotely to allocate memory capacity, therefore the system will have better overall performance. However, whenever the jobs are scheduled to less provisioned nodes in any system, they will borrow memory capacity from remote nodes.

The static approach consistently has a higher percentage of nodes sharing memory capacity than the dynamic approach. As an example, for all executions of the job large mix 100% (bottom row and rightmost panel), the static approach has a median of 31% of the nodes sharing memory, while for the dynamic approach, this percentage is 18.8%. It translates to more than 40% fewer nodes sharing memory when the system uses the dynamic approach. The highest percentage achieved is on the most underprovisioned system (28% total memory) with a median of 59.4% for the static and 39.8% for the dynamic approach. This difference is due to the fact that once the static approach allocates memory to a job, it will be retained throughout its execution and consequently, other jobs will have to share capacity in order to execute. On the other hand, using the dynamic approach the system will release unused memory capacity from jobs during their low usage period and likely share most of the capacity during their high usage period.

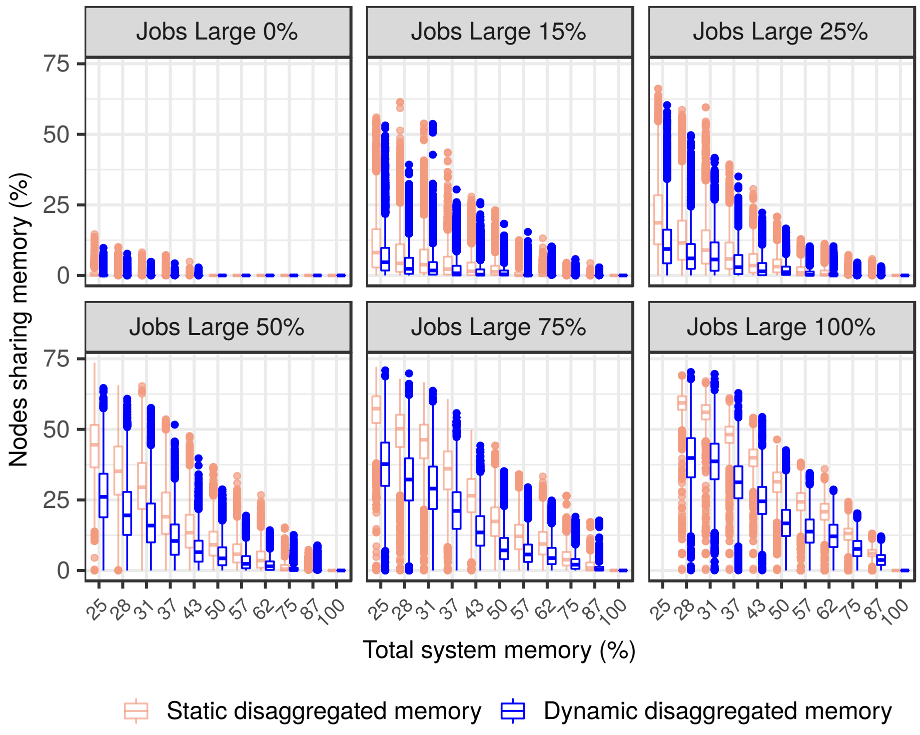


Figure 8.12 Percentage of nodes in the system sharing memory capacity (y-axis) as a function of each memory configuration (x-axis) considering different jobs large mix (+0% overestimation). The static approach has a higher percentage of sharing resources.

8.5 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, so memory resources management systems require the user to specify the job's peak memory demand. According to the memory demand specified at submission time, the allocated memory will remain unchanged throughout the job's entire execution.

This Chapter makes a case for a dynamic approach, which adapts the current allocation to the actual memory usage and reclaims much of the overallocated disaggregated memory. Thus improving throughput and waiting time, and increasing throughput per dollar by up to 38%. Furthermore, the presented approach also reduces the need for the user to provide an accurate bound on the memory footprint. Our experiments are based on publicly available traces, and all source code is available open source in the hope that others reproduce and build upon our work.

Part IV:

Epilogue

Hoje longe, muitas légua
Numa triste solidão
Espero a chuva cair de novo
Pra mim vortar' pro meu sertão
Luiz Gonzaga

CHAPTER 9

Conclusion

THIS Chapter summarizes the discussions and accomplishments achieved in this thesis. We began this thesis by addressing a generic approach to predict performance degradation due to the sharing of resources. The emerging proposal of a novel disaggregated memory architecture to allow a flexible and fine–grained allocation of memory capacity to compute nodes shifts the focus to sharing capacity, rather than coherent sharing of data as in the traditional shared memory processors. Sharing common memory devices or interfaces may incur an unsatisfactory loss of performance because concurrent memory access requests can saturate the components.

Part II of this thesis presented a Slowdown based methodology to build a contention model to predict the performance degradation that results from contention in remote memory access. We enhanced the methodology by adding the concepts of smoothing and read/write memory access ratio to create the correct sensitivity curve, in order to increase the accuracy and similarity with real executions. Using the characterization of an application's sensitivity to contentious pressure from remote access to the memory subsystem, we were able to predict an application's performance in a pairwise execution with 1.19% prediction error on average and 14.6% in the worst case.

In Part III of this thesis we explored disaggregated–memory–aware in the context of HPC resource management. In this direction, we proposed an extension to the Slurm resource manager to allocate memory capacity to jobs in a disaggregated memory system and its evaluation at scale. Since research in job scheduling requires a simulation platform that is both faster and less intrusive than running on a real system, we also extended an existing Slurm simulator to support disaggregated memories and to account for memory bandwidth contention in disaggregated memory leveraging our developed Slowdown based method. Our results showed 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.

In sequence, we dove into the effects of memory demands on the system's overall throughput, response times, and efficiency. It is known that users must express beforehand their resource demands when submitting jobs to HPC systems. The request will be based on users' knowledge or ability to estimate the resources necessary to run the job. Consequently, they will likely overestimate their demands to avoid having the job killed by running out of resources. According to various assumptions on the user's ability to predict memory consumption, we found that even when there is a large effect on system throughput (-25%) and response time (5× higher), there is a very little direct incentive for the users to be accurate in their estimates, with only an 8% increase in response time.

We further demonstrated that dynamic memory assignment delivers improvements up to 18% in throughput, 38% in throughput per dollar, and up to 69% reduction in job response time (median), compared to a static policy, when there are imbalanced memory usage and overestimated demands on underprovisioned systems. Adapting the actual memory usage reduces the need for the user to provide an accurate bound on the memory footprint, therefore increasing the throughput and efficiency of the system.

9.1 Future Work

Disaggregated systems have been proposed to become the standard method to build more efficient HPC infrastructures to achieve higher exascale performances. Although in its infancy, simulations and analysis around this new architecture are important to better understand disruptive architectural changes on future systems and to guide future researches on design space exploration. The analysis carried out in this thesis will become a valuable starting point for other studies as disaggregated systems continue to evolve. The remainder of this Section describes some interesting ideas and proposes new paths that would enhance the findings in this thesis.

Slowdown Based Methodology — Interesting avenues to build on this work in the future include taking account of the multiple execution phases in a single run of an application and reducing the need to profile them in advance. During the development and execution of the methodology, we acknowledge the applications go through different phases during a single run, which incurs dynamically varying levels of contention. However, we use makespan as a metric of interest, thus it allows us to analyze the problem on per–workload "average" granularity. Another enhancement to the methodology would be upgrading the contention model to account for other sources of delay, i.e. networking and distance-dependent latency, that impact the application's performance when accessing the disaggregated memory thus making the model more precise.

Additionally, for profiling unknown applications it could be possible during the execution of the target application with an unknown interfering application, to pause the target application for a short time to collect the performance counters of the interfering application on–the–fly and calculate its remote memory bandwidth. At that point, we would be able to predict the performance of the target application. However, this approach would have to be dynamically repeated throughout the execution to account for the different phases of the interfering application.

It is also important to point out that the slowdown method is only used for validation in our methodology. Its adaptation to a new architecture would be considerably straightforward since the complexity of the methodology scales linearly with the number of applications (as mentioned in Chapter 4). Furthermore, the numbers used in our model could also be calibrated to reflect the performance of the applications on the new system whenever the platform changes.

Disaggregated Memory Policies — A lot of research work around the RJMS has been carried out to solve problems related to the current state–of–art on HPC systems and their evolving memory hierarchy, but disaggregated systems are still under development. Moreover, allocation policies are not well understood in these novel disaggregated memory systems, making it important to investigate them to discover the best algorithm or heuristics to be employed for both performance and efficiency. In this context, it could be interesting to perform a broader design exploration and an evaluation at scale for new scheduling and allocation policies considering different types of components and layouts in the memory subsystems and the impact of their access speed on the application performance.

Pricing Schemes and System Cost — Access to large-scale HPC infrastructures usually requires the submission of proposals to undergo a peer-review process describing computational resources, and usually, resource usage is charged based on core-hours. From an operator's point of view, an interesting analysis would be analyzing fair pricing models to access disaggregated systems considering the contention experienced by the users due to the sharing of resources and the system's overall performance. As in cloud service providers, billing usage in HPC centers could follow the concepts of on-demand resource provisioning to reduce investment. We also envisage that the simulations are a very interesting approach to aid procurement decisions when building large HPC systems. It would be used to realize a more in-depth analyze of architectural options and decide what would be the best cost-benefit infrastructure to be deployed for the workload characteristics running in the center.

Job Trace Methodology — The study and design of computer systems require good models of the workload because it has a large effect on the observed performance. Therefore, realistic workloads are crucial to determine performance in practice. The need for realistic workloads is important in evaluating supercomputers because they are very expensive, therefore it is rarely an option to conduct extensive experiments in production. For our analysis, we presented a methodology to generate and use synthetic traces as well as the memory profile of existing systems. Usually profiling logs are sensitive data for HPC and research centers, so it could be interesting providing new workload models that would make possible the creation of new workload traces that are statistically similar to the memory profile of common applications that run on existing HPC systems. More detailed memory profile traces would be interesting to analyze the effects of memory hierarchic and memory demands on the designed infrastructures.

Dynamic Monitor Module — System's continuous measurement is an important aspect of an HPC infrastructure to detect and solve problems. Monitoring the job's performance using hardware performance counters with negligible overhead allows the operator or system software to correctly assess the system's state and step in for efficient resource utilization. Frameworks as LDMS [78] already demonstrated low overhead and good scalability to thousands of nodes with sampling intervals on the order of seconds. In our envisaged scenario, the agent on each node not only monitors the state of the application but also intercepts memory allocations beyond the actual node allocation. It intercepts and forwards the info with low latency to the controller which is able to expand the job's allocated memory capacity. Interesting avenues in this direction are analyzing the impact of the granularity in which the controller allocates the memory to maximize job and system's efficiency.

In summary, this thesis has demonstrated a methodology to build a contention model to predict performance degradation due to the interference of memory bandwidth for single and multi node applications and an evaluation at scale of a disaggregated–memory–aware job scheduler. We believe our study provides valuable insights into the importance of design space exploration for disaggregated memory HPC systems. We demonstrate that by understanding disruptive architectural changes on future systems and the demands of the workloads, system provisioning can be carefully designed to achieve the best cost–benefit.

APPENDIX A

Sensitivity Curves

In this Appendix, we present the sensitivity curves for all single and multi node applications profiled in Part II of this dissertation. Specifically, Part II of this thesis focused on presenting the performance prediction methodology to build the contention model used in our simulations. In this regard, we introduced a Slowdown based method that correlates interfering memory bandwidth and the application's performance. We also showed that contention is sensitive to the ratio between read and write memory access.

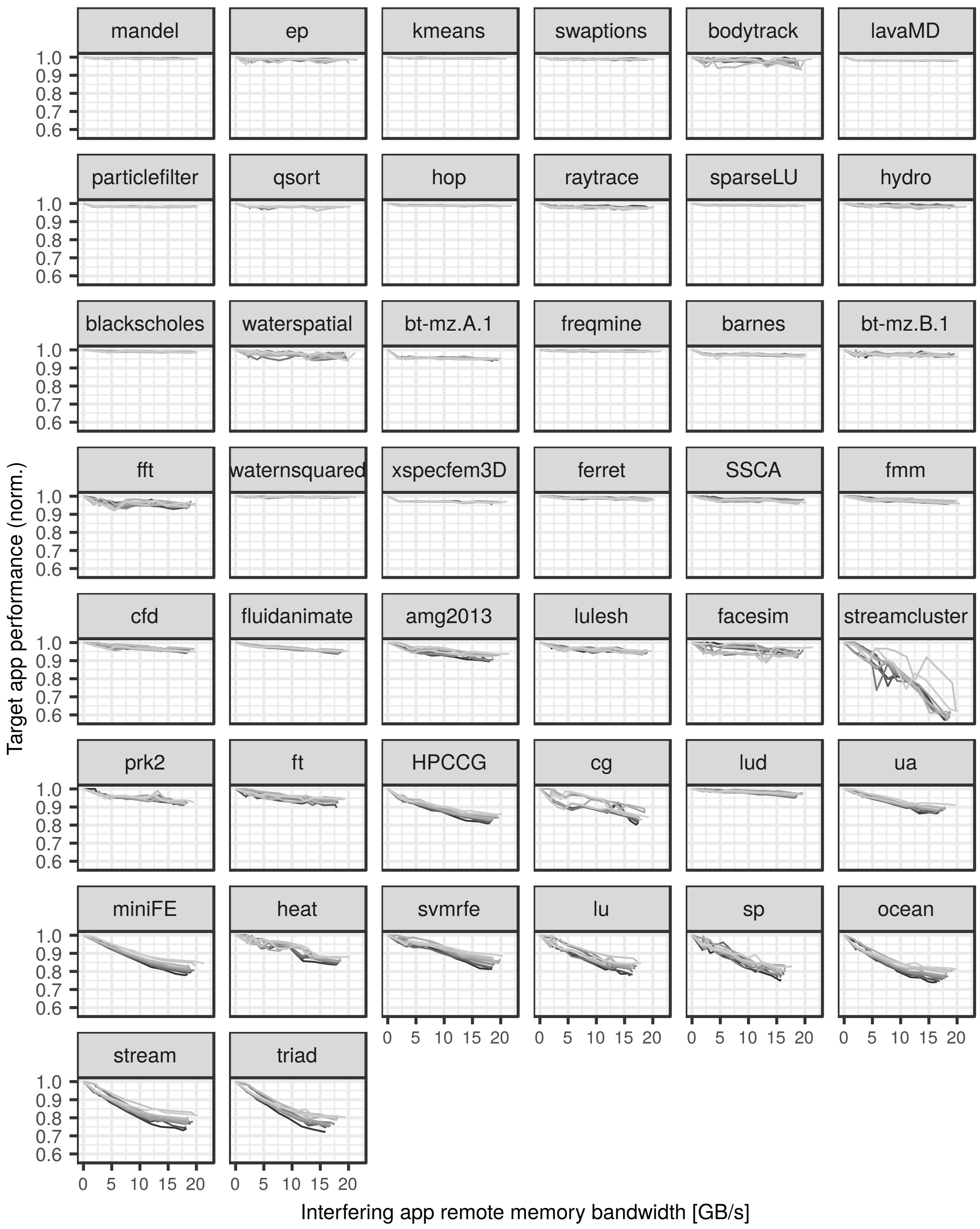


Figure 1.1 Sensitivity curve for all single node applications profiled in this thesis. Each line corresponds to a specific read/write ratio going from 50% Read to 100% Read, with the lighter lines indicating a higher percentage of read.

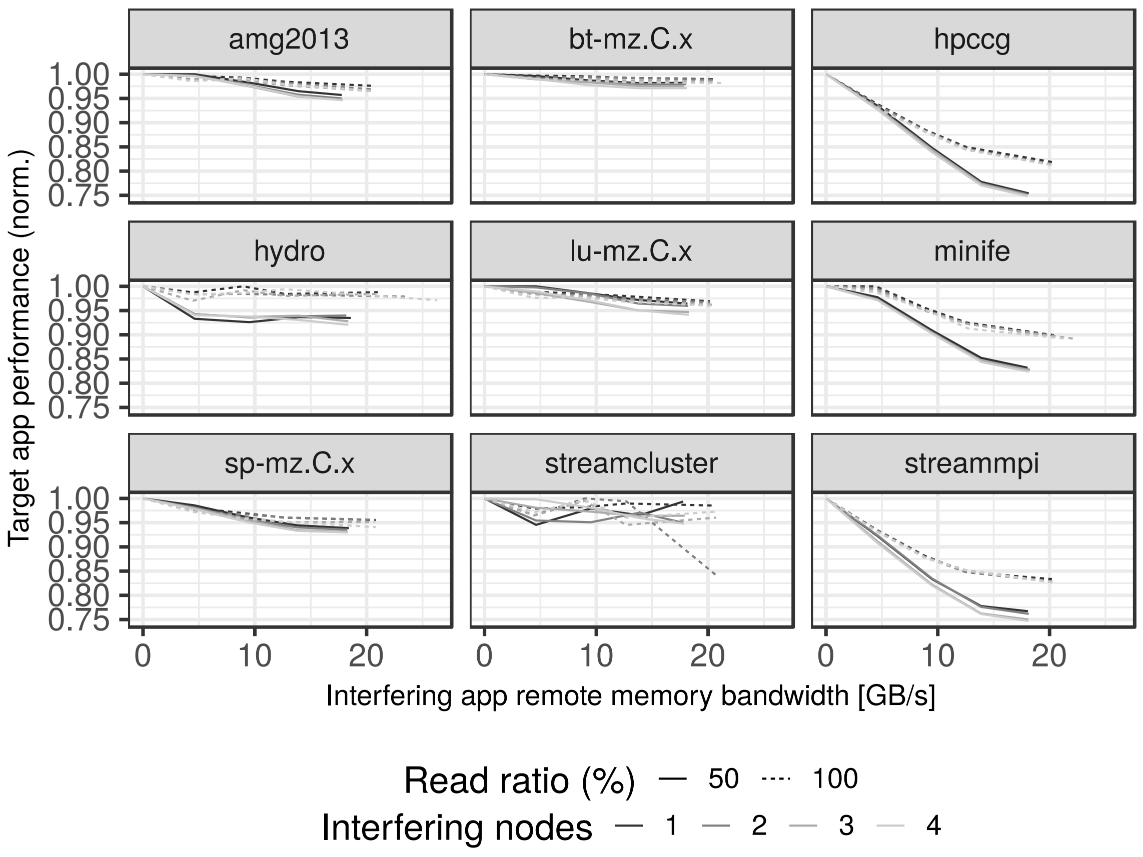


Figure 1.2 Sensitivity curve for all multi node applications profiled in this thesis. The applications run using 4 nodes.

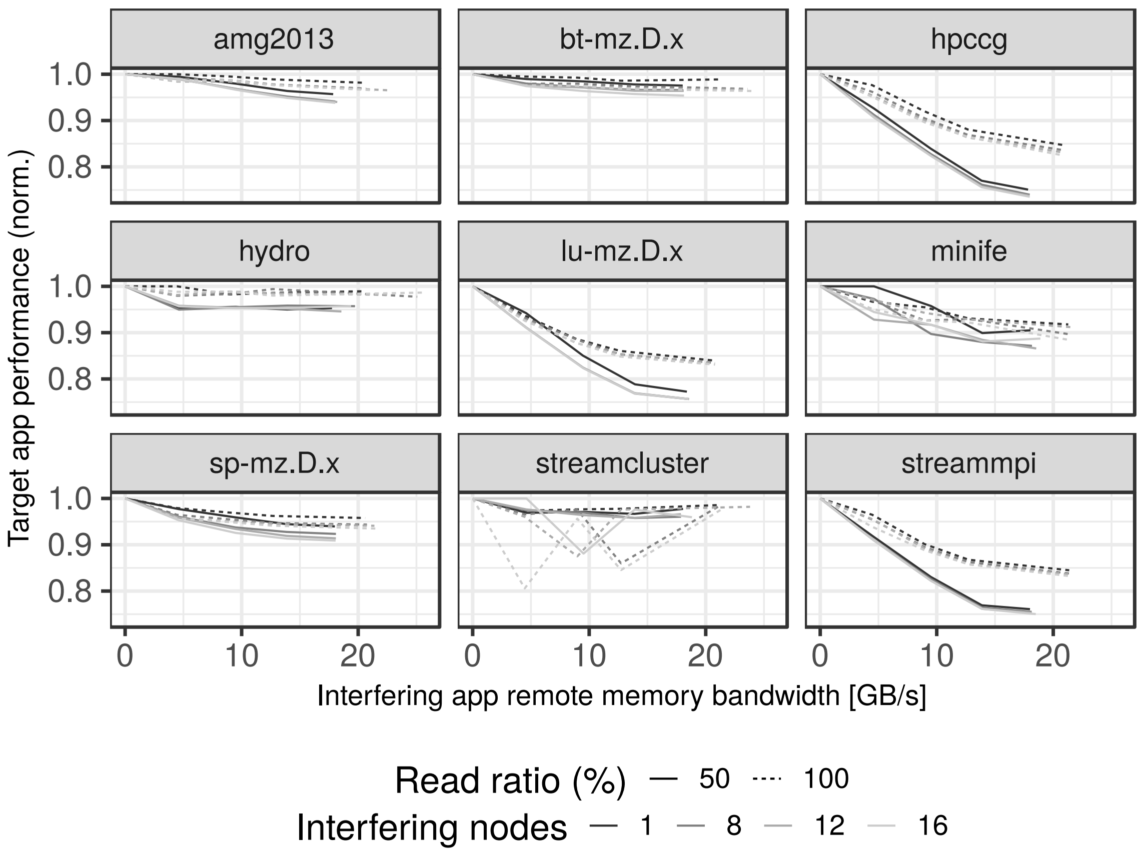


Figure 1.3 Sensitivity curve for all multi node applications profiled in this thesis. The applications run using 16 nodes.

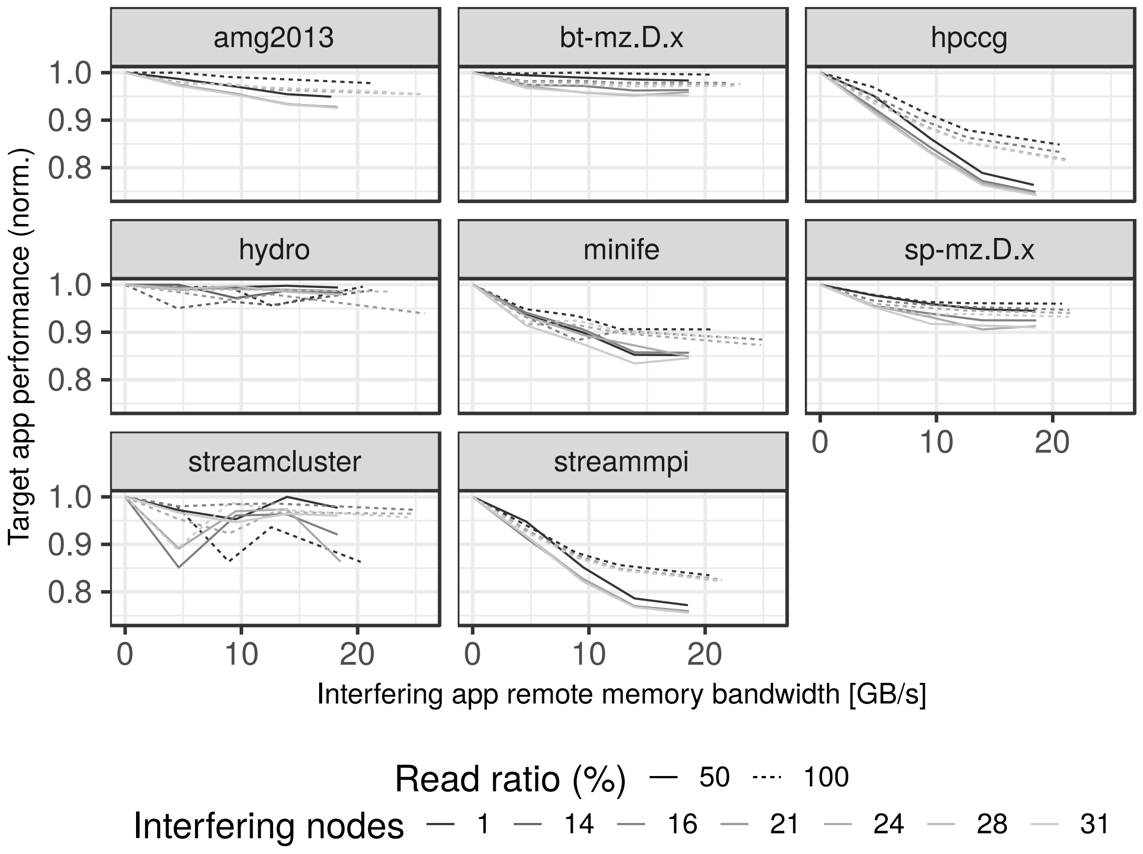


Figure 1.4 Sensitivity curve for all multi node applications profiled in this thesis. The applications run using 31 nodes. Hydro is the only application that runs using 28 nodes.

Part V:

Bibliography

Tchau Tchau amor, Eu vou embora
Tchau tchau amor, Chegou a hora
De me despedir, E dizer adeus

Lairton

References

[1] Antonios D Papaioannou, Reza Nejabati, and Dimitra Simeonidou. The benefits of a disaggregated data centre: A resource allocation approach. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2016.

[2] Vamsee Reddy Kommareddy, Amro Awad, Clayton Hughes, and Simon David Hammond. Exploring allocation policies in disaggregated non-volatile memories. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, pages 58–66. ACM, 2018.

[3] Carlos Vega, Jose Fernando Zazo, Hugo Meyer, Ferad Zyulkyarov, Sergio López-Buedo, and Javier Aracil. Diluting the scalability boundaries: Exploring the use of disaggregated architectures for high-level network data analysis. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 340–347. IEEE, 2017.

[4] Mozhgan Mahloo, João Monteiro Soares, and Amir Roozbeh. Techno-economic framework for cloud infrastructure: A cost study of resource disaggregation. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 733–742. IEEE, 2017.

[5] Howraa Mehdi Mohammad Ali, Taisir EH El-Gorashi, Ahmed Q Lawey, and Jaafar MH Elmirghani. Future energy efficient data centers with disaggregated servers. *Journal of Lightwave Technology*, 35(24):5361–5380, 2017.

[6] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. Main memory in HPC: Do we need more or could we live with less? *ACM Trans. Archit. Code Optim.*, 14(1):3:1–3:26, March 2017.

[7] Rajiv Nishtala, Paul Carpenter, and Xavier Martorell. Performance effects on HPC workloads of global memory capacity sharing. In *MULTIPROG*, 01 2019.

[8] Georgios Zervas, Hui Yuan, Arsalan Saljoghei, Qianqiao Chen, and Vaibhawa Mishra. Optically disaggregated data centers with minimal remote memory

latency: technologies, architectures, and resource allocation. *Journal of Optical Communications and Networking*, 10(2):A270–A285, 2018.

[9] Bulent Abali, Richard J Eickemeyer, Hubertus Franke, Chung-Sheng Li, and Marc A Taubenblatt. Disaggregated and optically interconnected memory: when will it be cost effective? *arXiv preprint arXiv:1503.01416*, 2015.

[10] Ivy Peng, Roger Pearce, and Maya Gokhale. On the memory underutilization: Exploring disaggregated memory on hpc systems. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 183–190. IEEE, 2020.

[11] IB Peng, I Karlin, MB Gokhale, K Shoga, M Legendre, and T Gamblin. A holistic view of memory utilization on hpc systems: Current and future trends. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2021.

[12] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, 2016.

[13] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, Ilias Syrigos, Maciej Bielski, Dimitris Theodoropoulos, Dionisios N Pnevmatikatos, and Georgios Zervas. A software-defined architecture and prototype for disaggregated memory rack scale systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 300–307. IEEE, 2017.

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[15] A Saljoghei, V Mishra, M Bielski, I Syrigos, K Katrinis, D Syrivelis, A Reale, DN Pnevmatikatos, D Theodoropoulos, M Enrico, et al. dReDbox: Demonstrating disaggregated memory in an optical data centre. In *2018 Optical Fiber Communications Conference and Exposition (OFC)*, pages 1–3. IEEE, 2018.

[16] Yves Durand, Paul M Carpenter, Stefano Adami, Angelos Bilas, Denis Dutoit, Alexis Farcy, Georgi Gaydadjiev, John Goodacre, Manolis Katevenis, Manolis Marazakis, et al. Euroserver: Energy efficient node for european micro-servers. In *2014 17th Euromicro Conference on Digital System Design*, pages 206–213. IEEE, 2014.

[17] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, and Christian Pinto. A software-defined SoC memory bus bridge architecture for disaggregated computing. In

Proceedings of the 3rd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems, page 3. ACM, 2018.

[18] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[19] Héctor Montaner, Federico Silla, Holger Froning, and José Duato. Memscale™: A scalable environment for databases. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 339–346. IEEE, 2011.

[20] Pramod Subba Rao and George Porter. Is memory disaggregation feasible?: A case study with Spark SQL. In *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, pages 75–80. ACM, 2016.

[21] Dhantu Buragohain, Abhishek Ghogare, Trishal Patel, Mythili Vutukuru, and Purushottam Kulkarni. Dime: A performance emulator for disaggregated memory architectures. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–8, 2017.

[22] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[23] Alvise Rigo, Christian Pinto, Kevin Pouget, Daniel Raho, Denis Dutoit, Pierre-Yves Martinez, Chris Doran, Luca Benini, Iakovos Mavroidis, Manolis Marazakis, et al. 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)*, pages 486–493. IEEE, 2017.

[24] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *JSSPP*, pages 44–60. Springer, 2003.

[25] Ana Jokanovic, Marco D'Amico, and Julita Corbalan. Evaluating slurm simulator with real-machine slurm and vice versa. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 72–82. IEEE, 2018.

[26] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 267–278. ACM, 2009.

[27] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, pages 248–259. ACM, 2011.

[28] Andreas De Blanche and Thomas 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.

[29] David Eklöv, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth Bandit: Quantitative characterization of memory contention. In *CGO 2013, 23-27 February, Shenzhen, China*, pages 99–108. IEEE Computer Society, 2013.

[30] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access,high-performance memory disaggregation with directcxl. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[31] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H Peter Hofstee. Thymesisflow: a software-defined, hw/sw co-designed interconnect stack for rack-scale memory disaggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 868–880. IEEE, 2020.

[32] Panos Koutsovasilis, Michele Gazzetti, and Christian Pinto. 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)*, pages 576–585. IEEE, 2021.

[33] Marco D'Amico, Ana Jokanovic, and Julita Corbalan. Holistic slowdown driven scheduling and resource management for malleable jobs. In *Proceedings of the 48th International Conference on Parallel Processing*, page 31. ACM, 2019.

[34] Sergio Iserte, Rafael Mayo, Enrique S Quintana-Ortí, Vicenç Beltran, and Antonio J Peña. Efficient scalable computing through flexible applications and adaptive workloads. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 180–189. IEEE, 2017.

[35] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh, Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf. A case for intra-rack resource disaggregation in hpc. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(2):1–26, 2022.

[36] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[37] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. 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)*, pages 1–8. 2019.

[38] Mohammed Tanash, Huichen Yang, Daniel Andresen, and William Hsu. Ensemble prediction of job resources to improve system performance for slurm-based hpc systems. In *Practice and Experience in Advanced Research Computing*, pages 1–8. 2021.

[39] Felippe Vieira Zacarias, Rajiv Nishtala, and Paul Carpenter. Contention-aware application performance prediction for disaggregated memory systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 49–59, 2020.

[40] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. Improving hpc system throughput and response time using memory disaggregation. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2021.

[41] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. Memory demands in disaggregated HPC: How accurate do we need to be? In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–6, Nov 2021.

[42] Felippe Vieira Zacarias, Paul Carpenter, and Vinicius Petrucci. Dynamic memory provisioning on disaggregated hpc systems. [manuscript submitted for publication]. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023.

[43] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. Topology-aware resource management for hpc applications. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–10, 2017.

[44] Yiannis Georgiou and Matthieu Hautreux. Evaluating scalability and efficiency of the resource and job management system on large hpc clusters. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 134–156. Springer, 2012.

[45] Bo Wang, Zhiguang Chen, and Nong Xiao. A survey of system scheduling for hpc and big data. In *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, pages 178–183, 2020.

[46] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scalable system scheduling for hpc and big data. *Journal of Parallel and Distributed Computing*, 111:76–92, 2018.

[47] Robert L Henderson. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer, 1995.

[48] Brett Bode, David M Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *Annual Linux Showcase & Conference*, 2000.

[49] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Software: practice and Experience*, 23(12):1305–1336, 1993.

[50] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford, and Joseph F Skovira. Workload management with loadleveler. *IBM Redbooks*, 2(2):58, 2001.

[51] Moab adaptive hpc suite. http://www.adaptivecomputing.com, 2022. Accessed: 2022-01-20.

[52] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[53] Pranav Joshi and Muda Rajesh Babu. Openlava: An open source scheduler for high performance computing. In *2016 International Conference on Research Advances in Integrated Navigation Systems (RAINS)*, pages 1–3. IEEE, 2016.

[54] SchedMD. Schedmd slurm development and supportg. https://slurm.schedmd.com/overview.html, 2021. Accessed: 2021-01-21.

[55] Changyeon Jo, Hyunik Kim, Hexiang Geng, and Bernhard Egger. Rackmem: a tailored caching layer for rack scale computing. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 467–480, 2020.

[56] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. Quantifying memory underutilization in HPC systems and using it to improve performance via architecture support. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 821–835, 2019.

[57] BSC slurm simulator. https://github.com/BSC-RM/slurm_simulator, 2021. Accessed: 2021-01-20.

[58] Steve J Chapin, Walfredo Cirne, Dror G Feitelson, James Patton Jones, Scott T Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer, 1999.

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

[60] Logs of real parallel workloads from production systems. https://www.cse.huji.ac.il/labs/parallel/workload/logs.html. Accessed: 2021-01-20.

[61] SGI Altix UV 1000 Datasheet. https://cutt.ly/McVBAFS, 2021. Accessed: 2021-01-21.

[62] Bull Coherent Switch (BCS). https://cutt.ly/IcVVZt3, 2021. Accessed: 2021-01-21.

[63] Maciej Bielski, Ilias Syrigos, Kostas Katrinis, Dimitris Syrivelis, Andrea Reale, Dimitris Theodoropoulos, Nikolaos Alachiotis, D Pnevmatikatos, EH Pap, George Zervas, et al. dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1093–1098. IEEE, 2018.

[64] EuroEXA project. H2020 project number 754337, 2009. Accessed: 2021-09-20.

[65] Stephen Van Doren. Hoti 2019: compute express link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18. IEEE, 2019.

[66] Debendra Das Sharma and Siamak Tavallaei. Compute express link 2.0 white paper. *Tech. Rep.*, 2020.

[67] Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/, 2022. Accessed: 2022-01-20.

[68] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[69] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[70] Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*, pages 140–148. IEEE, 2001.

[71] Walfredo Cirne and Francine Berman. Using moldability to improve the performance of supercomputer jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002.

[72] Li Ruan, Xiangrong Xu, Limin Xiao, Feng Yuan, Yin Li, and Dong Dai. A comparative study of large-scale cluster workload traces via multiview analysis. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 397–404. IEEE, 2019.

[73] Dror G Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982, 2014.

[74] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems*, pages 1–14, 2020.

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

[76] Memory statistics from open clusters - la-ur-19-28211. https://usrc.lanl.gov/data/LA-UR-19-28211.php, 2022. Accessed: 2022-01-20.

[77] Lanl cts-1 grizzly - tundra extreme scale, xeon e5-2695v4 18c 2.1ghz, intel omnipath. https://www.top500.org/system/178972, 2022. Accessed: 2022-01-20.

[78] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. IEEE, 2014.

[79] Andreas De Blanche and Thomas Lundqvist. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing*, 71(4):1451–1483, 2015.

[80] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH*, volume 41, pages 607–618. ACM, 2013.

[81] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Cache pirating: Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175. IEEE, 2011.

[82] Jiacheng Zhao, Huimin Cui, Jingling Xue, Xiaobing Feng, Youliang Yan, and Wensen Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 201–212. IEEE Press, 2013.

[83] Jiacheng Zhao, Huimin Cui, Jingling Xue, and Xiaobing Feng. Predicting cross-core performance interference on multicore processors with regression analysis. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1443–1456, 2015.

[84] Marc Casas and Greg Bronevetsky. Evaluation of HPC applications' memory resource consumption via active measurement. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2560–2573, 2015.

[85] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650. IEEE, 2013.

[86] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 62–75. ACM, 2015.

[87] Dongliang Xiong, Kai Huang, Xiaowen Jiang, and Xiaolang Yan. Providing predictable performance via a slowdown estimation model. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):25, 2017.

[88] Yogesh D Barve, Shashank Shekhar, Ajay Dev Chhokra, Shweta Khare, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun, and Aniruddha Gokhale. Fecbench: A holistic interference-aware approach for application performance modeling. *arXiv preprint arXiv:1904.05833*, 2019.

[89] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 23–33. IEEE, 2013.

[90] Hao Xu, Shasha Wen, Alfredo Gimenez, Todd Gamblin, and Xu Liu. DR-BW: identifying bandwidth contention in numa architectures with supervised learning. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 367–376. IEEE, 2017.

[91] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E Nagel. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 27–38. ACM, 2017.

[92] Giovanni Farias, Francisco Brasileiro, Raquel Lopes, Marcus Carvalho, Fábio Morais, and Daniel Turull. On the efficiency gains of using disaggregated hardware to build warehouse-scale clusters. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 239–246. IEEE, 2017.

[93] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. Dilos: adding performance to paging-based memory disaggregation. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 70–78, 2021.

[94] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[95] Wenqi Cao and Ling Liu. Dynamic and transparent memory sharing for accelerating big data analytics workloads in virtualized cloud. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 191–200. IEEE, 2018.

[96] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2022.

[97] Nikolaos D Kallimanis, Manolis Marazakis, and Manolis Skordalakis. Use-cases for remote memory in the unimem architecture. In *ExascaleHPC: the ExaNoDe, ExaNeSt, EcoScale, and EuroEXA projects workshop at HiPEAC, Manchester*, 2018.

[98] William Allcock, Bennett Bernardoni, Colleen Bertoni, Neil Getty, Joseph Insley, Michael E Papka, Silvio Rizzi, and Brian Toonen. Ram as a network managed resource. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 99–106. IEEE, 2018.

[99] Alexandru Uta, Ana-Maria Oprescu, and Thilo Kielmann. Towards resource disaggregation—memory scavenging for scientific workloads. In *2016 IEEE*

International Conference on Cluster Computing (CLUSTER), pages 100–109. IEEE, 2016.

[100] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric hpc system for deep learning. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 148–161. IEEE, 2018.

[101] Qi Zhang, Ling Liu, Calton Pu, Wenqi Cao, and Semih Sahin. Efficient shared memory orchestration towards demand driven memory slicing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1212–1223. IEEE, 2018.

[102] Kwangwon Koh, Kangho Kim, Seunghyub Jeon, and Jaehyuk Huh. Disaggregated cloud memory with elastic block management. *IEEE Transactions on Computers*, 68(1):39–52, 2018.

[103] Luis A Garrido and Paul Carpenter. Aggregating and managing memory across computing nodes in cloud environments. In *International Conference on High Performance Computing*, pages 642–652. Springer, 2017.

[104] Shouwei Chen, Wensheng Wang, Xueyang Wu, Zhen Fan, Kunwu Huang, Peiyu Zhuang, Yue Li, Ivan Rodero, Manish Parashar, and Dennis Weng. Optimizing performance and computing resource management of in-memory big data analytics with disaggregated persistent memory. In *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019*, pages 21–30. Institute of Electrical and Electronics Engineers Inc., 2019.

[105] Marcelo Amaral, Jordà Polo, David Carrera, Nelson Gonzalez, Chih-Chieh Yang, Alessandro Morari, Bruce D'Amora, Alaa Youssef, and Malgorzata Steinder. Drmaestro: orchestrating disaggregated resources on virtualized data-centers. *Journal of Cloud Computing*, 10(1):1–20, 2021.

[106] Sergio Iserte, Rafael Mayo, Enrique S Quintana-Orti, and Antonio J Pena. Dmrlib: Easy-coding and efficient resource management for job malleability. *IEEE Transactions on Computers*, 70(9):1443–1457, 2020.

[107] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 37–48, 2019.

[108] Andrea Borghesi, Andrea Bartolini, Michela Milano, and Luca Benini. Pricing schemes for energy-efficient hpc systems: Design and exploration. *The International Journal of High Performance Computing Applications*, 33(4):716–734, 2019.

[109] Partnership for Advanced Computing in Europe. https://prace-ri.eu/about/introduction/, 2021. Accessed: 2021-01-20.

[110] Extreme science and engineering discovery environment. www.xsede.org, 2021. Accessed: 2021-01-20.

[111] Innovative and novel computational impact on theory and experiment. https://www.doeleadershipcomputing.org/proposal/call-for-proposals/, 2021. Accessed: 2021-01-20.

[112] Alex D Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. Enabling fair pricing on hpc systems with node sharing. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, pages 1–12, 2013.

[113] Artan Mazrekaj, Isak Shabani, and Besmir Sejdiu. Pricing schemes in cloud computing: an overview. *International Journal of Advanced Computer Science and Applications*, 7(2):80–86, 2016.

[114] Aaqif Afzaal Abbasi, Almas Abbasi, Shahaboddin Shamshirband, Anthony Theodore Chronopoulos, Valerio Persico, and Antonio Pescapè. Software-defined cloud computing: A systematic review on latest trends and developments. *IEEE Access*, 7:93294–93314, 2019.

[115] Opeyemi O Ajibola, Taisir EH El-Gorashi, and Jaafar MH Elmirghani. Energy efficient placement of workloads in composable data center networks. *Journal of Lightwave Technology*, 39(10):3037–3063, 2021.

[116] Sulav Malla and Ken Christensen. Hpc in the cloud: Performance comparison of function as a service (faas) vs infrastructure as a service (iaas). *Internet Technology Letters*, 3(1):e137, 2020.

[117] Yang Lu, Xianrong Zheng, Ling Li, and Li D Xu. Pricing the cloud: a qos-based auction approach. *Enterprise Information Systems*, 14(3):334–351, 2020.

[118] Marco Ferretti and Luigi Santangelo. Cloud vs on-premise hpc: a model for comprehensive cost assessment. *Parallel Computing: Technology Trends*, 36:69, 2020.

[119] Anna Pupykina and Giovanni Agosta. Survey of memory management techniques for hpc and cloud computing. *IEEE Access*, 7:167351–167373, 2019.

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

[121] Daniel Molka, Daniel Hackenberg, and Robert Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, pages 1–10, 2014.

[122] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81. ACM, 2008.

[123] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, Washington, DC, USA, 2009. IEEE.

[124] 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*, pages 158–165, New York, NY, USA, 1991. ACM.

[125] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *ISPASS*, pages 101–111. IEEE, 2016.

[126] Andi Kleen. A numa api for linux. *Novel Inc*, 2005.

[127] Arnaldo Carvalho de Melo. The new linux'perf'tools.

[128] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[129] Milan Radulovic, Rommel Sánchez Verdejo, Paul Carpenter, Petar Radojkovic, Bruce Jacob, and Eduard Ayguadé. PROFET: Modeling system performance and energy without simulating the CPU. In *Abstracts of the 2019 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '19, pages 71–72, New York, NY, USA, 2019. ACM.

[130] BSC. Profet: Code for generating memory bandwidth load, for different read traffic ratios and bandwidth intensity., 2019. Accessed: 2019-10-16.

[131] F.V. Zacarias, V. Petrucci, R. Nishtala, P. Carpenter, and D. Mossé. Intelligent colocation of workloads for enhanced server efficiency. In *2019 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2019.

[132] Josué Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. Perf&fair: A progress-aware scheduler to enhance performance and fairness in SMT multicores. *IEEE Transactions on Computers*, 66(5):905–911, 2016.

[133] Tirtha Pratim Bhattacharjee. *Data Movement and Workload characterization: Intel Sandy Bridge Core and Uncore PMU features*, 2013.

[134] Zoltan Majo and Thomas R Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, page 12. ACM, 2011.

[135] John D. McCalpin. Sc16 invited talk: Memory bandwidth and system balance in hpc systems. https://sites.utexas.edu/jdm4372/tag/stream-benchmark/, 2019. Accessed: 2019-09-18.

[136] Intel Corporation. *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, 2018.

[137] Emmanuel Kayode Akinshola Ogunshile. Viability of small-scale HPC cloud infrastructures. In *CLOSER*, pages 275–286, 2018.

[138] Konstantin S Solnushkin. Saddle: A modular design automation framework for cluster supercomputers and data centres. In *International Supercomputing Conference*, pages 232–244. Springer, 2014.

[139] Konstantin S. Solnushkin. Automated design of torus networks. *CoRR*, abs/1301.6180, 2013.

[140] Andy Turner and Simon McIntosh-Smith. 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*, pages 250–260. Springer, 2017.

[141] Google cloud adds compute, memory-intensive vms. https://www.sdxcentral.com/articles/news/google-cloud-adds-compute-memory-intensive-vms/2019/08/, 2022. Accessed: 2021-03-22.

[142] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, 1(3):244–256, 1972.

[143] David H Douglas and Thomas K Peucker. 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*, 10(2):112–122, 1973.

[144] Disaggregated memory slurm simulator and allocation policy. https://github.com/felippezacarias/slurm_simulator, 2021. Accessed: 2021-04-08.

BIOGRAPHICAL SKETCH

Felippe Vieira Zacarias was born in Pojuca, Brazil. He received a Bachelor's degree in Information Systems from the State University of Bahia, Brazil in 2014 and a Master's degree in Computer Science from the Federal University of Bahia, Brazil in 2018. Worked at SENAI CIMATEC Supercomputing Center developing technical-scientific research on HPC. He also provided consultancy in innovation projects using HPC technologies.

He began his doctoral studies at Universitat Politècnica de Catalunya (UPC) and Barcelona Supercomputing Center (BSC), Spain where he worked under the graceful supervision of Paul Carpenter (Senior Researcher), Vinícius Petrucci and Xavier Martorell (Professor). His current research interests include high-performance computing, cluster and disaggregated architectures, memory subsystem, and performance evaluation.

Felippe Vieira Zacariasis expected to receive a Doctor of Philosophy degree in Computer Architecture in July 2023.