Delft University of Technology
Master of Science Thesis in Embedded System

Optimizing the Cost and Performance for Batch jobs on HPC by Utilizing Swap Space

Hanzhang Lin

TU Delft Delft University of Technology

Optimizing the Cost and Performance for Batch jobs on HPC by Utilizing Swap Space

Master of Science Thesis in Embedded Systems

Embedded Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

Hanzhang Lin
H.Lin-13@student.tudelft.nl
linhanzhang_07@outlook.com

August 28th, 2023

Author
Hanzhang Lin (H.Lin-13@student.tudelft.nl)
(linhanzhang_07@outlook.com)

Title
Optimizing the Cost and Performance for Batch jobs on HPC by Utilizing Swap Space

MSc Presentation Date
Monday August 28th, 2023

Graduation Committee

Prof. dr. Koen Langendoen (Chair)	Delft University of Technology
Dr. Johan Pouwelse	Delft University of Technology
Ir. Michel Roelofs	NXP Semiconductors

Abstract

High Performance Computing (HPC) facilities play a crucial role in accelerating Electronic Design Automation (EDA) procedures at NXP Semiconductors. The increasing number of job requests and workloads has led to a surge in memory demand, which is a costly resource. To address this, we leverage memory swap space on disk as a more affordable extension of main memory. By employing a job submit-time parameter called **memory reservation**, we can increase the cluster jobs' parallelism, and control the memory swap space usage.

The primary goal of this thesis is to find the optimal cost and performance by optimizing the **memory reservation** parameter. To achieve this goal, we created an effective parameter optimizer that employs a one-way search method to determine the best **memory reservation**, focusing on scenarios with a large homogeneous job set running on an overloaded system.

Our contributions include improving the memory utilization efficiency in HPC by utilizing swap space. Previous work used Machine Learning models to predict maximum memory usage and reduce over-reservation; and our work further reduces memory usage per job.

The results demonstrate that the parameter optimizer effectively minimizes the cost and performance at the same time, leading to up to 43% cost saving for a job among our test cases.

"Just because we can't find a solution it doesn't mean that there isn't one." –
Andrew Wiles

Preface

The rapid growth of job requests and workloads in Electronic Design Automation (EDA) at NXP Semiconductors has intensified the demand for memory in High Performance Computing (HPC) facilities. As memory is a costly resource, there arises a need for innovative strategies to optimize its usage. In this work, we explored the utilization of memory swap space on disk as a cost-effective extension of main memory.

I would like to thank my supervisors Prof.dr.K.G. Langendoen and Michel Roelofs for their invaluable guidance, unwavering support, and insightful feedback.

Besides, I am deeply grateful to my family for their unwavering love and encouragement throughout my academic journey. Their belief in me has been a constant source of motivation and inspiration.

I would also like to extend my heartfelt thanks to my friends who have always been by my side and help me through the toughest period. Their camaraderie and positivity have enriched my life.

Hanzhang Lin

Delft, The Netherlands
22nd August 2023

Contents

Chapter 1

Introduction

This thesis is done in collaboration with the High Performance Computing (HPC) team at NXP Semiconductors. HPC facilities are able to perform a vast number of complex computations per second, at a relatively low cost. Hence, it is widely used for computationally intensive tasks, including chip design and EDA usage. Due to the increasingly intensive competition in the market, semiconductor manufacturers want to shorten the chip development period and accelerate the time-to-market for new chips. Many semiconductor companies including NXP use HPC to reduce the job wait time and run time so as to enable faster delivery of products to the market. Besides, controlling the cost is also a significant concern for the companies. With lower infrastructure costs, enterprises are able to allocate more resources to research and development, driving innovation.

Currently, NXP primarily offers HPC solutions through on-premise facilities, specifically computer clusters located in their data center. However, there is a gradual shift by the HPC team towards migrating the HPC computing environment to the cloud [22]. This transition enables a more cost-effective, accessible, efficient, and collaborative HPC ecosystem [14]. Unless explicitly stated, all research, analysis, and experiments discussed in this work are conducted within the cloud environment. Compared to on-premise servers, cloud computing offers greater flexibility in resource pricing. HPC operators can rent hardware from cloud service providers for a specified and adaptable duration, paying only for the resources they rent. This flexibility allows for flexible scalability of the cloud cluster based on demand, leading to enhanced resource utilization efficiency and potential cost savings.

NXP's HPC infrastructure is built on the foundation of the job scheduler, a powerful workload management system that can manage and optimize the execution of computing cluster jobs across NXP's computing cluster. The HPC job is the basic unit for the submission, scheduling, and execution of works.

Table 1.1: **Server pricing.**

Server Name	Hourly Rate	CPU cores	Memory
serverA	$ x	64	256GB
serverB	$x+2.624	64	1024GB

The HPC team currently faces the challenge of managing a growing number of job requests and heavier workloads, which significantly increases the pressure on memory. However, memory is a resource that can incur substantial costs. As depicted in Table 1.1, the serverA and serverB, which are two different types of virtual servers provided by a cloud provider, have an identical number of cores. Nevertheless, the additional 768 GB of memory space incurs an extra cost of $2.624 per hour.

In our pursuit of cost-effective solutions, we have turned our attention to memory swapping. Memory swapping is a memory reclamation technique in which inactive memory contents are moved to the swap space on a disk, freeing up memory for other processes or applications. Initially, there were concerns regarding the performance implications of memory swapping. However, with the rapid advancements in storage speed and capacity, coupled with the relatively low cost (as shown in Figure 1.1), it is worth reconsidering the potential benefits of using swap space. Memory swapping can be viewed as an extension of limited memory resources, enabling the Operating System (OS) to accommodate requests that would otherwise exceed the system's capacity. By offloading the least recently used data to the swap space, expensive high-performance physical memory can be preserved for more computationally demanding operations [28]. To accurately assess the impact of memory swapping, we need to measure the latency caused by memory swapping during job execution. Fortunately, Linux provides a functionality named "delay accounting" for measuring the memory swap delay, which represents the duration a job temporarily suspends its execution while waiting for the required memory to be swapped in.

Our HPC system operates on a preconfigured Linux setup, where the OS handles memory-swapping operations transparently to users. Consequently, manipulating the underlying memory-swapping behavior is challenging, given the HPC environment's black box nature. However, there are two cluster job submit-time parameters that offer a controllable and quantifiable approach to managing memory swap usage. The parameter R allows for the reservation of a dedicated amount of memory within the system, effectively controlling the maximum parallelism of cluster jobs running concurrently. When this memory is reserved, it's subtracted from the overall available memory in the system. If there is no sufficient available memory for new job submissions, the system determines how many jobs it can schedule simultaneously. By increasing cluster job parallelism, the system can be moderately overloaded, permitting global control over the total memory to be swapped out. The OS then decides which jobs should undergo memory swapping. *(NOTE: The term "parallelism" in this report refers specifically to the number of cluster jobs concurrently running in the system, as opposed to the number of threads operating in parallel within an individual cluster job.)*

The second parameter, M, empowers users to set an upper limit on the total physical memory usage for each job throughout its entire execution. Any memory usage exceeding this limit (total virtual memory usage minus physical memory usage) gets automatically swapped to the disk's swap space by the OS. For a detailed introduction to these parameters, please refer to Section 2.1.

Based on all the investigations above, we aim to develop a tool capable of automatically determining the best M and R that can lead to the best cost and performance in real-world Research and Development (R&D) processes. To streamline the problem, we concentrate on a scenario where the system memory

is fully utilized, and a substantial batch of jobs with homogeneous workloads needs to be managed. This scenario mirrors a frequent situation in NXP's actual production environment, particularly during extensive parameter sweep tasks. Here, engineers systematically explore a range of values for specific circuit design parameters, a practice that aligns with the requirements of our research.

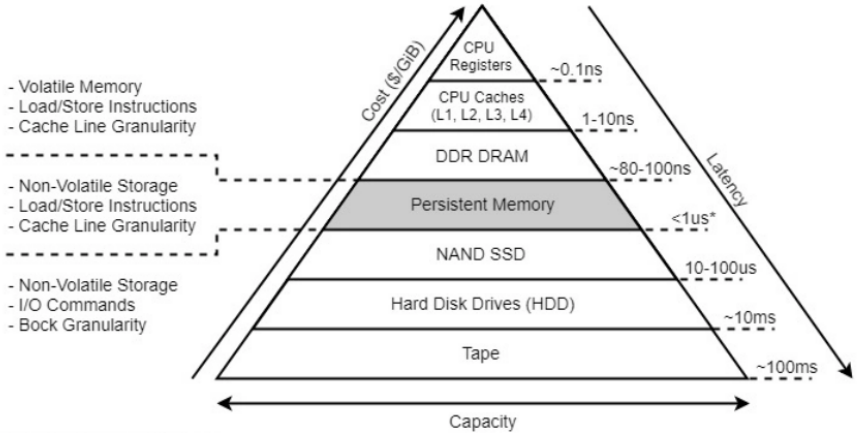


Figure 1.1: **Memory hierarchy of the modern computer. Compared to Dynamic Random Access Memory (DRAM), the disk (Persistent Memory (PMEM), NAND Solid State Drive(SSD), Hard Drive Disk (HDD)) has a larger capacity and lower cost. Traditional storage like HDD has a millisecond latency. PMEM and NAND SSD which are becoming popular these years can reach a microsecond latency, bringing up the performance of storage by a notch. However, there is still a significant speed gap between it and DRAM, which has nanosecond latency [17].**

1.1 Problem Statement

Optimizing resource utilization in HPC has long been a prominent research area, with particular emphasis on memory utilization. Numerous studies have focused on predicting maximum memory usage to optimize job scheduling and enhance overall cluster resource utilization [30] [32] [18] [27] [21]. However, conventional memory usage optimization has primarily concentrated on Random Access Memory (RAM) itself, neglecting the potential benefits of utilizing swap space—an economical extension to RAM.

The main problem statement of this thesis will therefore be as follows:

How to find a balance between using the RAM and the memory swap space based on the delay accounting data collected in HPC environment so that one can realize the optimal cost and performance?

To solve this problem, the problem statement is separated into the following parts:

1. *Build a delay accounting tool and deploy it to one of the HPC clusters to collect delay accounting data over time.* This tool will enable us to measure the impact of memory swapping on job performance by collecting delay accounting statistics for each completed job. Since a job may comprise multiple processes, it is essential to construct a process tree for each job and aggregate the delay data accordingly.

2. *Conduct delay accounting measurements on the most typical circuit simulation jobs to comprehend the impact of swap usage on performance:* In order to understand the effects of memory swapping on performance across different job types and validate the accuracy of our measurements, we will focus on measuring the delay accounting of the most representative circuit simulation jobs for each single and individual job and verify the measurement results. In addition, comparison experiments will be conducted by modifying certain configurations, such as storage type and job parallelism in the system, to gain a comprehensive understanding of the impact.

3. *Compare the performance on different parameter setting schemes on an overloaded system*: Since we are free to choose between the usage of R and M at submit-time. We need to identify which combination leads to the best cost and performance.

4. *Design of an automated Tool to find the best submit-time parameter combination of R and M*: We will design and implement a user-friendly tool capable of identifying the optimal R and M combination leading to the lowest cost and best performance. This tool will employ a search algorithm that iteratively evaluates the cost and adjusts M and R. It is crucial to integrate this tool seamlessly into the current R&D environment to facilitate its practical application.

5. *Evaluation of the benefits of parameter optimization*: To quantitatively evaluate the effectiveness of our parameter optimization tool, we will focus on measuring the cost savings and performance improvement achieved when applying the tool to common circuit simulation jobs. Specifically, we will assess the reduction in expenses for every 1000 runs of these jobs.

1.2 Contribution

In this thesis, our objective is to develop an HPC job submit-time parameter optimizer based on delay accounting statistics to optimize both cost and performance in the context of HPC. To the best of our knowledge, no previous work has focused on building a parameter optimizer for memory limit in HPC environments. Our contributions can be summarized as follows:

- We design and prototype a delay accounting tool specifically tailored for HPC. This tool accurately measures and accounts for job-wise delay accounting data in various complex scenarios. It is a versatile tool that can be effectively utilized in various HPC clusters and servers.

- We explore the potential of enhancing memory utilization efficiency in HPC environments by intentionally allowing a moderate degree of over-

load and utilizing swap space. Prior research has primarily focused on optimizing resource management by preventing overestimation of resource usage. However, these approaches always ensure that physical memory resources are adequate for all jobs, and as a result, they lack the exploration of intentionally overloading the system and utilizing swap space in the context of HPC. Our work has filled this gap.

- We design and prototype an iterative parameter optimizer that effectively searches for the best parameter setting. Unlike previous researchers who predominantly employ machine learning techniques for predicting HPC cluster parameters, our approach utilizes a simple yet effective search algorithm for optimizing HPC cluster parameter settings.

- We make a contribution to the field by investigating the simultaneous optimization of cost and performance within a specialized chip-design context, thereby creating a mutually beneficial scenario. Prior research has largely overlooked the exploration of these dual objectives.

1.3 Structure

The following parts of this thesis report are organized as follows: we will first briefly introduce the background knowledge and related work in depth (Chapter 2). Chapter 3 will introduce how we implement the delay accounting tool, as well as the experiments and results for the delay accounting measurements of some typical circuit simulation jobs. Then in Chapter 4, we show the experimental setup and results for finding the Pareto front and optimal cost-efficiency solution. how we build a tool to search for the optimal memory limit parameter automatically as well as the evaluation of annual cost savings by applying the parameter optimization tool. Finally, we will give our conclusions and future work (Chapter 5).

Chapter 2

Background and Related Work

This chapter provides background information to help readers understand the basic concepts of the HPC computing environment, memory swap and delay accounting, classification models, etc. It serves as a theoretical basis for the following chapters.
The chapter is constructed as follows. It starts with introducing the basic working principle of job submission and execution on HPC in Section 2.1. In our pursuit of optimizing memory swap space utilization in Linux systems, Section 2.2 delves into the operational mechanisms of memory swapping in Linux and highlights the reasons why a reevaluation of memory swapping is warranted in modern times. To strike the ideal balance between cost and performance, we employ delay accounting as a means to quantify the performance impact of memory swapping. Section 2.3 provides an insightful exploration of how delay accounting is measured in Linux and examines the distinct contributions of different delay fields to the overall performance degradation. Our research and experiments revolve around the typical circuit simulation jobs executed by the NXP design team. Furthermore, the influence of memory swapping is contingent upon the storage type of the cloud provider servers. Section 2.4 conducts a comparative analysis of two different storage types. Section 2.5 delves into existing research endeavors focused on predicting resource usage and optimizing the trade-off between cost and performance. We also highlight the innovative aspects of our work.

2.1 HPC Computing Environment

HPC is a technology that uses clusters of powerful processors, working in parallel, to process massive multi-dimensional datasets (big data) and solve complex problems at extremely high speeds. HPC systems generally operate at significantly higher speeds compared to commodity desktop, laptop, or server systems.

For decades the HPC system paradigm was the supercomputer, a purpose-built computer that embodies millions of processors or processor cores [2].

HPC plays an important role in fields that requires high computing power including financial services, chip designs.

HPC relies on resource management applications to manage available and allocated resources, while batch scheduler systems manage user jobs according to administrator policies and resource availability. This work was conducted on a cluster running job scheduler.

2.1.1 Job Scheduling Overview

The job scheduler for HPC redefines cluster virtualization and workload management by providing a tightly integrated solution that can increase both user productivity and hardware utilization while decreasing system management costs.

The job scheduler is industry-leading enterprise-class software. Job scheduler distributes work across existing heterogeneous IT resources to create a shared, scalable, and fault-tolerant infrastructure, that delivers faster, more reliable workload performance and reduces cost. Job scheduler balances load and allocates resources and provides access to those resources.

Job scheduler provides a resource management framework that takes the job requirements (Figure **??**), finds the best resources to run the job, and monitors its progress. Jobs always run according to host load and site policies.

Some important concepts in the resource management framework are listed below:

- **Hosts** Hosts in the cluster perform different functions.
- **Master host** Job scheduler server host that acts as the overall coordinator for the cluster, doing all job scheduling and dispatch.
- **Server host** A host where jobs are submitted or run on.
- **Client host** A host where jobs are submitted into the job queue.
- **Job** A unit of work that is running in the Job scheduler system. Job scheduler schedules, controls, and tracks the job according to configured policies. In our experiments, an Job scheduler job is commonly known as a circuit simulation job. To distinguish cluster job from the general concept of a "job" in an OS, we will exclusively refer to them as "cluster jobs" in our discussions. Typically, a cluster job is submitted to a job queue using the **bsub** command. Along with the command, we can specify the following submit-time parameters:
  - the Linux command for running the job script
  - the number of job slots N reserved for this job. A job slot is the basic unit of processor allocation in Job scheduler. Normally we assign one job slot to a sequential job and N job slots to a parallel job. Basically, the total number of job slots in a system equals the number of physical cores,i.e. half of the number of coress.
  - the memory limit M_{slot} for each job slot (optional). The total memory limit M for a job with N job slots is $M_{slot} \cdot N$
  - the memory reservation value R (optional). When deciding whether to schedule a job on a host, job scheduler considers the reserved resources of jobs that have previously started on that host. For each load index, the amount reserved by all jobs on that host is summed up

and subtracted (or added if the index is increasing) from the current value of the resources as reported by the Load Information Manager (LIM) to get the amount available for scheduling new jobs. The LIM is capable of automatically collecting resource usage information on the host.

- **Queue** A cluster-wide container for cluster jobs. All cluster jobs wait in queues until they are scheduled and dispatched to hosts. Queues do not correspond to individual hosts; each queue can use all server hosts in the cluster or a configured subset of the server hosts. When you submit a cluster job to a queue, you do not need to specify an execution host. The job scheduler automatically dispatches the cluster job to an appropriate execution host in the cluster based on the cluster job requirements and administrator-defined preferences. Queues implement various cluster job scheduling and control policies to facilitate efficient resource allocation and workload management.
- **Resources** Resources are the objects in the cluster that are available to run work. For example, resources include but are not limited to memory, storage and CPU slots [4].

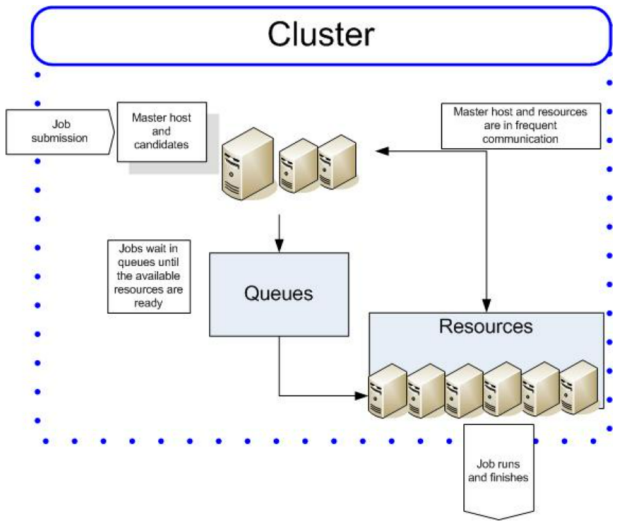


Figure 2.1: **Job scheduler resource management framework. A cluster job is submitted on a server host, with resources and the running queue specified. The request is sent to the master host, where it is assigned to the proper server host with sufficient resources to run the cluster job. The cluster job then starts on that server after being dispatched to it**[4]**.**

2.1.2 Job Starter

A cluster job starter is a specified wrapper script or executable program that typically performs environment setup for the cluster job, then calls and starts

the cluster job itself, which inherits the execution environment created by the cluster job starter. [3]. The cluster job starter is called when a cluster job is dispatched to a certain server and is ready to start execution on it.

2.1.3 Job scheduler Resource Management

All cluster job processes are controlled by the Linux cgroup system so that cgroup memory and swap limits cannot be exceeded. Job scheduler enforces memory and swap limits for cluster jobs by periodically collecting cluster job usage and comparing it with limits set by users. A simple example for setting the physical and virtual memory limit at submission time is as below:

```
$ bsub -M 100  -v 50 ./mem_eater
```

In this example, after the application uses more than 100 MB of memory, the cgroup will start to use swap for the cluster job process. The kernel will page out some pages of the process, satisfying the limits defined in the cgroup (which will be depicted detailedly in Section 2.2. The cluster job is not killed until the application reaches 150 MB memory usage (100 MB memory + 50 MB swap) [5].

2.2 Memory Swapping

Job scheduler runs on Linux servers. This section discusses the functioning of memory swapping in Linux and provides justification for reevaluating its usage.

2.2.1 Linux Paging and Memory Swap

To facilitate efficient management of virtual memory and enhance data access speed, paging is introduced as a logical concept in operating systems. Paging is a storage mechanism that enables the operating system, including earlier versions predating Linux, to retrieve processes from secondary storage (such as a hard disk) into the main memory in the form of pages [34].

In a modern OS, the virtual memory is divided into fixed-length contiguous blocks called a page, memory page, or virtual page, which is the smallest unit of data for memory management in a virtual memory OS [9]. Similarly, in the paging method, the main memory is divided into small fixed-size blocks of physical memory, which are called frames. The size of a frame should be kept the same as that of a page to have maximum utilization of the main memory and to avoid external fragmentation [34].

Swap Space

Swapping is a memory management technique for swapping data between main memory and secondary memory for better memory utilization [23]. As Figure 2.2 shows, When a page fault occurs, swap-in will happen, and a page or a process will be moved from secondary storage to main memory (RAM). When the physical memory is full, swap-out will happen, and a page or process will be taken out from the main memory and placed in the secondary memory [23].

The advantage of memory swapping is remarkable. It is a critical component of memory management, enabling an OS to handle requests that would

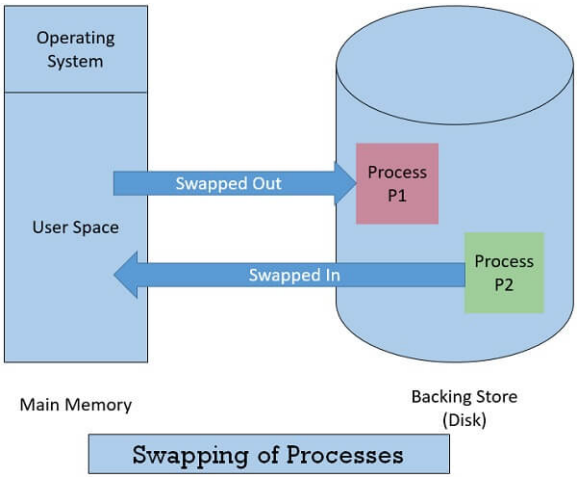


Figure 2.2: **What is memory swap[28]?**

otherwise overwhelm a system; application processes of lesser importance and demand can be relegated to swap space, saving the higher-performance physical memory for higher-value operations [28].

However, it is important to note that swap is not without its drawbacks and the most important and obvious one is that the disk is significantly slower than RAM, and if frequently addressing a large amount of memory, no amount of swap or expensive high-performance disks will make it run within a reasonable time, only RAM will help [1]. That is why we should always use memory swapping in a careful manner. Another limitation is that memory swapping is limited by the available swap space that has been allocated by an OS or hypervisor [28].

2.2.2 Justification of Reconsidering Memory Swap

In the Linux community, "swapping" has traditionally been viewed negatively, with efforts made to minimize its usage. Swapping refers to the process of reclaiming memory by moving less frequently used data from RAM to the swap device, usually a slower rotating storage medium. This practice has been avoided due to the performance impact caused by scattered I/O operations and seek times on rotating storage devices.

However, Linux developers suggest that recent advancements in hardware, such as fast random I/O devices like SSDs and persistent memory, warrant a reevaluation of swapping. Swapping can now be seen as more than just a last-resort overflow mechanism, but rather as an extension of memory that optimizes the balance between the page cache (file-backed pages) and the anonymous working set (anonymous pages) even under moderate load.

For example, developer Johannes Weiner's patch set proposes changes to optimize swapping and balance file-backed pages with anonymous working sets. By tracking the cost of page reclamation and considering rotations and refaults, the kernel can determine the best list to reclaim from. Initial benchmarks show improved performance. Such practices suggest that memory swapping can now

be seen as an extension of memory, optimizing in-memory balance and potentially enhancing system performance [10].

2.3 Delay Accounting

Delay accounting is a crucial tool for assessing the performance implications of memory swapping. This section elucidates the measurement process employed by the Linux kernel for delay accounting and provides an in-depth analysis of the various fields involved in this accounting mechanism. Note that this is just a theoretical example and in the real world normally swapping delay may only take milliseconds.

2.3.1 What is Delay Accounting?

Delay accounting can be used as a metric for measuring the impact of memory swap as well as finding the optimal Resident Set Size (RSS), i.e., the maximum physical memory usage.

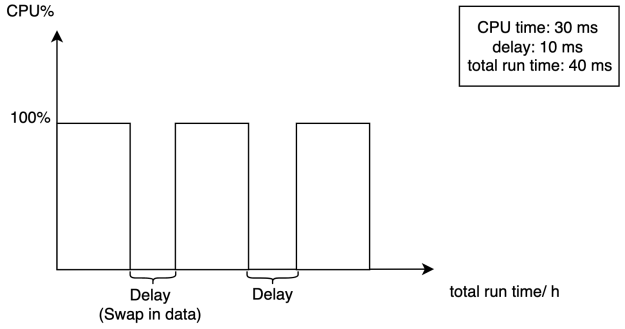


Figure 2.3: **Delay accounting. The task first runs for 10 ms, then stops for 5 ms to wait for the memory to swap in, and so and forth. In total, the real execution time is 30 ms but the total elapsed time reaches 40 ms, which means 10 ms are spent in memory swapping.**

Delay is the time period when a task temporarily stops its work and waits for some kernel resource to become available during execution, e.g., wait for memory to swap in (Figure 2.3). Delay accounting is pretty useful to identify the resource bottlenecks in the system and optimize the resource allocation. One of the most significant usage of delay accounting is that it can help us to define the optimal RSS.

The Linux kernel provides delay measurement aggregated for all threads belonging to a thread group (process) through continuous monitoring.

The per-process delay accounting functionality measures the delays experienced by a process while it is waiting for the completion of block I/O, the memory swap-in, and the memory claim (the memory swap-out delay) respectively. These statistics are aggregated in a per-process manner and made available to userspace through the **Taskstats** interface after the completion of all

the threads of a thread group(process) [16]. **Taskstats** also involves other process running-time information like process id, process elapsed time, and accumulated core memory usage. Table 2.1 lists the main delay accounting and running-time measurement fields in **Taskstats**. **Blkio_delay_total** represents the time the task waits for the completion of synchronous block I/O. It shows the impact of I/O operations initiated by the task on overall performance. **Swapin_delay_total** indicates the time spent waiting for swapping in pages synchronously. It demonstrates the impact of synchronous memory swapping on system performance. **Freepages_delay_total** captures the time spent waiting for memory reclaim, both synchronous and asynchronous. It shows the impact of memory reclamation operations on system performance. These accounting fields could all be affected by memory swapping to some extent and thus help evaluate the impact of swapping delay accurately and divisibly. The complete **Taskstats** fields list can be found in Linux delay accounting document [19].

Table 2.1: **Delay accounting and running-time measurement fields in struct *Taskstats* [16].**

Description	Time Unit	Field Name
waiting for completion of synchronous block I/O initiated by the task	ns	blkio_delay_total
waiting for swapping in pages (synchronous)	ns	swapin_delay_total
waiting for memory reclaim (both synchronous & asynchronous)	ns	freepages_delay_total
elapsed time	us	ac_etime
user CPU time	us	ac_utime
system CPU time	us	ac_stime

Delay accounting can be obtained by creating a socket connection between user space and kernel space using Netlink, which is a communication mechanism in the Linux kernel.

2.3.2 Break-Down of Delay Accounting

Figure 2.4 illustrates the measurement of various delay accounting fields in the Linux kernel. When the kernel encounters blocking I/O, the corresponding time interval is recorded as either blkio delay or swap-in delay, depending on the status of the page fault flag. On the other hand, the duration between the occurrence of insufficient main memory and the subsequent cleanup of pages from the main memory is categorized as freepage delay. Notably, freepage delay involves non-blocking I/O operations and is significantly faster than blocking I/O.

It is essential to note that the measurement of blocking and non-blocking I/O delays is independent, although some overlap between them may occur. As a result, we can establish that:

$$blkio\ delay + swapin\ delay + freepage\ delay > Total\ delay \tag{2.1}$$

In terms of program execution, the user CPU time is expected to remain constant and unaffected by environmental factors like system workloads. Ideally, the elapsed time would be the sum of the total delay, user CPU time (ac_utime), and system CPU time (ac_stime):

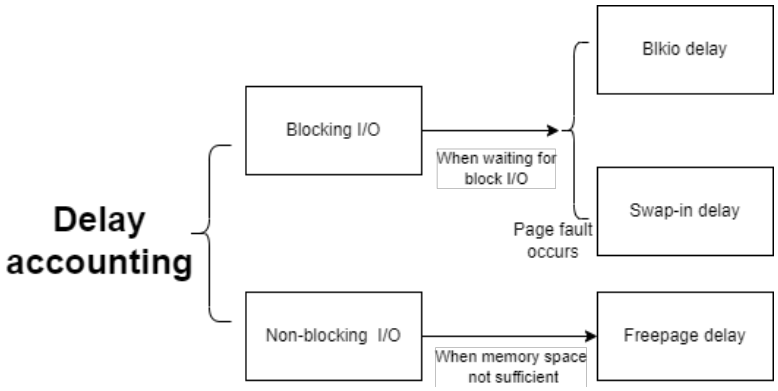


Figure 2.4: **Measurement of delay accounting in Linux kernel [25].**

$$Total\ delay + ac\_utime + ac\_stime = ac\_etime \tag{2.2}$$

However, due to potential overlap between system CPU time and delay, the actual relationship is:

$$Total\ delay + ac\_utime + ac\_stime > ac\_etime \tag{2.3}$$

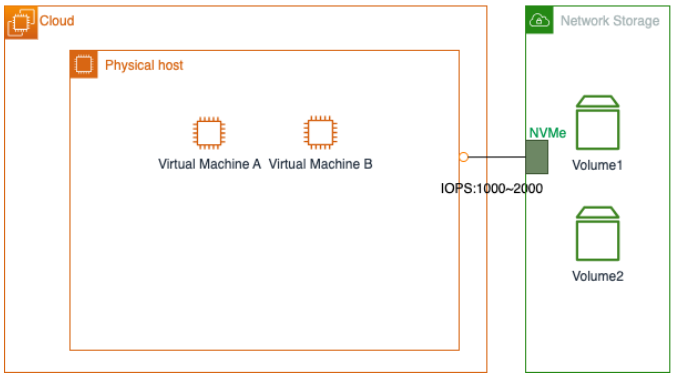
2.4 Cloud Servers' Computer Architecture

NXP's HPC facilities are currently undergoing a transition as they move to the cloud. cloud provider offers a wide range of server types, allowing users to select the most appropriate configuration based on their specific application requirements, performance needs, and budget constraints. cloud provider server types refer to the different virtual machine configurations available for deployment on cloud. A server type defines the hardware of the host computer used for the virtual machine and determines the resources available to the server, including the CPU, memory, storage, and networking capabilities.

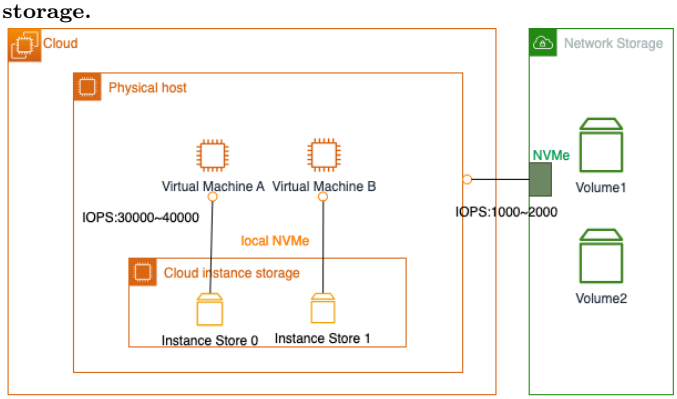
2.4.1 Cloud Server Storage

The cloud provider offers two prominent storage services: network storage and server storage. cloud provider provides these two options in different server types. An example is that the server without local NVMe uses network storage (Figure 2.5), while the server with local NVMe is similar to the former one but includes additional local NVMe-based SSD block-level storage that is physically connected to the host server.

The network storage is a block-level storage service provided by cloud provider. It allows users to create persistent block storage volumes and attach them to the cloud servers. The network storage volumes are network-attached storage devices that provide durable block-level storage, similar to traditional hard drives. The network storage volumes are highly durable and maintain data integrity. They can persist independently of the lifecycle of a cloud server, allowing for data retention even after a server termination.



(a) **Computer Architecture of server without local NVMe storage.**



(b) **Computer Architecture of server with local NVMe storage.**

Figure 2.5: **Server without and with local NVMe storage.**

On the other hand, server storage volumes, also known as ephemeral storage, are temporary block-level storage volumes that are physically attached to the host server of a cloud server. These volumes provide temporary storage that is ideal for applications and workloads that require high I/O performance or temporary data storage. server Storage volumes offer high-performance storage with low latency and high IOPS (Input/Output Operations Per Second). They provide fast and direct access to the underlying hardware, making them suitable for applications that require high-speed data access and processing. server Storage volumes are physically attached to the host server, offering low-latency and high-bandwidth storage access. This proximity to the server can result in better performance compared to network-attached storage solutions like network storage. However, server storage is not persistent, meaning that the data stored in it is lost if the server stops or terminates [7].

Regarding the impact on memory swap delay, it is important to note that server Storage volumes, being physically attached to the host server, provide faster access to data compared to network-attached storage. This can lead to

reduced latency in memory-swapping operations, as the data can be retrieved and written at higher speeds. On the other hand, the network storage volumes, being network-attached, may have slightly higher latency in memory swap operations due to the additional network overhead involved.

2.5 Related Works and Research Gaps

This section presents the related literature works in four significant sub-topics of the project, analyzing and comparing the goal and the scientific approaches of the existing works with our project.

2.5.1 Resource Prediction in HPC

Predicting cluster job resource usage has been a significant area of research for resource optimization of HPC in recent years and a lot of research work has been done on not only memory usage prediction, but also CPU usage, job wall time predictions, etc.

Almost all current works use Machine Learning to directly predict the value of job resource usage. The predicted value then can be integrated by resource managers to help users or administrators to reduce the over-reservation of resources. Taghavi et al. [30] from Qualcomm introduces their work on predicting job memory usage using various machine learning algorithms and tools, which aims to ensure jobs do not under-request memory more than 5% of the time. Their work shows the memory usage of prior historical job statistics can be a good guess for the new jobs, and by applying a simple linear regression model, the prediction accuracy has reached 90%, while the average pending time has dropped by 27%˜34%. Tanash et al. [31] have performed similar work with Decision Tree Regressor (DTR), and proposed a detailed performance evaluation test that assesses the performance improvement benefiting from the predictor by different metrics. Their results also indicate that for larger jobs, the improvement of turnaround time and resource utilization is more substantial. Li et al. [18] from the IBM research team target only predict job memory usage for large memory jobs using a two-stage prediction method, which is more practical for production usage and promotes the prediction accuracy for those jobs really cared by resource manager users. Rodrigues et al. [27] from another IBM research team proposes an innovative tool that leverages the predictions of all methods and selects the most promising ones in a given situation, which significantly over-performs the single machine learning model method. They also introduce the details of the implementation of the prediction tool and how it was incorporated into the Job scheduler batch scheduler. Matsunaga et al. [21] compares the advantages and disadvantages of several machine learning methods in resource prediction. They also propose a novel method called PQR2, which is an extension of an existing classification tree algorithm, and exhibits better accuracy when compared to other algorithms, due to its ability to better adapt to scenarios with different characteristics (linear and non-linear relationships, high and low density of training data points) by choosing different models for its nodes and leaves. Besides, some other researchers [26] [6] [32] [11] also use similar machine learning approaches to predict the maximum memory usage.

Most of the previous work is aiming at finding the read-line of memory usage

saving. Their goal is mainly to ensure that the memory reservation is just beyond the actual peak memory usage; so no considerable resource waste will be caused, and no job crash will be caused by memory underestimation as well. They regard the physical and virtual memory as a whole and do not consider their usage separately. Our goal, however, is to further set an extra limit on physical memory usage and make use of the swap space in a proper way.

2.5.2 Optimize Cost and Performance

A Myriad of studies to find the trade-offs between cost and performance in HPC and cloud computing environments can be found in the literature. Inggs et al. [15] treats the problem of finding the optimal cost-performance trade-offs problem as a multi-objective optimization problem, in which they try to find the Pareto optimal trade-offs. They also proposed a formal Mixed Integer Linear Programming (MILP) technique, which produces a trade-off that is up to 110% faster and over 50% cheaper than a simple heuristic approach. Nunez et al. [24] focus on the impact of a high number of parameters and present a model that computes a cost-per-performance metric using different combinations of hardware configurations based on the simulation in a complex simulation platform. They also employ a weighted-sum objective function to evaluate the trade-off between different objectives. Singh et al. [29] present a multi-objective genetic algorithm formulation for selecting the set of resources to be provisioned that optimize the application performance while minimizing resource costs. They use trace-based simulations to compare the application performance and cost using the provisioned and the best-effort approach with a number of artificially generated workflow-structured applications. However, we can still draw valuable insights from their approach. For server, we can adopt a weighted objective function to effectively evaluate the trade-off between different objectives, namely cost and performance. Additionally, the utilization of an iterative evaluation-adjustment method can also be beneficial for our project.

2.5.3 Memory Swap and Performance Optimization

There is a very limited amount of related work in the area of using memory swapping and cgroup configurations to improve the system performance. Though there are various posts and blogs elaborating on how to use memory swapping, we have not seen much similar work in literature that identify or address the relevant problems. Here we list some works we have found so far. Zhuang et al. [35] have published works that present the findings about memory-related performance issues of cgroups during certain scenarios. Specifically, (1) memory is not reserved for cgroups (as with virtual machines); (2) both anonymous memory and page cache are part of memory limit and the former can evict the latter; (3) OS can steal page cache from any cgroups; (4) OS can swap any groups. These issues can significantly affect the performance of the applications running in cgroups.

2.5.4 Combined Results with another Team

Currently, the Data Analysis team at NXP is conducting work that aligns well with our research. The team's focus lies in addressing the issue of HPC

users over-requesting memory for certain jobs, resulting in significant resource wastage, while underestimating memory requirements for other jobs, leading to a higher risk of job crashes. Their work involves classifying jobs into two distinct types: high-memory-wasting jobs and high-risk jobs. Based on these classifications, they further predict the maximum memory usage ($R_{predict}$) of a job, which serves as a reference for user input settings. The memory predictor's output range is expressed in Inequality 2.4.

$$R_{real} <= R_{predict} < max(R_{real} + 2 + 0.5 * \frac{R_{real}}{ln(max(R_{real}, 3))}, 4) \tag{2.4}$$

In our research, we extend their approach by introducing a constraint on memory usage (R_{opt}) that effectively utilizes swap space on the disk. As seen in Inequality 2.5, our memory limit R_{opt} is typically set to be no greater than the actual memory usage of a job, allowing for enhanced cost savings and achieving an optimal weighted tradeoff between cost and performance. This improvement builds upon their work by considering additional factors such as disk and swap space utilization, enhancing the overall efficiency and effectiveness of memory management in the HPC environment.

$$R_{opt} <= R_{real} <= R_{predict} \tag{2.5}$$

Figure 2.6 illustrates the comprehensive workflow that combines the two pieces of work. Initially, users submit their job sets to Job scheduler. These jobs are then classified as either high-risk or high-memory-wasting jobs using a machine learning model. For jobs classified as high-memory-wasting, the machine learning model predicts their memory usage ($R_{predict}$). Subsequently, the parameter optimizer utilizes $R_{predict}$ as the initial upper bound of the search space and iteratively determines an optimal memory usage (R_{opt}) that achieves an optimal weighted sum trade-off between cost and performance. Upon obtaining R_{opt}, the jobs are updated accordingly and subsequently submitted to the job queue, from where they are dispatched to different servers within the Job scheduler cluster.

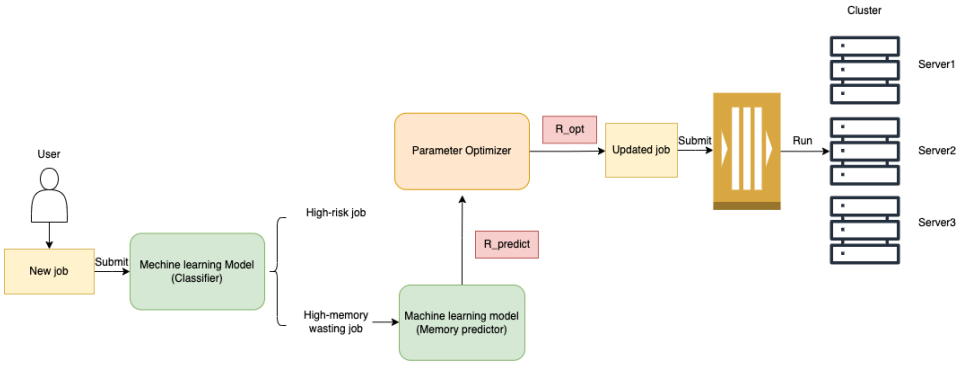


Figure 2.6: **Workflow diagram of the combined model.**

Chapter 3

Delay Accounting Measurements

This chapter presents the development of the delay accounting tool, the methods employed for conducting delay accounting measurements, along with an analysis of the obtained results.

3.1 Development of the Delay Accounting Tool

This section presents the development process of the delay accounting tool, the verification of its results, and its subsequent deployment on the cluster.

3.1.1 Build the Delay Accounting Tool

a cluster job can comprise multiple processes, while we are particularly interested in obtaining cluster job-level delay accounting results; however, the Linux kernel only provides per-process delay accounting statistics through its **Taskstat** interface. Therefore, we need to develop a delay accounting tool to aggregate these statistics for each cluster job and obtain the per-cluster job delay accounting results. In order to establish the relationship between processes and build a process tree, we can utilize the process event connector utility provided by Linux, which reports fork, exec, and exit events to user space. To collect per-process measurement results and accumulate the results for each cluster job, we create two sockets: one for receiving **taskstat** messages and another for listening to process events. Upon receiving the event message "process (id=x) fork process (id=y)", we record the PID of the new process in a process tree. When receiving **taskstat** data for process y, we accumulate the results in the total result and remove PID y from the process tree. When all processes are removed from the process tree, it signifies the completion of cluster job accounting, and we can output the result.

However, managing processes is not always as straightforward as summing up subresults; it can be highly complex. We may encounter various scenarios, such as very short-lived processes that pose challenges in capturing accounting data, orphan processes where a child process exits later than its parent, and multi-threading. The most challenging problem arises from the asynchrony between

the two sockets, leading to potential race conditions, where a process's taskstat data may be received before the fork event x. In this case, the **taskstat** might just be discarded since the PID is not recorded in the process tree yet, resulting in missed delays in the final results.

To address the race condition, we propose a three-event algorithm. We introduce a structure to store the three events that indicate the complete lifecycle of a process: fork, exec, and exit, along with the corresponding taskstat data. Only when a cluster job receives all three events can we conclude that it has truly finished, enabling us to remove it from the process tree. This approach resolves the race condition issue.

Additionally, it is important to note that due to the resource constraints in HPC systems, the tool should be lightweight, consuming minimal memory and CPU resources.

3.1.2 Verification of the Results

To validate the accuracy and robustness of the delay accounting tool in complex scenarios involving multiple processes and threads, we have devised a comprehensive test case. Within the **taskstat** structure, there exists a field called "readchar", which corresponds to the cumulative number of bytes passed to the read() and pread() system calls. This value is expected to remain constant across different runs of the same cluster job. To verify the accuracy of the delay accounting tool in collecting data from all processes, we execute the same command in all the threads of a cluster job. By comparing the total "readchar" measurement of the entire cluster job with the product of the number of threads and the per-thread "readchar", we can ascertain whether the measurement results of all subprocesses are accurately collected.

For the following dd command, the "readchar" measurement value is 1,048,576.

```
dd if=/dev/zero bs=1k count=1k
```

We devised a cluster job script that employs a recursive process forking approach, generating a total of $2^n - 1$ processes (n=6). Within each process, five threads are created. By measuring the "read char" metric of this cluster job, we obtained a value of 398,131,968. This quantity corresponds precisely to the product of 1048576, 63, and 6. Thus, our findings serve as a validation demonstrating the accuracy of the delay accounting tool in scenarios involving multiple processes and threads.

3.1.3 Deployment of the Tool

Figure 3.1 illustrates the process of collecting and aggregating delay accounting data within the HPC cluster. The deployed delay accounting tools are distributed across all worker servers, generating accounting results. These results are subsequently transmitted to a central log collector for consolidation.

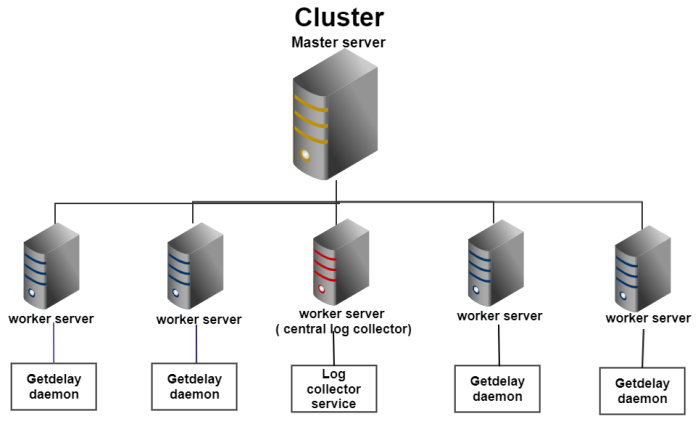


Figure 3.1: **Deployment of delay accounting tool on a cluster.**

3.2 Evaluate the Performance Influence of Memory Swapping

3.2.1 Idea

Measuring delay accounting is of utmost importance in gaining a comprehensive understanding of the impact of memory swapping on cluster job performance. This methodology involves adjusting the memory limit (M) during cluster job submission and meticulously documenting the delay accounting data. By systematically varying the memory limit, we can analyze the impact on cluster job performance metrics and identify the specific delay accounting fields responsible for these changes.

We measure the cluster job performance using elapsed time, which represents its execution duration. To understand the impact of memory limits, we analyze various delay accounting fields, including blkio, swap-in delay, and freepage delay. By examining these fields individually, we can identify the specific factors contributing to the increase in elapsed time.

Memory limits can affect different delay accounting fields differently. Memory swapping, for example, results in elevated swap-in delay and swap-out delay (freepage delay). Notably, swap-in delay, characterized by blocking I/O operations, is more significant as it suspends process execution until the I/O operation completes.

To quantify the influence of each delay accounting field, we use the Linux kernel's measurement capabilities to obtain absolute swap-in, blkio, and freepage delay data. Additionally, we calculate the relative delay for each field with the

following equation,

$$Relative\ delay_{delay\ field} = \frac{Absolute\ delay_{delay\ field}}{Elapsed\ time\ of\ the\ job\ without\ memory\ limit\ set} \cdot 100\%$$

which provides insights into their respective contributions to the overall performance change.

3.2.2 Test Set-ups

Table 3.1 presents the specifications of the cloud provider the cloud servers used in our experiments, namely serverA. Such large servers are commonly used in R&D processes. To investigate the influence of storage type on performance, we conducted a controlled experiment using serverA_disk as well.

Table 3.1: **The cloud server for delay accounting measurement. "4x1900 NVMe SSD" means 4 NVMe SSDs, each with a capacity of 1900 GB. Generally, cloud provider provided a pool of NVMe SSDs that the coress can access. Thus the 4 NVMe SSDs are shared between all the cores.**

server Type	cores	Memory (GB)	server Storage (GB)	Network Bandwidth (Gbps)	network storage Bandwidth (Gbps)
serverA	128	256	network storage-Only	50	40
serverA_disk	128	256	4x1900 NVMe SSD	50	40

The measurements were conducted for three distinct simulation tests, each representing a different stage in the analog design process. And they are all characterized by a substantial memory footprint. These tests were executed using the SimulatorA simulator, which accurately emulates the memory-intensive workloads typically conducted by the NXP design team.

Our original intention was to primarily investigate relatively large cluster jobs, specifically those with memory usage larger than 50GB, as they have more potential for cost-saving optimizations. However, we have faced the challenge of limited availability from some colleagues to provide help, currently, we were unable to obtain these large cluster job test cases. Nonetheless, the available SimulatorA tests still provide valuable insights and understanding of the system's behavior.

Table 3.2 provides detailed descriptions of the simulation jobs along with their corresponding memory usage and the number of cluster job slots they use. Typically, when engineers submit a simulation job, they have the option to specify two parameters: the number of threads the job will utilize and the number of job slots the cluster job will occupy. The number of threads and job slots does impact job performance by influencing the parallelism within the

Table 3.2: **Representative circuit simulation jobs. The memory usage statistics are provided by NXP analog design team. Normally, we have the flexibility to change the number of job slots when submitting the simulation job. While the number of threads that we can spawn is also adjustable, for most of the simulation jobs, we typically use 8 threads. The parameters listed in this table are the ones we utilized in the delay accounting measurement in Section 3.2.3.**

Simulation Job Name	Description	Memory Usage (GB)	Number of Job Slots	Number of Threads
TC4-hb_SimulatorA	Phase Locked Loop (PLL) using CLN28HPC technology. Uses Cyclostationary noise analysis to account for frequency folding effects due to the harmonic content of the periodic steady state.	2.2	2	8
TC4-ChirpPLLpnoisejitter	Phase Locked Loop (PLL) using CLN28HPC technology. Measuring circuit jitter using a combination of STEADY and PNOISE analysis.	8.8	8	8
SimulatorA–test3	Analog Digital Converter developed using SMOS10 technology. Measures calibration using transient analysis.	26	8	8

cluster job. However, this impact has a relatively minor effect on overall memory usage.

Additionally, measurements were carried out on seven benchmark tests using the SimulatorB simulator, with relatively smaller memory usage.

For each job, we selected a range of memory limits, M, based on our prior knowledge of the estimated memory usage range. A suitable increment step, denoted as M_{step}, was selected to ensure clear observation of the curve's change trend. Subsequently, for each memory limit M, the measurement process was repeated N times to account for potential experimental errors in individual runs.

In each iteration, a cluster job with the memory limit set to M was submitted, and upon its completion, the delay accounting results were recorded. Relative delays were computed for each delay accounting field by dividing the job's absolute delay by its elapsed time. The average value and standard error were calculated to provide a comprehensive evaluation.

3.2.3 Measurement Results

Figure 3.2 presents the results of the delay accounting measurements conducted on three circuit simulation test cases with substantial memory requirements. The experimental results reveal a consistent reduction in both the elapsed time and swap-in delay as the memory limit is increased. Additionally, it is noteworthy that the relationship between elapsed time, swap-in delay, and memory limit is non-linear. Particularly, as the memory limit approaches smaller values, both the elapsed time and swap-in delay curves become steeper, indicating more rapid changes in performance.

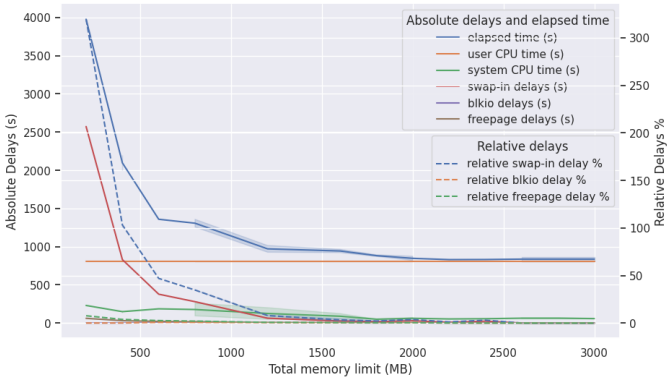
Meanwhile, the graph clearly demonstrates that the swap-in delay is significantly higher than the blkio delay and freepage delay, indicating that the performance impact is primarily governed by the swap-in delay, in line with our expectations specified in Section 3.2. Additionally, the user CPU time remains constant regardless of changes in the memory limit, which aligns with our expectations as the user CPU time for a program should remain unchanged. Furthermore, we also notice that the line representing the elapsed time remains parallel to the line representing the absolute swap-in delay, suggesting that the difference between them is constant and attributable to the user CPU time. Moreover, it is observed that the memory limit at which the absolute swap-in delay reaches zero corresponds to the maximum memory usage of the cluster job provided by the NXP analog design team.

The measurement results for the seven small-memory usage test cases run by SimulatorB can be found in Appendix A.

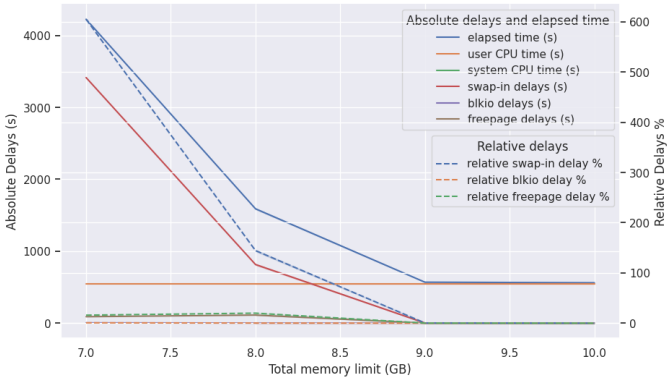
3.2.4 Verification

In Section 3.2.3, we discussed our observation that the memory limit where the absolute swap-in delay reaches zero corresponds to the maximum memory usage of the job, as provided by the analog design team. To further support this finding, we conducted a comparison between the maximum memory usage measured by the delay accounting method and the memory usage measured by The job scheduler.

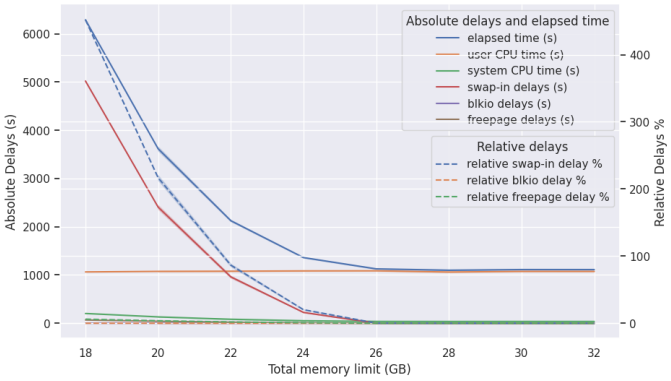
The job scheduler obtains memory usage information for a cluster job by leveraging the Linux cgroup system when cgroup is enabled. This system enables



(a) **SimulatorA-test1 test**



(b) **SimulatorA-test2 test**



(c) **SimulatorA -test3 test**

Figure 3.2: **Delay accounting measurement results by SimulatorA. The area around the line represents the standard error bar.**

Table 3.3: **Memory usage measured by delay accounting vs. by The job scheduler.**

Benchmark test	Memory usage per cluster job when swapping delay begins to happen	Maximum memory usage per cluster job measured by The job scheduler (with no limit)
SimulatorB-test1	65MB	65MB
SimulatorB-test2	840MB	843MB
SimulatorB-test2_4thread	840MB	843MB
SimulatorB-test3	90MB	88MB
SimulatorB-test4	70MB	66MB
SimulatorB-test5	160MB	154MB
SimulatorD	70MB	73MB
SimulatorA-test2	9.7GB	9.7GB
SimulatorA-test1	2.2GB	2.2GB
SimulatorA–test3	26GB	26GB

accurate control and monitoring of processes within cluster job environments. Through the utilization of cgroups, The job scheduler effectively captures all cluster job processes, ensuring precise and reliable memory usage data [20]. This data plays a critical role in NXP's analysis and decision-making processes regarding cluster job memory management.

Table 3.3 presents the comparison results for all ten test cases. Evidently, the maximum memory usage measured by the delay accounting test closely aligns with the measurements obtained by The job scheduler. This observation provides strong evidence supporting our findings.

3.2.5 Job Sensitivity to Memory Swapping

The experiment revealed that the impact of decreasing physical memory usage varied among different simulation jobs. Table 3.4 summarizes the relative swap-in delay, which serves as a rough indicator of performance penalty, observed when applying different memory limits to the three simulation jobs.

It was observed that the "SimulatorA-test1_SimulatorA" test exhibited the least sensitivity to memory limit, with a 40% memory saving resulting in only a 13% performance loss. On the other hand, the "SimulatorA-test2" test showed the highest sensitivity, where a mere 10% memory saving caused a 378% performance loss. The "SimulatorA -test3" test demonstrated sensitivity levels between the two aforementioned cases.

Based on these preliminary findings, it appears more worthwhile to impose memory limits on insensitive jobs such as the "SimulatorA-test1" test.

Given the diverse range of job types and our current limitation in performing delay-accounting measurements on a substantial number of jobs, accurately estimating the percentage of insensitive jobs poses a challenge. However, based on our analysis, it is highly likely that a significant number of such jobs exist.

Table 3.4: **Relative memory saving vs. performance penalty.**

Simulation Job Name	Memory limit M (GB)	Relative Memory saving	Absolute Memory Saving (GB)	Relative swap delay
SimulatorA-test1 test	2.22 (no limit)	0%	0	0%
	2.00	10%	0.22	2%
	1.75	20%	0.44	5%
	1.50	30%	0.66	7%
	1.25	40%	0.88	13%
	1.00	50%	1.10	41%
	0.75	60%	1.32	70%
	0.50	70%	1.54	78%
SimulatorA-test2 test	9 (no limit)	0%	0	0%
	8.5	5%	0.5	100%
	8	10%	1	378%
	7.5	16%	1.5	450%
	7	22%	2.0	651%
SimulatorA -test3 test	26 (no limit)	0%	0	0%
	24	7%	2	20%
	22	14%	4	86%
	20	21%	6	216%
	18	28%	8	457%

3.2.6 SimulatorC Job

Given the complexity of SimulatorA tests and our limited access to the source code, we pursued a supplementary analysis on a specific job performed with the SimulatorC tool, for which we do have access to the source code. This experiment aims to pinpoint scenarios in which memory swapping has a lesser impact.

The results of delay accounting measurements for this job are depicted in Figure 3.3. We observe that the job is relatively insensitive to memory swapping, indicating a potential high benefit from leveraging memory swapping strategies. This behavior can be attributed to the specific data manipulations performed by the tool. Initially, the tool loads file(s) into dataset 1, followed by a transformation into dataset 2, and finally, the simulation of dataset 2 is conducted independently of dataset 1. Consequently, dataset 1 can be safely swapped out with minimal impact on performance.

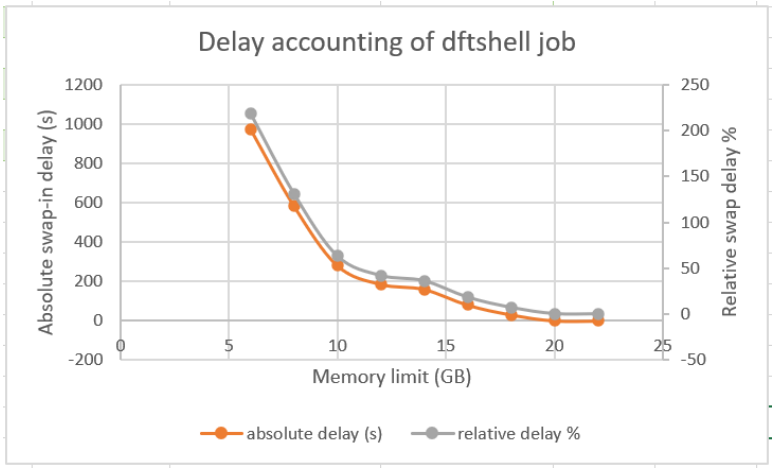


Figure 3.3: **Deployment of delay accounting tool on a cluster.**

3.2.7 Compare Two Storage Types

As discussed in section 2.4.1, local NVMe storage typically exhibits lower swap delay compared to network storage storage. However, servers equipped with local NVMe may come at a slightly higher cost than network storage-only servers. For server, the serverA server costs 5.44 $/h, while the serverA_disk server, which includes additional local NVMe storage, costs 6.45 $/h. To aid in decision-making between these storage types, we aim to quantify the difference in swap delay between them.

Table 3.5, Table 3.6, and Table 3.7 compare the swap delay and elapsed time for three SimulatorA jobs using both storage types. The results demonstrate a significant 4.5 times difference in absolute swap delay between the two storages.

Table 3.5: **SimulatorA-TC4-hb: Compare delay on NVMe and network storage**

Total memory limit (MB)	network storage-only (serverA)			Local NVMe (serverA_disk)		
	Absolute swap-in delay(s)	Relative swap-in delay	Elapsed time(s)	Absolute swap-in delay(s)	Relative swap-in delay	Elapsed time(s)
2200(no limit)	0	0.0%	871	0	0%	865
1000	173	19.8%	1189	24	2.7%	1153
800	280	32.1%	1255	54	6.2%	1172
600	377	43.3%	1325	104	12.2%	1285

Table 3.6: **SimulatorA–test3: Compare delay on NVMe and network storage**

Total memory limit (GB)	network storage-only (serverA)			Local NVMe (serverA_disk)		
	Absolute swap-in delay(s)	Relative swap-in delay	Elapsed time(s)	Absolute swap-in delay(s)	Relative swap-in delay	Elapsed time(s)
28 (no limit)	0	0%	1098	0	0%	1072
26	1.5	0.2%	1135	2	0.2%	1081
24	223	20.3%	1358	62	5.8%	1227
22	940	85.6%	2154	213	19.9%	1512
20	2378	216.8%	3576	473	44.1%	2019
18	5020	457.3%	6289	871	81.3%	2643

Table 3.7: **SimulatorA TC4- ChirpPLLpnoisejitter: Compare delay on NVMe and network storage**

Total memory limit (GB)	network storage-only (serverA)			Local NVMe (serverA_disk)		
	Absolute swap-in delay(s)	Relative swap-in delay	Elapsed time(s)	Absolute swap-in delay(s)	Relative swap-in delay	Elapsed time(s)
9 (no limit)	0	0%	538	0	0%	529
8	2036	378%	2781	69	13%	632
7	3503	651%	4592	718	135.7%	1763
6	10220	1899%	11810	2205	416.8%	3759

3.3 Compare the Performance with and without Explicit Memory Limit

(NOTE: In this section, the term "parallelism" refers specifically to the number of cluster jobs concurrently running in the system, as opposed to the number of threads operating in parallel within an individual cluster job.)

As mentioned in Chapter 1, both R and M serve as quantifiable parameters for managing memory swapping in the system. Since we must enforce constraints on job parallelism to prevent over-commitment in the system, utilizing R is essential. We have two fundamental options for their application. The first approach involves using R exclusively to regulate system parallelism, allowing the OS to determine which jobs require more memory swapping. The second approach employs identical values for both R and M, enabling us to explicitly set memory limits, thereby defining parallelism and imposing restrictions on each job's maximum memory usage.

To determine which scheme offers superior performance, we designed an experiment to compare their respective swapping delays.

Table 3.8 showcases the outcomes of cluster job parallelism experiments conducted with and without explicitly set memory limits (M) on two distinct storage types: network storage (performed on serverA server) and local NVMe (conducted on serverA_disk server).

An explicit memory limit indicates that we set both M and R to the average of the total available system memory, based on the desired parallelism. For example, if the system has 160GB memory and we aim for parallelism=4, then we explicitly assign a memory limit of 40GB to all 4 jobs. Conversely, the absence of an explicit limit implies that we solely set R to that average memory value.

Both serverA and serverA_disk servers offer a total memory of 256 GB, with 22 GB reserved for the operating system. This allocation results in 234 GB of available memory for running cluster jobs. The experiments were conducted with varying parallel cluster job counts (9, 10, and 11). We selected a parallelism level of 9 as it marks the threshold where jobs begin utilizing swap space. The results reveal that the presence of explicit memory limits notably increases memory swap delay (both time and count) and elapsed time, especially evident in the cases with network storage storage.

To elucidate why better performance is achieved without explicit memory limits, we conducted experiments measuring memory usage over time by recording per-job memory usage collected by cgroup. We compared two setups showing the most significant gap in swap delay: parallelism=11 on network storage, with and without explicit memory limits. Figure 3.4a illustrates that without explicit limits, maximum memory usage can reach 26GB, while Figure 3.4b shows explicit limits restrict it to 21.27GB. Figure 3.4c. 3.4d shows the contribution of the 11 jobs to the total memory usage in the system. And we can observe that in both cases the peak system memory usage reaches 234GB.

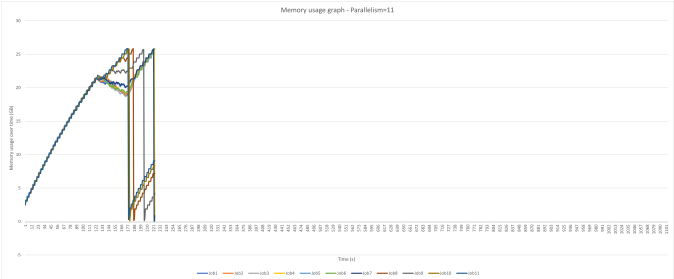
We also observed a delay in memory usage phases among different jobs, stemming from slight variations in start times. Jobs that start earlier obtain priority for physical memory, while those lagging behind experience more swapping delays. These observations collectively suggest that jobs without explicit memory limits can use available memory from lower usage jobs, reducing the

Table 3.8: **cluster job parallelism experiments with or without explicitly set M. All measurements in the table are average statistics for each cluster job in the parallel cluster job set. Swap count in this table means the number of swap-in operations. On a given type of storage device, each swap operation takes the same amount of time.**

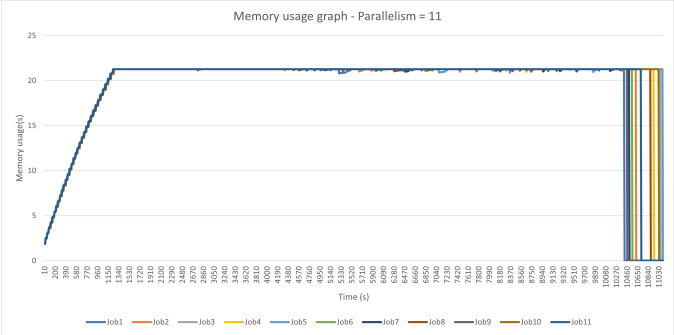
Memory limit	Storage type	Parallelism	Swap count	Swap delay(s)	Elapsed time (s)
explicit M	network storage	9	4,929	2	1,651
		10	1,593,705	2,395	4,100
		11	6,122,192	9,827	11,711
explicit M	local NVMe	9	5,451	1	1,647
		10	2,351,574	284	2,013
		11	5,819,554	676	2,505
non-explicit M	network storage	9	0	0	1,640
		10	143,661	242	1,917
		11	223,679	366	2,078
non-explicit M	local NVMe	9	0	0	1,651
		10	210,524	25	1,705
		11	518,287	59	1,740

need for memory swapping. In contrast, explicit memory limits prompt immediate data swapping to disk upon reaching the limit, leading to performance slowdowns. Frequent memory swapping can contend for disk I/O bandwidth, diminishing effective bandwidth for all system jobs. Consequently, explicitly set memory limits may cause larger memory swap delays and potentially decrease overall system performance.

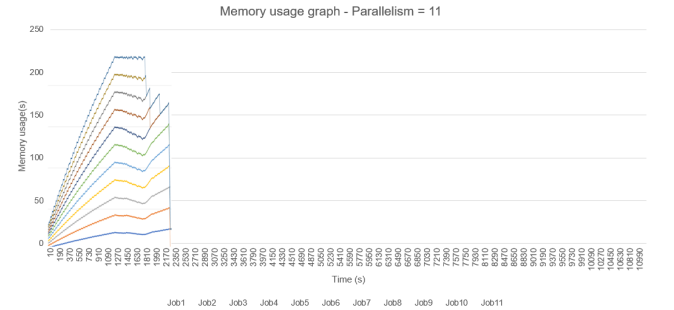
Figure 3.5 illustrates the impact of increasing parallelism beyond 11 on cluster job performance. We observe a sharp decline in performance when parallelism reaches 12; however, this decline is still significantly more favorable compared to the scenario of applying explicit memory limits.



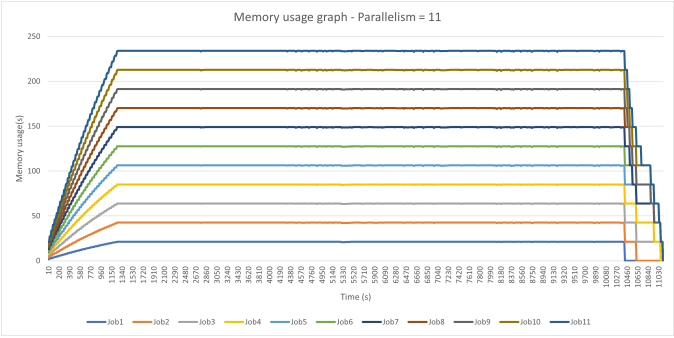
(a) **Without explicit memory limit - line plot**



(b) **With explicit memory limit - line plot**



(c) **Without explicit memory limit - stacked-line plot**



(d) **With explicit memory limit - stacked-line plot**

Figure 3.4: **Memory usage graph of all the jobs when parallelism=11. Both experiments are done on an server with network storage storage.**

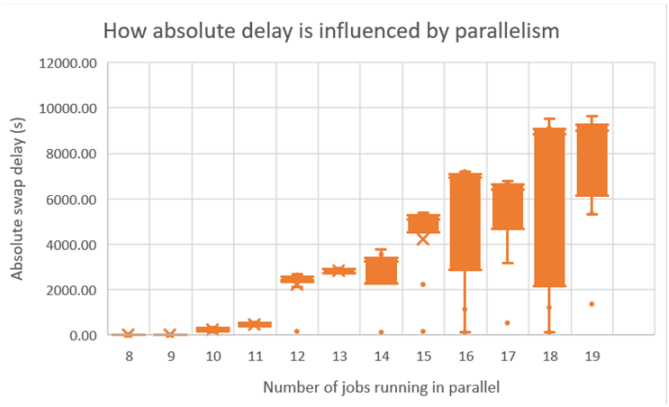


Figure 3.5: **Boxplot with very large cluster job parallelism and system overload, done on network storage and no explicit memory limit set.**

Chapter 4

Parameter Optimizer for the Optimal Cost and Performance

In the context of the job scheduler system, users typically interact with the system through job inputs and outputs, often without complete knowledge of how their chosen parameters, specifically the values of M and R, influence the overall cost and performance on a dedicated server. To bridge this gap and empower users with the ability to make better decisions, we are developing a parameter optimizer. This optimizer will automate the process of identifying the most suitable parameter settings to achieve optimal cost and performance, based on specific job requirements and system conditions.

In our evaluation of performance, we commonly use the reciprocal of job execution time [12] as a metric. Given our emphasis on batch jobs, we use the total execution time of a set of jobs, measured from the start of the first job until the completion of all jobs, to represent the system's performance. This allows us to make meaningful comparisons and optimizations in a batch job context.

In real production environments, users have the flexibility to submit diverse types of jobs at any given time, which can make estimating the cost for each individual job challenging. To simplify the cost estimation, we make the following assumption:

1. We are working with a substantial simulation job set comprising jobs from a shared parametric sweep analysis. A parametric sweep is a widely adopted circuit simulation technique that involves sweeping a specific parameter (e.g., voltage, temperature) through a range of values. This analysis is typically performed for transient, AC, or DC sweep simulations [13]. In that case, all the individual jobs in the job set use a similar amount of memory. This assumption greatly simplifies the task of estimating the cost for each job, as we can straightforwardly compute the average cost across all jobs in the set.

2. We make the assumption that the job set is submitted and executed in batches, and in each batch, we submit as many jobs as the available memory resources can accommodate. This simplification allows for easier

cost calculation and closely emulates the conditions of a fully utilized system in real-world scenarios.

3. Considering the impact of the cloud provider server type on the optimal value of parameter M, we assume that all servers within the cluster are of the same server type. Specifically, we focus on the widely used serverA and serverA_disk server types, which are prevalent in HPC environments at NXP. We make this assumption because different systems may have distinct optimal parameters. Therefore, it is essential to ensure that the evaluation system used to determine the optimal parameters matches the system to which these parameters will be applied.

4. We consider the absence of any concurrent jobs running on our cluster, apart from our dedicated parametric sweep analysis job set. By assuming no interference from other jobs, we can mitigate resource competition and potential performance variations introduced by unrelated tasks.

Our first task is to experimentally investigate how parameters R and M influences cost and performance, aiming to find the parameter combination that brings the best performance (section 4.1). Building upon these findings, we will propose one or more solutions for the parameter optimizer (section 4.2.1) as well as design how the parameter optimizer fit in real cluster (section 4.2.2). Subsequently, we will implement the parametric optimizer (section 4.3) and assess the accuracy of the results and the optimizer's performance, including the cost of the search process (section 4.4). Finally, we will also evaluate the cost-saving benefits resulting from the implementation of the parameter optimizer (section 4.5) and draw relevant and concise conclusions based on our findings (section 4.6).

4.1 How Different Parameter Combinations Influence Cost and Performance

This section presents an experimental study conducted to gain an understanding of how each combination scheme of M and R influences both cost and performance. This study serves as the foundation for the design of our parameter optimizer.

Based on the aforementioned assumption, we can derive the following equation to describe the relationship between cost and performance:

$$Cost_{avg} = \frac{Total\ execution\ time(h) \times Pricing(\$/h)}{Number\ of\ Jobs} \tag{4.1}$$

We have observed that, for a job set of a specific size, the total execution time exhibits a linear relationship with the average cost per job. This finding implies that optimizing the cost is equivalent to optimizing the performance of the job set. To simplify matters, we will henceforth use the average cost as the sole metric.

Table 4.1: **Hardware and software configuration. The system has a total of 64 job slots, with each job utilizing 2 slots. Therefore, the maximum achievable parallelism in the system is 32.**

Input Parameter		System Configuration	
M_1	768 MB	the cloud server Type	serverA
M_2	3072 MB	Storage Type	network storage-only
M_{step}	256 MB	Maximum Available Memory	30 GB
N	5	Maximum parallelism	32
Job_{test}	SimulatorA-test1_SimulatorA test		

4.1.1 Test Set-up

We propose two parameter combination schemes. One involves setting both M and R to the same value. This ensures that the memory usage of each individual job is limited to a specific range. The other scheme involves only setting R and omitting M. In this case, we solely control the job parallelism in the system, allowing the operating system to dynaSimulatorAlly determine which jobs require more memory swapping.

The results presented in Chapter 3.3 suggest that the latter scheme may offer better performance. However, we need to substantiate this through quantitative evaluations.

Table 4.1 provides an overview of the software and hardware configuration utilized in the experiment. The SimulatorA-test1_SimulatorA test case was chosen as the focus of the study due to its significant potential for memory saving. Based on the findings presented in Table 3.4, where it is observed that the job has a total memory usage of 2304 MB and experiences a 70% performance loss when the memory limit (M) is set to 768 MB, we selected a range of memory usages from 768 MB to 3072 MB. This range allows for a significant variation in parallelism, ranging from 10 to 32, and demonstrates noticeable changes in performance.

The system employed in this experiment was the cloud server of type serverA, which is widely utilized within NXP. The system's native memory capacity was 234GB; however, for the purpose of this study, a deliberate limitation was imposed, reducing the available memory of the entire system to 30GB.

Algorithm 1 outlines the procedure for conducting this experiment. In this algorithm, the variable U represents the memory usage value that is set either when both M and R are specified or when only R is specified.

For each memory usage U, we conduct N measurements of the total execution time for a set of p jobs. This approach helps mitigate the impact of measurement result variability. Following the N repetitions, we compute the average cost per job for that specific U.

Algorithm 1: Evaluate cost with different memory usage

Input: Number of measurement repetitions N, memory usage lowerbound M_1, memory usage upperbound M_2, memory usage increment step M_{Step}, pricing per hour P, test case Job_{test}, total available memory in the system m

Output: Average cost per job

for $U = M1$ **to** $M2$ **by** M_{Step} **do**
- $p = \lfloor \frac{m}{U} \rfloor$;
- **for** $i = 1$ **to** N **do**
  - Submit p identical jobs Job_{test} with memory usage U to job queue;
  - **while** *not all jobs are finished* **do**
    - continue;
  - Write total execution time $ET_i(U)$ to log;
- Calculate average cost $cost(U) = \frac{\sum_{i=1}^{N} ET_i(U) \cdot P}{N \cdot p}$

4.1.2 Test Results and Explanation of Results

Figure 4.1 displays two curves depicting the average cost associated with setting both the memory reservation parameter (R) and the job's memory limit (M) to the same value, and the scenario where only R is set. Both curves exhibit a distinctive V-shaped pattern.

The V-shape results from two opposing influences. As R decreases, parallelism increases, which tends to reduce total execution time and cost. Conversely, more intensive swapping occurs with lower R, which leads to an increase in execution time and cost. The varying rates at which these factors change together create a convex curve (a V-shape), initially declining steadily and then rising consistently.

In addition to these curves, we establish a baseline case, which represents the existing practice in NXP's real production environment. In this baseline, users are required to set a certain memory reservation parameter, denoted as R, when submitting jobs to the job queue. This reservation can be either greater than or lower than the actual memory usage. Since we lack precise statistics on the distribution of user reservation values, we empirically assume that the average user input is equal to the real memory usage. By comparing the baseline to the two curves, we observe that both methods can reduce the average cost. However, it's evident that without an explicit memory limit (M), we can achieve even lower costs. This advantage becomes increasingly pronounced when the average memory usage is smaller. This observation aligns with the findings discussed in Chapter 3.3, where we discovered that when the memory limit M for each job is not explicitly specified, the operating system (OS) is able to make more effective swapping decisions. In conclusion, employing only parameter R tends to result in better performance. We will incorporate this conclusion into the design of our parameter optimizer.

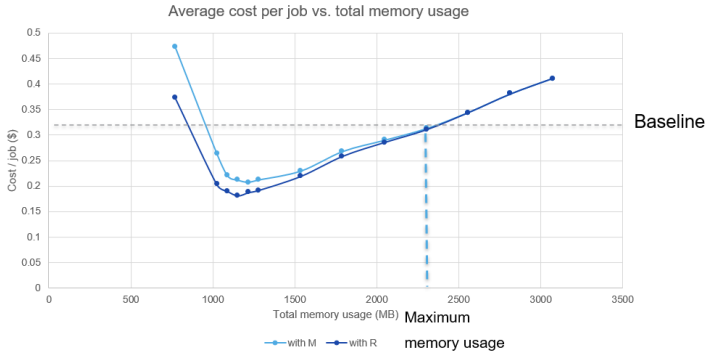


Figure 4.1: **How average cost per job is influenced by memory usage for SimulatorA-TC4-hb respectively. The light blue curve represents the scenario where both R and M are set to the memory usage value on the x-axis. The dark blue curve represents the scenario where only R is set to the value on the x-axis. All the solutions together form a V-shape. The blue vertical dashed line indicates the maximum memory usage of 2304 MB, whereas the gray horizontal dashed line represents the average cost for the baseline.**

4.2 Method description

This section describes the algorithms we employ in the parameter search method as well as the test results.

4.2.1 Existing Parameter Search Methods

From the conclusion in section 4.1.2, it is evident that each R can be associated with a value cost(R), represented by a V-shaped function. This indicates the presence of a unique local minimum (and likely a global minimum) within the search space, implying it is a unimodal function. Therefore, our optimization problem is simplified to finding the minimum of this 1-D unimodal function. Besides, we lack the explicit expression of cost(R) and cannot calculate its derivative. Moreover, since each evaluation can take a considerable amount of time, we need an efficient search method with minimal function evaluations.

Among the available iterative search methods, the golden-section search algorithm and the ternary search algorithm fulfill our requirements. Both algorithms enable efficient search in an unimodal function and are derivative-free. Table 4.2 presents a comparison between the two methods. Notably, the golden-section search demonstrates a higher convergence rate, necessitating fewer iterations to reach the solution. Moreover, leveraging the unique properties of the golden-section ratio, we can reuse probes across iterations, significantly reducing the overall function evaluation cost. Consequently, the golden-section search emerges as the most suitable unimodal search method for our case.

Table 4.2: **Compare two unimodal search methods [8].**

Search Algorithm Name	Interval	Converge rate	Re-usage probes acrosses iterations	Number of function evaluations
Golden-section search	Golden ratio	0.382	yes	$\lceil \log_{0.618} \left(\frac{t}{w}\right) \rceil + 1$
Ternary search	Equally divided in three parts	0.333	no	$2 \cdot \lceil \log_{0.666} \left(\frac{t}{w}\right) \rceil$

4.2.2 How Jobs Land in the System

We propose a two-stage method to search for the optimal R and apply it to a job set containing a large number of jobs, e.g. 1000 jobs. Figure 4.2 shows the workflow of this method. In **stage 1**, we conduct an iterative parameter search on a single server. The search begins with R set to $R_{predict}$, which represents the maximum memory usage predicted by the machine learning model (as described in Chapter 2.5.2). During each iteration, we evaluate the cost of the current R. Jobs are dispatched in a controlled manner, with the number of dispatched jobs determined by the maximum parallelism achievable under the current configuration.

However, the parallelism is not solely constrained by the memory usage per job; it is also limited by the total available job slots in the system. To address this, We use

$$p = min(\frac{total\ job\ slots}{jobs\ slots\ per\ job}, \frac{total\ memory\ resource\ in\ system}{memory\ reservation\ per\ job}) \tag{4.2}$$

to calculate the maximum parallelism. After completing all the jobs, we record the total execution time, and then calculate the average cost using equation 4.1. By comparing the current cost with previous ones, we determine whether we have found the optimal value and should terminate the search or proceed to evaluate the next M (determined by the search algorithm) in the next iteration. It is worth noting that in **stage 1**, each batch of jobs must be executed sequentially, leading to some evaluation time cost.

Upon concluding that we have found the optimal M, we transition to **stage 2**. Here, we can dispatch the remaining jobs to all other worker servers in the cluster, applying the R_{opt} obtained from **stage 1**. This allows the jobs to run in parallel, further optimizing the system's resource utilization and overall performance.

Additionally, to prevent unnecessary re-evaluation of R_{opt} for jobs we have previously assessed, we employ a map to store the mapping relationship between job information, system information, and the corresponding R_{opt} that was identified. When initiating a new job set, we initially check if the recorded R_{opt} for the given job and system information is already available. If it is, we retrieve the previously determined R_{opt} and directly proceed to **stage 2** with this value. Conversely, if it has not yet been established, we perform **stage 1** to ascertain it.

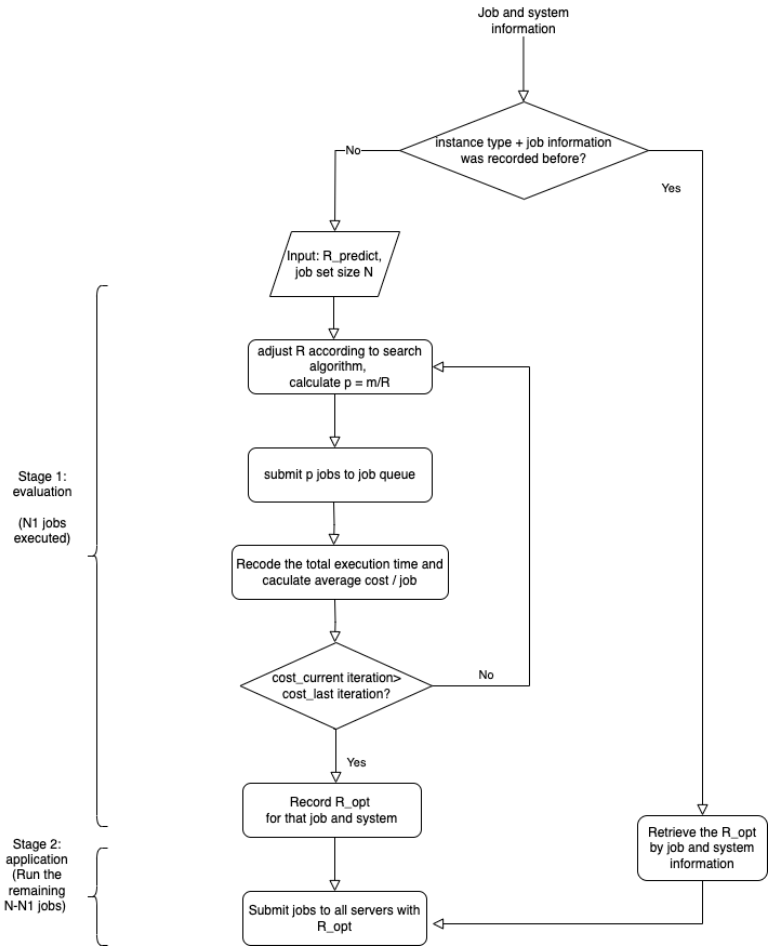


Figure 4.2: **Two-stage method for finding the optimal R_{opt} and apply it in use.**

4.3 Search Algorithm

Based on the analysis presented in section 4.2.1, we have identified the golden-section search algorithm as a potential optimal solution. The Golden Section Search algorithm is an iterative optimization method used to find the minimum of a unimodal function within a given interval. It operates by repeatedly dividing the search interval based on the golden ratio (approximately 0.618) and evaluating the function at two interior points.

Algorithm 2 describes the working principle. The algorithm starts with an initial interval [a, b] and calculates two interior points, c and d, using the golden ratio. It then compares the function values at c and d. If the function value at c is smaller, the interval is updated by setting b to d; otherwise, a is set to c. This process continues until the difference between c and d becomes smaller than the specified tolerance.

By iteratively updating the interval boundaries, the algorithm effectively narrows down the search region and converges towards the minimum of the unimodal function. Finally, it returns the average of the final interval boundaries as the estimated minimum value.

Algorithm 2: Golden Section Search Algorithm [33]

Function `GoldenSectionSearch`($cost(R), a, b, tol$)`:`

- $\phi \leftarrow (1 + \sqrt{5})/2$
- $c \leftarrow b - (b - a)/\phi$
- $d \leftarrow a + (b - a)/\phi$
- **while** $|c - d| > tol$ **do**
  - **if** $f(c) < f(d)$ **then**
    - $b \leftarrow d$
  - **else**
    - $a \leftarrow c$
  - $c \leftarrow b - (b - a)/\phi$
  - $d \leftarrow a + (b - a)/\phi$
- **return** $(a + b)/2$

Input : cost function $cost(R)$, lower bound $a = 0$, upper bound $b = R_{predict}$, tolerance tol

Output: Estimated minimum value

`GoldenSectionSearch`($cost(R), a, b, tol$)

However, we have observed that searching in different regions of the function cost(R) incurs varying time costs. In particular, due to the non-linearlity of the relation between performance and memory reservation R , searching in the left-half of the V-shape in cost(R) function (as presented in Figure 4.1) can lead to significantly higher time costs. Considering that users desire their jobs to be completed in a reasonably short time and cannot afford excessive waiting for the parameter searching, we propose a new algorithm called one-way search (Algorithm 3). This algorithm operates by adjusting the parallelism level and recalculating the predicted memory until an optimal solution is found.

The one-way search algorithm starts with an initial predicted memory value (R) and parallelism level (p) based on the available system memory (m). In each iteration, it evaluates the cost of the current predicted memory. If the cost

increases compared to the previous iteration, indicating a deteriorating solution, the algorithm terminates and returns the previously predicted memory (R_{prev}).

On the other hand, if the cost decreases or remains the same, the algorithm updates the parallelism level by incrementing with 1. It then recalculates the predicted memory by dividing the system memory (m) by the updated parallelism (p). This process continues until a deteriorating solution is encountered.

By employing the one-way search algorithm, we aim to find an optimal solution more efficiently, avoiding extensive searches in regions that are likely to lead to higher time costs and focusing on areas that show potential for performance improvements.

Algorithm 3: One-way Search Algorithm

Function `oneway`($cost(R), R_{predict}$)**:**

$R \leftarrow R_{predict}$
$cost_{prev} \leftarrow \infty$
$R_{prev} \leftarrow \infty$
while *True* **do**
  $p \leftarrow \lfloor m/R \rfloor$
  if $cost(R) > cost_{prev}$ **then**
    return R_{prev}
  $cost_{\text{prev}} \leftarrow cost(R)$
  $R_{\text{prev}} \leftarrow R$
  $p \leftarrow p + 1$
  $R \leftarrow m/p$

Input : cost function $cost(R)$, predicted memory $R_{predict}$, available system memory m
Output: Final predicted memory
`oneway`($cost(R), R_{predict}$)

4.4 Test and Results

In order to evaluate the practicability of the two parameter search algorithms and compare their time cost for the evaluation stage, we designed five groups of configurations to represent different simulation job scenarios, storage types, and job slots, encompassing various potential environment variables encountered in real production environments.

Table 4.3 presents the configuration details of the 5 groups of tests. For the initial search space, we chose a number within the possible interval of $R_{predict}$, as represented by inequality 2.4. The experiments using the storage type "network storage" were conducted on serverA servers, while those with "local NVMe" were performed on serverA_disk servers, both having 234GB available memory in total.

To compensate for the unavailability of large-memory jobs, which have more value and potential for cost saving, we artificially reduced the total available memory size in the system. This ensured that the memory usage per job remained sufficiently large compared to the total system memory.

Additionally, we tested different job slots requests for a specific job to assess whether the parameter optimizer could still function effectively when the

Table 4.3: **Test set-up to validate the parameter search tool. The tests show that the tool can work for different types of jobs and storage types. The total number of available job slots in the system is 64.**

id	Job name	Search space	Storage type	Available memory in system (GB)	Slot per job
1	SimulatorA-test1	[0, 2500] (MB)	network storage	30	2
2	SimulatorA-Chirpllpnoisejitter	[0, 9] (GB)	Local NVMe	50	8
3	SimulatorA-TC6-ADC	[0, 28] (GB)	network storage	100	8
4	SimulatorA-TC6-ADC	[0, 28] (GB)	Local NVMe	100	8
5	SimulatorA-TC6-ADC	[0, 28] (GB)	Local NVMe	100	2

parallelism is constrained by the number of job slots.

Table 4.4 displays the optimal parameter R found by the two algorithms and the time cost of evaluation in 5 tests, which is the average result of three repetitive runs. Overall, the results demonstrate that both algorithms yield similar outcomes in terms of the optimal parameter R, accompanied by the same parallelism. This serves as a mutual verification and proves that both algorithms should be able to give correct results. However, the one-way search algorithm shows a notable advantage in the evaluation stage, requiring less time to reach the optimal solution.

Table 4.4: **Compare the results and performance of golden section search and one-way search.**

id	Golden section search		One way search	
	R_{opt} Found	Extra time cost Eval (s)	R_{opt} Found	Extra time cost Eval (s)
1	1216 (MB)	7,283	1229 (MB)	2,109
2	8.19 (GB)	52,934	8.33 (GB)	18,098
3	23.91 (GB)	13,808	25.00 (GB)	5,767
4	19.78 (GB)	7,234	20.00(GB)	4,503
5	23.91 (GB)	13,218	25.00(GB)	5,655

In addition, we notice that for test 1, the R_{opt} value of 1216GB matches the results shown in Figure 4.1. Furthermore, comparing the R_{opt} values found in tests 3 and 4, we notice that using local NVMe storage allows for more memory saving compared to network storage. Moreover, tests 3 and 5 demonstrate that the parameter search results are not influenced by the per-job job slots setting, provided that the maximum job slots in the system are not the primary resource

constraint.

We also conducted experiments based on the original system memory size (234GB), where we did not notice obvious memory usage saving after optimization due to the fact that the job memory usage is quite small compared to the overall system memory. The results are thus omitted.

4.5 Cost Saving Estimation

Based on the before and after optimization cost values derived from the 5 test cases and the pricing of cloud provider servers in Table 4.5, we calculated the cost savings for each test case over 1000 runs, as shown in Table 4.6. Although we artificially reduced the total memory resources in the system, we assumed that the pricing remained the same.

Table 4.5: **server types and pricing**

server Type	Pricing ($/h)
serverA	x
serverA_disk	x+1.01

By comparing the three SimulatorA jobs, we can conclude that overall, the SimulatorA-test1 job has the most potential for cost saving, which is around 43%. The cost saving for SimulatorA-TC6-ADC is slightly less, ranging from 6.5% to 24%. In comparison, we observed the least cost saving from SimulatorA-Chirpllpnoisejitter, with a cost saving of only 0.9%. This discovery aligns with our summary of job sensitivity in section 3.2.5. Additionally, comparing tests 3 and 4, we noticed that using local NVMe can potentially result in higher cost savings.

Table 4.6: **Cost saving for each test case.**

id	Job Name	original cost/job ($)	optimized cost/job ($)	cost saving per 1000 runs ($)
1	SimulatorA-test1	0.313	0.178	135
2	SimulatorA-Chirpllpnoisejitter	1.322	1.310	12
3	SimulatorA-TC6-ADC	1.175	1.098	77
4		1.401	1.055	346
5		1.175	1.097	78

4.6 Conclusions

Based on all the experimental results above, we can conclude that only using R as the submit-time parameter leads to better performance than using both R and M. Besides, we can conclude that the one-way search algorithm is better than the golden-section search algorithm considered for our case. It is simple, with relatively low evaluation time-cost, and consistently yields correct optimal parameters.

Furthermore, we observed that the number of per-job slots does not significantly impact the parameter search result.

Regarding cost saving, we achieved the highest cost saving with the most memory swap-insensitive job, SimulatorA-test1, reaching 43%. For each 1000 runs, we can save up to $135. On the other hand, with the most memory-sensitive job, SimulatorA-Chirpllpnoisejitter, we only achieved 0.5% cost saving.

The experiments conducted on the original system memory size showed that if the memory usage of a job is too small compared to the total system memory size, then we cannot obtain significant memory cost savings.

Chapter 5

Conclusions and Future Works

In this chapter, we present the method employed to address the problem, showcase our results, and address the central research question. Additionally, we provide suggestions for future research. Currently, NXP is engaged in predicting the maximum memory usage of HPC jobs with the aim of reducing over-reservation of memory resources and achieving cost savings. Building on their efforts, our goal is to further optimize memory usage and cost by exploring the potential of swap space.

The primary objective of this thesis is to identify the most appropriate balance between memory and swap space utilization in HPC systems, ultimately achieving the optimal cost and performance for a homogeneous work set in an overloaded system. By striking this balance, we aim to enhance the efficiency and cost-effectiveness of HPC operations, paving the way for more resource-efficient computing.

5.1 Methods and Results

In this thesis, we pursue three primary objectives. The first objective focuses on comprehensively understanding the impact of memory swapping on HPC system performance. To achieve this, we developed a delay accounting tool to measure various delay fields and conducted extensive measurements using more than ten circuit simulation jobs and two different storage types. The results indicate that the primary contributor to performance degradation when memory swapping occurs is the memory swapping-in delay. Moreover, the sensitivities of different circuit simulation jobs to memory swapping vary, depending on their specific data manipulation methods. A comparison between network storage and local NVMe storage suggests that local NVMe demonstrates superior swapping performance, serving as a useful reference for future server-type selection in cloud providers.

In the second part, we propose two approaches for controlling swap space utilization: one involves setting both R and M (defining the parallelism level and setting explicit memory limits for individual jobs), and the other employs only R (specifying the parallelism level while allowing the OS to determine

swapping priorities). By comparing the performance of these two schemes, the results highlight that using only R leads to better performance. This outcome can be attributed to the OS's ability to optimize memory usage and make more effective swapping decisions in this configuration.

The third part of our thesis aims to develop a practical tool capable of automatically determining the optimal memory reservation parameter R, leading to the best balance between cost and performance. We focus our research on a scenario featuring overloaded systems, a homogeneous work set, identical servers on the cluster, and the absence of irrelevant jobs. These assumptions allow us to derive that the optimal cost and performance can be simultaneously achieved. Drawing on our insights from the investigations, we propose a parameter optimizer that utilizes a one-way search method. This optimizer employs an iterative evaluation and adjustment approach to gradually increase parallelism and minimize the cost (maximize the performance). Our results demonstrate that this parameter optimizer efficiently achieves its objectives with relatively low time costs, resulting in significant cost savings of up to 43%.

5.2 Limitations and Future Works

This section analyzes the limitations in the current works and proposes potential areas for future investigation that could add value to our findings.

5.2.1 Explain the Non-linearity of Delay Curve

We have observed that the delay curve concerning memory limits is not linear. In our measurements so far, we have noticed that as the memory limit decreases, the impact of memory swapping increases more rapidly. Further investigation is required to explain this phenomenon in detail.

5.2.2 Identify the Bottlenecks in the System and Explore the Solutions

The job parallelism experiments have shed light on potential bottlenecks in the system, such as thermal and power constraints, and disk I/O bandwidth limitations. To validate these assumptions, additional experiments can be conducted to specifically assess the impact of these factors on the system's performance.

Conducting experiments with different levels of job parallelism and observing their impact on the system's thermal and power behavior, along with monitoring disk I/O throughput, can provide valuable insights to verify the hypothesized constraints in the system.

Regarding possible solutions, the following approaches can be considered:

- server Type Selection: Choose cloud provider server types that better match your workload requirements. Opting for servers with faster storage options like local NVMe or provisioned IOPS on network storage volumes can help tackle the disk bandwidth bottleneck.

- Parallelism and Resource Management: Optimize the parallelism and resource allocation for your jobs. Fine-tune the number of parallel jobs running on each server to avoid overloading the system. Implementing job

scheduling policies and resource limits can also help to prevent memory overcommitment and unnecessary swapping.

- Caching and Data Prefetching: Utilize caching mechanisms and prefetching strategies to reduce the need for frequent disk I/O. Storing frequently accessed data in memory or caching results can help minimize disk access.

5.2.3 Relate the Features of the Simulation Job to the Delay Output

Currently, we have observed that different jobs exhibit varying sensitivity to memory swap. Unfortunately, due to the lack of access to the source code of these jobs, we are unable to gain deep insights into their specific memory usage patterns at each stage. However, valuable information was obtained from discussions with the SimulatorA team developers. They highlighted that analog simulation jobs are influenced by numerous factors such as circuit size, number of saved signals, and analysis type, making it challenging to identify a common pattern among them. On the other hand, for digital simulations, the situation might be more straightforward, and we could potentially use features like circuit size and simulation length to predict the impact of memory limits. In the future, we plan to conduct additional digital simulation jobs, extract relevant features, and explore the possibility of building a machine learning model to predict sensitivity for these types of jobs. This model can also offer guidance for optimizing the code, making it less sensitive to memory swapping, and thereby improving the potential for cost savings.

5.2.4 Optimize the Parameter Optimizer

Currently, the parameter optimizer incurs significant time costs during the evaluation of obj(M). In the future, we can refine the optimizer to further reduce this time overhead. Possible solutions include improving the search algorithm's efficiency, optimizing the evaluation process to reduce computation time, or leveraging parallel computing by running evaluations on multiple servers simultaneously. Moreover, the current method has certain limitations, such as being restricted to dedicated premises where all servers are uniform, and no other jobs run concurrently. To enhance its versatility and applicability, we can work towards adapting the method to more complex environments that closely resemble real-world scenarios.

5.2.5 Evaluate and compare the cost on network storage and local NVMe

In our initial comparison, we evaluated the swap efficiency of network storage and local NVMe when a single job is running in the system. However, it is important to investigate how the efficiency difference scales with increasing parallelism. Further research can be conducted to quantify the relationship between parallelism levels and the divergence in swap efficiency between the two storage types. Additionally, considering that servers with local NVMe storage may come at a slightly higher cost, it is crucial to explore the trade-off between cost

and swap delay. Quantifying this relationship will provide valuable insights to make informed decisions when choosing between servers.

5.2.6 Obtain more large-memory job cases

Our original intention was to primarily investigate relatively large jobs, specifically those with memory usage larger than 50GB, as they have more potential for cost-saving optimizations. However, due to the time limitation and some other constraints, currently, we were unable to obtain these large job test cases. In the future, we can try to get more large-memory job cases, with which we can do delay-accounting measurements and evaluate if we can save more cost on them with the parameter optimizer.

5.2.7 Implement the Combined System

During our discussions with the Data Analysis team, we collaborated on designing documents to integrate our results with theirs effectively. Unfortunately, due to time constraints, we were unable to implement the complete system. Thus, this remains a potential avenue for future work.

Bibliography

[1] Swap management. https://www.kernel.org/doc/gorman/html/understand/understand014.html.

[2] what is hpc? introduction to high-performance computing.

[3] About job starters. https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=starters-about-job, Jan 2023.

[4] Introduction to ibm spectrum lsf, Jan 2023.

[5] Memory enforcement based on linux cgroup memory subsystem. https://www.bsc.es/support/LSF/9.1.2/lsf_admin/index.htm?cgroup_mem_enforce9111.html~main, 2023.

[6] Dan Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. Machine learning for predictive analytics of compute cluster jobs. *arXiv preprint arXiv:1806.01116*, 2018.

[7] aws. Amazon ec2 instance store. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html.

[8] Eric Cai. The golden section search method: Modifying the bisection method with the golden ratio for numerical optimization. https://chemicalstatistician.wordpress.com/2013/04/22/using-the-bisection-method-with-the-golden-ratio-for-numerical-optimization/.

[9] Wikipedia Contributors. Page (computer memory). https://en.wikipedia.org/wiki/Page_(computer_memory), May 2021.

[10] Jonathan Corbet. Reconsidering swapping. https://lwn.net/Articles/690079/.

[11] Brandon Dunn. *Optimizing high performance computing system's, resource utilization and throughput by leveraging machine learning.* PhD thesis, 2021.

[12] Sumathy Eswaran. Performance measurements and issues: Computer architecture. https://witscad.com/course/computer-architecture/chapter/performance-measurements-and-issues, Nov 2020.

[13] Dennis Fitzpatrick. *Analog design and simulation using OrCAD Capture and PSpice.* Newnes, 2017.

[14] Todd Hoff. Moving hpc to the cloud: A guide for 2020 - high scalability -, Sep 2020.

[15] Gordon Inggs, David B Thomas, George Constantinides, and Wayne Luk. Seeing shapes in clouds: On the performance-cost trade-off for heterogeneous infrastructure-as-a-service. *arXiv preprint arXiv:1506.06684*, 2015.

[16] The kernel development community. Delay accounting — the linux kernel documentation. `https://www.kernel.org/doc/html/v5.4/accounting/delay-accounting.html`.

[17] Nathan Kirsch. Samsung 860 qvo ssd review - 1tb/2tb drives tested, Nov 2018.

[18] Xiuqiao Li, Nan Qi, Yuanyuan He, and Bill McMillan. Practical resource usage prediction method for large memory jobs in hpc clusters. In *Supercomputing Frontiers: 5th Asian Conference, SCFA 2019, Singapore, March 11–14, 2019, Proceedings 5*, pages 1–18. Springer, 2019.

[19] linux kernel contributor. Linux delay accounting. `https://www.kernel.org/doc/html/v5.4/accounting/taskstats-struct.html`.

[20] LSF. How lsf collects memory usage? `https://www.ibm.com/support/pages/how-lsf-collects-memory-usage`, Oct 2018.

[21] Andréa Matsunaga and José AB Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504. IEEE, 2010.

[22] Javy de Koning Matt Morris and Maarten Kelderman. How nxp is moving chip design to aws. `https://aws.amazon.com/blogs/industries/nxp-semiconductors-selects-aws-as-its-preferred-cloud-provider-to-power-silicon-design`

[23] Mohita Narang. Swapping in operating system. `https://www.naukri.com/learning/articles/swapping-in-operating-system`, Nov 2022.

[24] Alberto Núñez, César Andrés, and Mercedes G Merayomg. Optimizing the trade-offs between cost and performance in scientific computing. *Procedia Computer Science*, 9:498–507, 2012.

[25] Jemalo Qiu. Netlink communication between kernel and user space. `https://dev.to/jemaloqiu/netlink-communication-between-kernel-and-user-space-2mg1`, Jan 2022.

[26] M Rezaei, A Salnikov, and A Shiryaev. Developing a toolkit for task characteristics prediction based on analysis of queue's history of a supercomputer. 2021.

[27] Eduardo R Rodrigues, Renato LF Cunha, Marco AS Netto, and Michael Spriggs. Helping hpc users specify job memory requirements via machine learning. In *2016 Third International Workshop on HPC User Support Tools (HUST)*, pages 6–13. IEEE, 2016.

[28] Sean Michael Kerner. What is memory swapping? https://www.enterprisestorageforum.com/hardware/what-is-memory-swapping/, 2019. Last accessed: Jan. 25, 2023.

[29] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in grids. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 117–126, 2007.

[30] Taraneh Taghavi, Maria Lupetini, and Yaron Kretchmer. Compute job memory recommender system using machine learning. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 609–616, 2016.

[31] Mohammed Tanash. *Improving HPC system performance by predicting job resources for submitted jobs using machine learning techniques*. Kansas State University, 2021.

[32] Mohammed Tanash, Daniel Andresen, and William Hsu. Ampro-hpcc: A machine-learning tool for predicting resources on slurm hpc clusters. In *The Fifteenth International Conference on Advanced Engineering Computing and Applications in Sciences ADVCOMP*, pages 20–27, 2021.

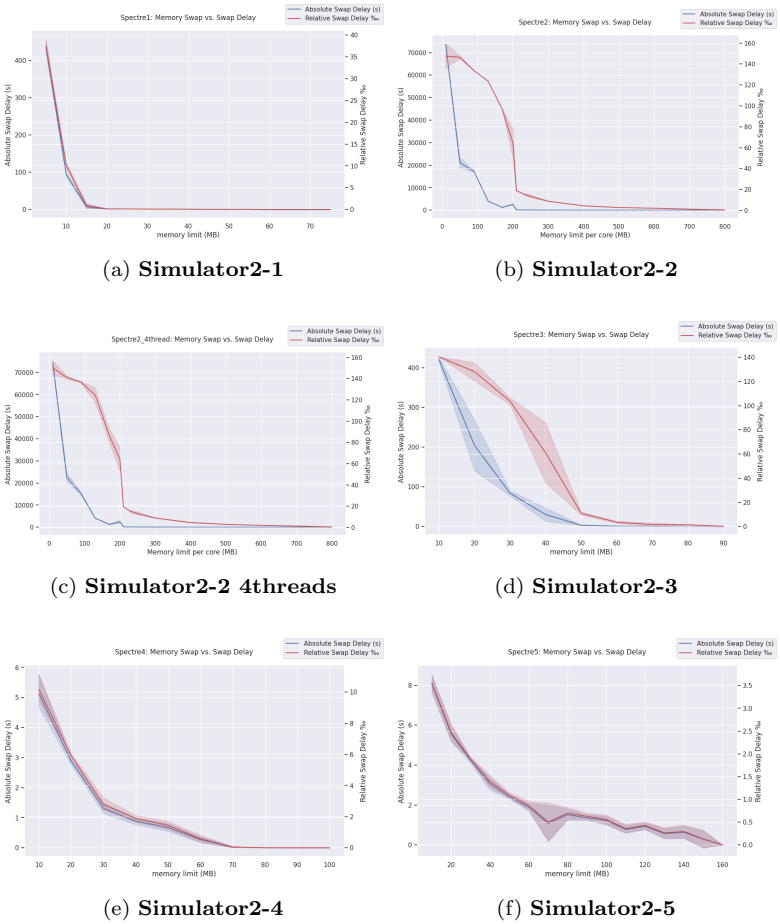
[33] wiki contributor. Golden-section search. https://en.wikipedia.org/wiki/Golden-section_search.

[34] Lawrence Williams. Paging in operating system (os). https://www.guru99.com/paging-in-operating-system.html, Dec 2022.

[35] Zhenyun Zhuang, Cuong Tran, Jerry Weng, Haricharan Ramachandra, and Badri Sridharan. Taming memory related performance pitfalls in linux cgroups. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 531–535. IEEE, 2017.

Appendix A

Simulator2 test results

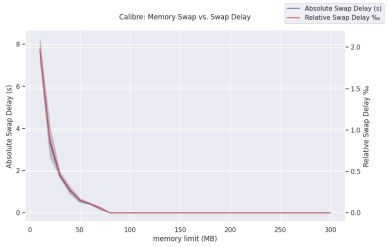


(a) **Simulator2-1** (b) **Simulator2-2**

(c) **Simulator2-2 4threads** (d) **Simulator2-3**

(e) **Simulator2-4** (f) **Simulator2-5**

Figure A.1: **Delay accounting measurement results for benchmark tests by Simulator2.**



(a) **Simulator4**

Figure A.2: **Delay accounting measurement results for benchmark tests by Simulator4.**