A Design and Implementation of user-PC Computing System Platform using Docker

September, 2022

Hein Htet

Graduate School of
Natural Science and Technology

(Doctor's Course)
OKAYAMA UNIVERSITY

Dissertation submitted to
Graduate School of Natural Science and Technology
of
Okayama University
for
partial fulfillment of the requirements
for the degree of
Doctor of Philosophy.

Written under the supervision of

Professor Nobuo Funabiki

and co-supervised by
Professor Satoshi Denno
and
Professor Yasuyuki Nogami

Okayama University, September 2022.

To Whom It May Concern

We hereby certify that this is a typical copy of the original doctor thesis of
Hein Htet

Signature of

the Supervisor

Prof. Nobuo Funabiki

Seal of

Graduate School of

Natural Science and Technology

Abstract

Recently, the computing ability of a *personal computer (PC)* has been greatly increased with the faster CPU clock cycle, the more number of CPU cores, the larger memory size, and the higher storage capacity/access speed. Besides, a PC can be available with a low cost. As a result, the collection of PCs that are owned by users in an organization, called *user-PCs* in this thesis, will provide a significantly efficient computing platform with a very small cost for complex computational projects by running them on the idling resources of user-PCs.

To realize this concept, in this thesis, I present the design and implementation of the *user-PC computing (UPC)* system platform for a low-cost and high performance distributed computing based on the *master-worker model*. The UPC system offers computation capabilities for the members in an organization by using the idling computing resources of their PCs.

In the UPC system platform in this thesis, a user can submit a job from the web browser. Then, the web server sends the job to the *UPC master*. Besides, a user application program can submit a job to the master through the shared storage or the FTP service. When the master accepts a job, it will assign the job to an idling *UPC worker*, and receive the result from it after the completion. After that, the *UPC master* will return the result to the web browser of the user.

As the first contribution of the thesis, I present the design and implementation of the UPC system platform. The *Docker container* technology is adopted in the UPC system to solve environment dependency problems at running various jobs on various PCs. *Docker* is a software tool that has been designed to make it easier to create, deploy, and run an application program on various platforms using the container technology by bundling the required software. The *container* allows the application developer to package up the required software to run the application program, such as libraries, middleware, parameters, and other dependencies, into one package called the *container image*, to be shipped out. By using *Docker*, the UPC system allows various jobs or applications to run on user-PCs as workers with different platforms and environments.

As the second contribution of the thesis, I implement the *UPC web interface* using *HTML*, *CSS*, and *JavaScript*. The *web server* is implemented using *Node.js* that is the open source server that offers the running environment of *JavaScript* programs. The users can easily submit the job and download the results by accessing to the *web interface* through the web browser.

As the third contribution of the thesis, I implement the two online job acceptance functions. Previously, the UPC system only accepts jobs from local users manually using the web interface and the UPC web server runs in private networks due to cost and security concerns. Some application systems will need to submit the jobs automatically to the UPC system for speeding up their job processing time. The first function uses *Secure Shell File Transfer Protocol (SFTP)* to interact the *UPC web server* with the files in the application system over the reliable data stream. The second function uses *cloud storage* to share the files with the application system. For evaluations, we adopt *Android programming learning assistance system (APLAS)* and *Exercise and performance learning assistant system (EPLAS)* as the application systems, which require the more CPU ability

and have been developed in our group. The experiment results show that all jobs from the both systems are successfully accepted and the total CPU time is reduced by 90.5% for *APLAS* and 55.1% for *EPLAS* of the original, respectively.

As the fourth contribution of the thesis, I implement the *job migration function*. By adopting this *job migration*, the UPC system can accelerate the job completion by dynamically migrating or moving it to a faster PC when it becomes idling. The function is implemented using the two open-source software named *Checkpoint Restore in Userspace (CRIU)* and *Podman*. CRIU can save all the data related to the running job into image files, which is called *Checkpointing*. *Podman* can manage the containers with the same permissions as the user who launched the containers. *CRIU* can create the checkpoint and restore the running containers (*jobs*.) To evaluate the implemented function, we conducted extensive experiments using the testbed UPC system with 14 jobs, six workers of different specifications, and three job scheduling algorithms considering the migration. In these experiments, first, we verified the validity of the implemented migration function. Then, we confirmed the effectiveness of the function by comparing the job completion performances when three scheduling algorithms were adopted. The results show that the total CPU time and *makespan* by the algorithms are significantly reduced by improving the resource utilizations and balancing the workloads of the workers through the dynamic migration.

As the last contribution of the thesis, I implement the *job running backup function*. Some jobs may require the long CPU time to be completed. Then, it can increase the probability of causing a failure of the worker while running a job. To avoid it, the current state of running the job on the worker should be automatically backed up so that the job should run from the backed up state on another healthy worker. *CRIU* is periodically applied to capture the job running state of the running job at a worker and it is controlled by the *Python* script. When the master detects the failure, it automatically migrates the job to another worker. To evaluate the function, we conducted experiments using the testbed UPC system with 14 jobs and six workers of different specifications, and confirmed that the proposal successfully resumes the job running from the interrupted point at another worker.

The automatic *Docker* image generation for a newly submitted job, the use of GPU devices for *workers*, the automatic join/leave of *workers*, and the collaboration of multiple *masters* for the scalable UPC system will be in future works.

Acknowledgements

It is my great pleasure to express my heartiest thankfulness to those who gave me the valuable time and supported me in making this dissertation possible. I believe that you are the greatest blessing in my life. Thanks to all of you for making my dream successful.

I owe my deepest sense of gratitude to my honorable supervisor, Professor Nobuo Funabiki for his excellent supervision, meaningful suggestions, persistent encouragements, and other fruitful help during each stage of my Ph.D. study. His thoughtful comments and guidance helped me to complete my research papers and present them in productive ways. Besides, he was always patient and helpful whenever his guidance and assistance were needed in both of my academic and daily life in Japan. I have really been lucky in working with a person like him. Needless to say, it would not have been possible to complete this thesis without his guidance and active support.

I am indebted to my Ph.D. co-supervisors, Professor Satoshi Denno and Professor Yasuyuki Nogami, for taking their valuable time to give me advice, guidance, insightful comments, and proofreading of this thesis.

I want to express my profound gratitude to Associate Professor Minoru Kuribayashi in Okayama University for his valuable discussions during my research. I would like to thank all my course teachers during my Ph.D. study for sharing their great ideas and knowledge with me. I would also like to give special thanks to Professor Wen-Chung Kao of National Taiwan Normal University and Professor Shinji Sugawara of Chiba Institute of Technology, for their advices and supports, especially in proofing my papers before submissions.

I would like to acknowledge the Ministry of Education, Culture, Sports, Science, and Technology of Japan (MEXT) for financially supporting my Ph.D. study at Okayama University.

I would like to convey my appreciations to all the members of FUNABIKI Lab. Especially, I would like to give special thanks to Dr. Nobuya Ishihara, Ms. Keiko Kawabata, Dr. Htoo Htoo Sandi Kyaw, Mr. Ariel Kamoyedji, Mr. Xudong Zhou, Mr. Xu Xiang, Dr. Yan Watequlis Syaifudin, Dr. Kwenga Ismael Munene, Dr. Pradini Puspitaningayu, Dr. Md Mahbubur Rahman, Ms. Soe Thandar Aung, Ms. Ei Ei Htet, Ms. Shune Lae Aung, Ms. San Haymar Shwe, Mr. Lynn Htet Aung, Ms. Irin Tri Anggraini, Mr. Sujan Chandra Roy, Mr. Yuanzhi Huo, and all members in general. To share the time with these people in Okayama gave great experiences and unforgettable moments to me. Thank you so much for your great supports and helpfulness in this study and my daily life.

I would also like to express my special gratitude to my Japanese Language teachers, especially Dr. Yin Moe Thet, Dr. Rie Kuroe, and Dr. Miho Sato for their dedicated teaching.

Last but not least, my special thanks go to my beloved parents, and brother who always encourage and support me throughout my life.

Hein Htet
Okayama University, Japan
September 2022

List of Publications

Journal Papers

1. **Hein Htet**, Nobuo Funabiki, Ariel Kamoyedji, Minoru Kuribayashi, Fatema Akhter, and Wen-Chung Kao, "An implementation of user-PC computing system using Docker container," International Journal of Future Computer and Communication (IJFCC), Vol. 9, No. 4, pp. 66-73 (2020).

International Conference Papers

2. **Hein Htet**, Nobuo Funabiki, Ariel Kamoyedji, Xudong Zhou, and Minoru Kuribayashi, "An implementation of job migration function using CRIU and Podman in Docker-based user-PC computing system," 9th International Conference on Computer and Communications Management (ICCCM 2021), pp. 92-97 (Singapore, Singapore, 2021).

3. **Hein Htet**, Nobuo Funabiki, Ariel Kamoyedji, Xudong Zhou, Yan Watequlis Syaifudin, Irin Tri Anggraini, and Minoru Kuribayashi, "Implementations of online job acceptance functions in user-PC computing system," IEEE 4th Global Conference on Life Sciences and Technologies (LifeTech 2022), pp. 121-122 (Osaka, Japan, 2022).

4. **Hein Htet**, Nobuo Funabiki, Ariel Kamoyedji, Xudong Zhou, Xu Xiang, Shinji Sugawara, and Wen-Chung Kao, "An implementation of job running backup function in user-PC computing system," IEEE 4th International Conference on Computer Communication and the Internet (ICCCI 2022), pp. 156-161 (Chiba, Japan, 2022).

Other Papers

5. **Hein Htet**, Nobuo Funabiki, Ariel Kamoyedji, and Minoru Kuribayashi, "Design and implementation of improved user-PC computing system," IEICE Technical Report, NS2020-28, pp. 37-42 (Online, Japan, 2020).

List of Figures

1.1 Overview of UPC system. 2

2.1 Usage of Docker in UPC system. 6
2.2 Docker image generation process at master. 7
2.3 Memory usage rate without job control. 9
2.4 Memory usage rate with job control. 9

3.1 Operation flow of job submission and results accessing. 11
3.2 UPC web server platform. 12
3.3 Uploaded jobs and their processing status on UPC web interface. 13
3.4 Upload and download files on UPC web interface using *fs* module. 13
3.5 Use of *JavaScript* functions for web interface submission. 14

4.1 UPC online job acceptance using SFTP. 16
4.2 UPC online job acceptance using cloud storage. 16

5.1 State capturing at PC. 20
5.2 State restoring at new PC. 20
5.3 Run Docker container on Windows with WSL. 22
5.4 Job migration between same OS worker PCs. 22
5.5 Scenarios for validating migration function. 25
5.6 CPU time of job migration from PC1. 25
5.7 CPU time of job migration from PC2/PC3. 25
5.8 CPU time of job migration from PC4. 26
5.9 CPU time of job migration from PC5. 26
5.10 CPU time of job migration from PC6. 26
5.11 FCFS_L2H algorithm with and without migration. 28
5.12 FCFS_H2L algorithm with and without migration. 29
5.13 Hjob_L2H algorithm with and without migration. 29

6.1 Job check-pointing at worker and backup saving at master. 32
6.2 Noticing worker failure and job restoration at healthy worker. 32

List of Tables

4.1 PC specifications in experiments. 17
4.2 CPU time for *APLAS*. 17
4.3 CPU time for *EPLAS*. 18

5.1 Worker specifications. 23
5.2 Job specifications. 24
5.3 Jobs standard processing time. 24
5.4 Total CPU time improvement rates by migration. 27
5.5 Comparison of with and without migration for 10 random cases. 28
5.6 Analysis of CPU time and makespan results. 28
5.7 Migrated job conditions. 30
5.8 Makespan results (H:M:S). 30
5.9 Total CPU time results (H:M:S). 30

6.1 Number of checkpoints and average loads. 35
6.2 CPU time for two jobs (H:M:S). 35

Contents

Abstract i

Acknowledgements iii

List of Publications v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Contributions . . . 2
  - 1.2.1 Docker based UPC System Platform . . . 2
  - 1.2.2 Web Interface Submission . . . 2
  - 1.2.3 Online Job Acceptance . . . 2
  - 1.2.4 Dynamic Job Migration . . . 3
  - 1.2.5 Job Running Backup . . . 3
- 1.3 Contents of This Dissertation . . . 3

2 Implementation of Docker based UPC System Platform 5
- 2.1 Web Server . . . 5
  - 2.1.1 Three Basic Functions . . . 5
- 2.2 UPC Master . . . 5
  - 2.2.1 MySQL and Docker . . . 6
  - 2.2.2 Four Basic Functions . . . 6
  - 2.2.3 Docker Image Generation . . . 6
  - 2.2.4 Worker Management . . . 7
- 2.3 UPC Worker . . . 8
  - 2.3.1 Five Basic Functions . . . 8
  - 2.3.2 Job Control Function . . . 8
- 2.4 Summary . . . 9

3 Implementation of Web Interface Submission 11
- 3.1 Operation Flow . . . 11
- 3.2 Web Server Platform . . . 12
  - 3.2.1 HTTP Web Server . . . 12
  - 3.2.2 Web Interface . . . 12

3.2.3 File System Control . . . 13
3.3 Summary . . . 14

4 Implementation of Online Job Acceptance Function 15
4.1 Online Job Acceptance Function . . . 15
4.1.1 Online Job Acceptance Function using SFTP . . . 15
4.1.2 Online Job Acceptance Function Using Cloud Storage . . . 16
4.2 Evaluation of Online Job Acceptance Function . . . 16
4.2.1 Evaluation Setup . . . 16
4.2.2 Results for APLAS Jobs with SFTP . . . 17
4.2.3 Results for EPLAS Jobs with Cloud Storage . . . 18
4.3 Summary . . . 18

5 Implementation of Job Migration Function 19
5.1 Job Migration Function . . . 19
5.1.1 Job Migration Process . . . 19
5.1.1.1 State Capturing . . . 19
5.1.1.2 State Restoring . . . 20
5.1.2 Software Tools . . . 21
5.1.3 Worker Search for Migration . . . 21
5.1.4 Procedure of Job Migration Function . . . 21
5.1.5 WSL Layer in Windows PC . . . 22
5.1.6 Job Migration Limitation . . . 22
5.2 Evaluation of job migration function . . . 23
5.2.1 Evaluation Setup . . . 23
5.2.2 Validity of Migration Function . . . 23
5.2.3 Job Scheduling Algorithms with and without migration . . . 28
5.2.3.1 Analysis of Migrated Jobs . . . 29
5.2.3.2 Makespan Results . . . 29
5.3 Summary . . . 30

6 Implementation of Job Running Backup Function 31
6.1 Job Running Backup Function . . . 31
6.1.1 Job Check-Pointing Process . . . 31
6.1.2 Job Restoring Process . . . 32
6.1.3 Software Tools . . . 33
6.1.4 Worker Search for Checkpoint Restoration . . . 33
6.1.5 Procedure of Job Running Backup Function . . . 33
6.2 Evaluation of job running backup function . . . 34
6.2.1 Correctness of Checkpoint Execution . . . 34
6.2.2 Correctness of Job Running Backup . . . 34
6.3 Summary . . . 35

7 Related Works in Literature 37

8 Conclusion 41

References 43

Chapter 1

Introduction

1.1 Background

Recently, the computing ability of a *personal computer (PC)* has been greatly increased with the faster CPU clock cycle, the more CPU cores, the larger memory size, and the higher storage capacity/access speed. PCs are usually available with very low costs. As a result, the collection of PCs that are owned by users in an organization, called *user-PCs* in this thesis, will provide a significantly efficient platform with very small costs for complex computational projects by running them on their idling resources. To realize this concept, we have studied the *User-PC Computing (UPC)* system as a low-cost and high performance distributed computing platform based on the *master-worker model* [1].

The UPC system offers high computational powers for the members in the organization by using the idling computing resources of their PCs [2]. In addition, the UPC system is different from the *Volunteer Computing (VC)* system [3], it can achieve the high dependency by using the trusted PCs in the same organization or group. Figure 1.1 illustrates the UPC system overview. In the UPC system, a user may submit a job from the web browser to the web server. Then, the web server sends the job to the *UPC master*, which will assign the job to a *UPC worker* and receive the result from it after the completion. After that, the *UPC master* will return the result to the web browser of the user.

The UPC system allows various application programs to run on various PC environments for workers using *Docker* [4]. *Docker* is a popular software tool that has been designed to make it easier to create, deploy, and run an application program on various platforms using the *container technology* [5]. The *Docker container image* is a lightweight, standalone, and executable package containing all the software that need to run the application program. It includes the source codes, the runtime environments, the system tools, the system libraries, and the setting parameters.

The usage flow of the UPC system consists of seven steps: 1) a user submits *computing projects (jobs)* from the Web browser to the *UPC master* via the Web server, 2) the master generates the *Docker image* for each job, 3) the master finds the schedule of assigning the jobs to the *UPC workers*, 4) the master transmits the *Docker images* of the jobs to the scheduled workers, 5) the UPC worker computes the assigned job using the *Docker container* and transmits the result to the master when it is finished, 6) the master receives the job result from the worker, and 7) the master returns the project result to the user when it receives the results for all the jobs from workers.

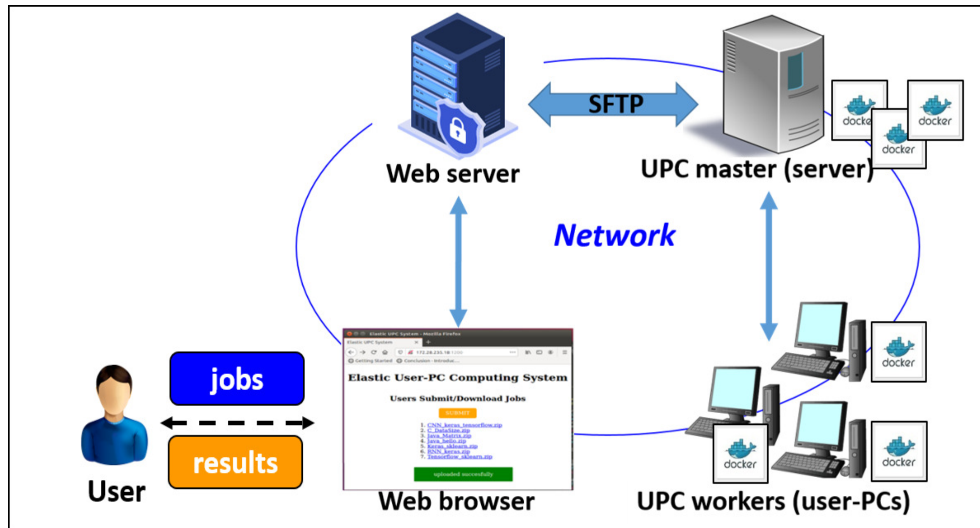

Figure 1.1: Overview of UPC system.

1.2 Contributions

In this thesis, I have carried out the following research contributions.

1.2.1 Docker based UPC System Platform

The first contribution of the thesis is the design and implementation of the UPC system platform using *Docker* [1]. By adopting *Docker*, the UPC system allows various jobs or applications to run on user-PCs as workers with different platforms and environments. There are three main components in the UPC system: the *web server*, the *UPC master*, and *UPC workers*. The user submitted jobs are transformed into the container based jobs, so called, *Docker image* at the *UPC master*. The *Docker* image consists of all the necessary software to run the application program (*job*), such as the libraries, the middleware, the parameters, and the other dependencies, and this images are distributed to the *UPC workers* for carrying out the execution process.

1.2.2 Web Interface Submission

The second contribution is the implementation of the web interface in the UPC system for job submissions by the users [4]. The *UPC web interface* is implemented using *HTML*, *CSS*, and *JavaScript* for submitting the jobs and downloading the results by the user. It is hosted on the *web server*, which is implemented using *Node.js* [6]. *Node.js* is the open source server that can run on multi-platforms of *Linux*, *Windows*, and *Mac OS*, and offers the running environment of *JavaScript* programs on the server.

1.2.3 Online Job Acceptance

The third contribution is the implementation of the two online job acceptance functions for submitting the jobs by the outside application systems [7]. For the first function, we adopt *SSHFS* as the *filesystem client* [8]. The application system needs to install *ssh* client, creates job and result directory, and allows *port 22*. Then, the web server can interact with the remote file system via

SFTP for accessing and transferring files over reliable data streams. For the second function, we adopt *pCloud* as a free cloud storage service [9]. Directories of the jobs and the results are made in *pCloud* where the application system can freely access to them via *http protocol*. For the file access to *pCloud API* from the web server, the downloading and uploading functions are implemented by *JavaScript* at the web server.

1.2.4 Dynamic Job Migration

The fourth contribution is the implementation of the *job migration function* [10]. By adopting this *job migration*, the UPC system can accelerate the job completion by dynamically migrating or moving it to a faster PC when it becomes idling. It is implemented using the two open-source software, named *Checkpoint Restore in Userspace (CRIU)* and *Podman*. CRIU can save all the data related to the running job into image files, which is called *Checkpointing*. *Podman* can manage the *Docker containers*. Besides, since these tools have been developed for *Linux OS*, *Windows Subsystem for Linux (WSL)* is added at the additional layer to cover for the PCs running on *Windows OS*. *WSL* enables the PC to run *Linux* distributions directly on *Windows10* alongside *Windows* applications, without the overhead of a traditional virtual machine.

1.2.5 Job Running Backup

The fifth contribution is the implementation of the *job running backup function* [11]. Some jobs may require the long CPU time to be completed. Then, it can increase the probability of causing a failure of the worker while running a job. To avoid it, the current state of running the job on the worker should be automatically backed up so that the job should run from the backed up state on another healthy worker. *CRIU* is periodically applied to capture the job running state of the running job at a worker and it is controlled by the *Python* script. When the master detects the failure, it automatically migrates the job to another worker.

1.3 Contents of This Dissertation

The remaining part of this thesis is organized as follows: Chapter 2 presents the design and implementation of the UPC system platform using *Docker*. Chapter 3 presents the implementation of the web interface for job submissions. Chapter 4 presents the implementations of the online job acceptance functions and the evaluations. Chapter 5 presents the implementation of the job migration function and the evaluations. Chapter 6 presents the implementation of the job running backup function and the evaluations. Chapter 7 reviews relevant works in literature. Finally, Chapter 8 concludes this thesis with some future works.

Chapter 2

Implementation of Docker based UPC System Platform

In this section, we present the design and implementation of UPC system platform using Docker [1]. The UPC system is composed of the three components, the *web server*, the *UPC master*, and *UPC workers*, shown in Figure 1.1. The implementations of the basic functions in each component will be discussed.

2.1 Web Server

The web server is implemented by using *Node.js* [6]. *Node.js* is an open source server environment and can run on various platforms including Linux, Windows, and Mac OS. It offers the running environment of JavaScript programs on the server [6]. Thus, the following three basic functions are implemented by *JavaScript* programs.

2.1.1 Three Basic Functions

In the web server, the following three functions are implemented with different threads.

- The *job acceptance thread* accepts the jobs submitted from the web browser. One job usually consists of the source codes, the required platforms, and the library lists.
- The *job transmission thread* transfers the submitted jobs to the UPC master using *SSH File Transfer Protocol (SFTP)* [8].
- The *result reception thread* receives the results of the jobs from the UPC master and stores them so that user can download them.

In our implementation, the *Linux OS* is adopted. The built-in module in *Node.js* is used to listen to the server ports and give the responses to the UPC master. The browser page programs are implemented using HTML5, CSS, and JavaScript.

2.2 UPC Master

The programs in the UPC master are implemented using *Python*. The *Python* multi-threaded module supports powerful and high-level threads [12]. Multiple workers are connected with the UPC

master, where one thread in the server program is allocated to each worker.

2.2.1 MySQL and Docker

MySQL [13] is adopted as the database system to manage the data of the UPC system.

The *Docker* container technology [14] is used to provide the flexibility and portability for running various jobs on different worker platforms. It builds the *Docker* image to offer the software environment for running each job, including source codes, libraries, middle ware, and parameters, so that the job can run on any worker PC without considering the installed software.

2.2.2 Four Basic Functions

In the UPC master, the following four basic functions are implemented with different threads.

- The *job management thread* receives the request for a new job from the web server by detecting the newly updated files using SFTP. Then, it prepares a new job by unzipping, inserting and modifying the *Docker file template*, and builds and saves the complete job running environment.
- The *worker management thread* receives the joining request from a UPC worker. When the master receives the request, it creates a new thread for the new worker, collects the information on the worker, and stores them at the master' database.
- The *job transmission thread* sends a job in the job queue to the assigned worker. It is repeated until the job queue is empty.
- The *result uploading thread* sends the result from the worker to the web server using SFTP.

2.2.3 Docker Image Generation

The UPC master accepts the jobs from the web server. Then, for each job, it prepares the *Docker file* that contains the list of the instructions to build the *Docker image* that bundles the environments and the applications, and executes it as a *Docker container*, shown in Figure 2.1. In our implementation, the *Docker file* is automatically created by analyzing the list of the requirements for the job from the user and the extensions of source codes.

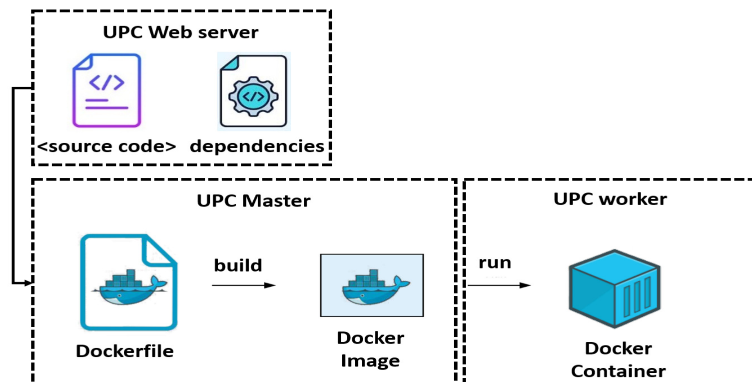

Figure 2.1: Usage of Docker in UPC system.

Figure 2.2 shows the details of the process. The UPC master performs the following steps to generate the Docker Image for each submitted job.

1. It unzips the job, examines the program type, and explores the requirement list.
2. It compares and checks the information obtain in step 1 with the log data under the temporary information directory that stores the previously built *Docker* image information.
3. It refers the previous built *Docker* image if the running environment, libraries, and dependencies are almost similar with the current job's requirements.
4. If not, it refers the base image of the previously built *Docker* image when only the running environment is same.
5. Otherwise, it generates a new *Docker* image for the current job by following the instructions of the generated *Docker file*.
6. It accesses to *Public Remote Repository* to download and install the necessary images, libraries, and platforms, and chooses the small and light package to reduce the image size to a minimum.
7. It saves them as a *Docker* image when successfully finished, and adds it in the correspondence job list.

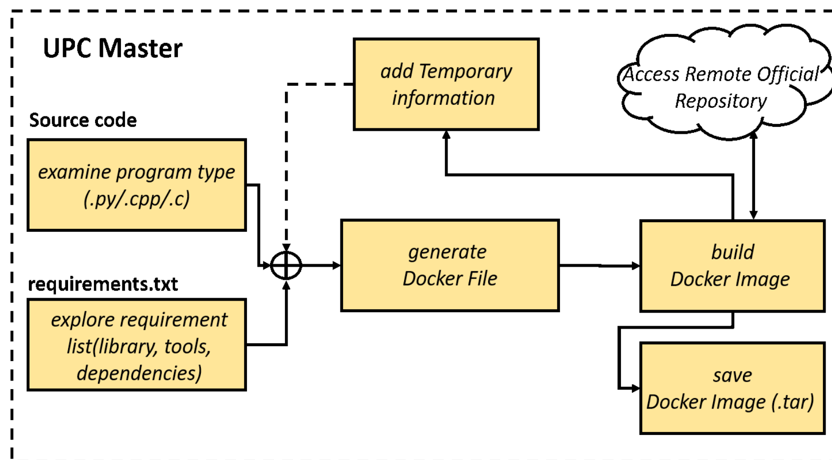

Figure 2.2: Docker image generation process at master.

2.2.4 Worker Management

When a PC joins the UPC system as a new worker, the UPC master collects the static performance-related information of the PC, such as the memory size, the CPU clock rate, the number of cores, and the hard disk size using *psutil* [15]. The master also periodically collects the dynamic performance information of the PC, such as the percentage of the current resource usage and the available resource status. The UPC master keeps all the information in the database. Thus, if the worker cannot keep running the job because of the resource usage shortage, the UPC master can send the stop alert of the running job to the worker, and the resume alert when resources are available to use.

2.3 UPC Worker

The programs of the UPC worker are also implemented using *Python*. The *Docker* container technology is installed to run the *Docker* image for each job on the worker assigned by the UPC master.

2.3.1 Five Basic Functions

The following five basic functions of the UPC worker are implemented with different threads:

- The *connection initiation thread* finds the address and the port of the UPC master from the socket. Then, the worker is connected to the master by sending the necessary information.
- The *job reception thread* receives the *Docker* image for the job with the *.tar* file and temporarily allocates it in the disk space of the worker.
- The *job execution thread* starts to load and run the received *Docker* image as a container.
- The *job restoring thread* saves the current running states of the jobs in the hard disk and sends the state to the master when the worker runs out all the available resources.
- The *result transmission thread* transfers the result of the job when successfully completing it.

2.3.2 Job Control Function

In the UPC system, any running job on a worker must not disturb the use of the PC by the owner. Thus, the job control function is implemented to stop the running container job and free the memory when the memory usage rate exceeds the given threshold, where 90% is selected from our experiment results [4]. The suspended job would be reloaded to the memory for resuming the job when it falls below the threshold. We discuss the implementation of worker PC memory control on *Linux* or *Windows* operating system.

First, we discuss the implementation for *Linux*. ***'kill'*** command is used to stop the job. Then, ***'kill -STOP #ProcessName'*** command is used to free the memory. If the job can run there again, ***'kill -CONT #ProcessName'*** command is used to resume the job.

Next, we discuss the implementation for *Windows*. ***'taskkill'*** command is used to stop the job. Then, ***'Stop-Process -Name #ProcessName'*** command is used to free the memory. If the job can run there again, ***'Cont-Process -Name #ProcessName'*** command is used to resume the job.

Figure 2.3 shows the change of the memory usage rate of the *Convolutional Neural Network (CNN)* job program. The PC does not work properly at the fourth run. When it exceeds 90%, the PC is hung up and needs to be rebooted, where all the running processes are lost. Therefore, the memory usage rate for the UPC job must be carefully controlled to avoid the problem.

Figure 2.4 shows the change of the memory usage rate when the same *CNN* job program runs on the PC five times. Every time the rate exceeds the given threshold 90%, the job is automatically stopped and about 36% of the memory is released to keep running daily processes by the PC owner.

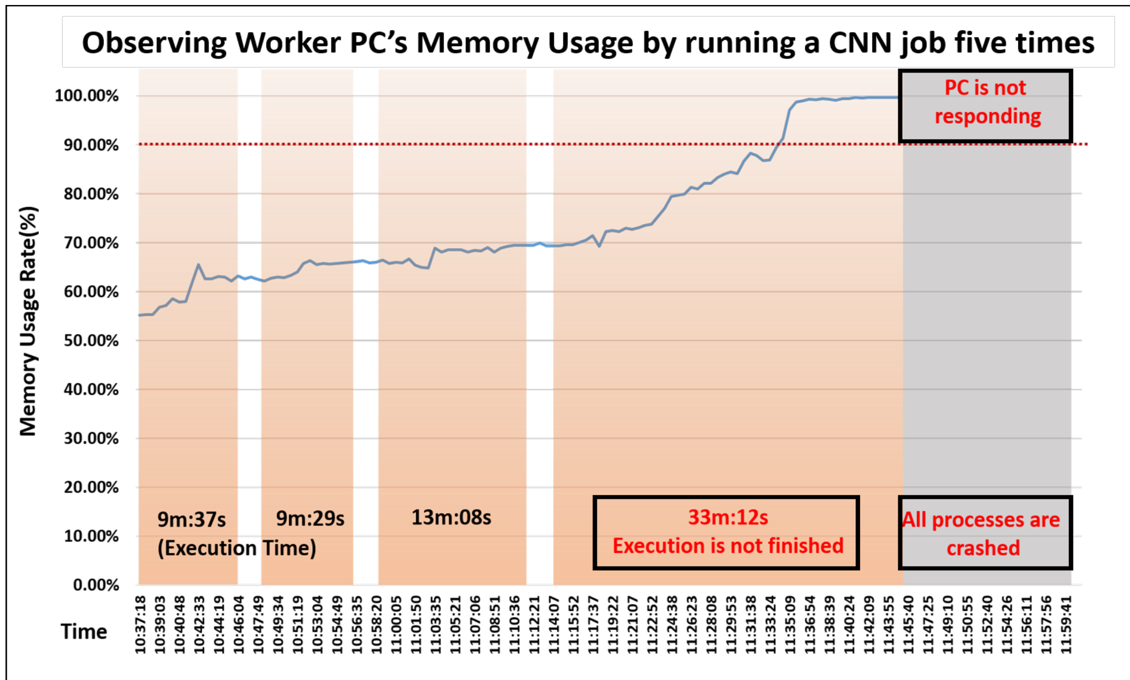

Figure 2.3: Memory usage rate without job control.

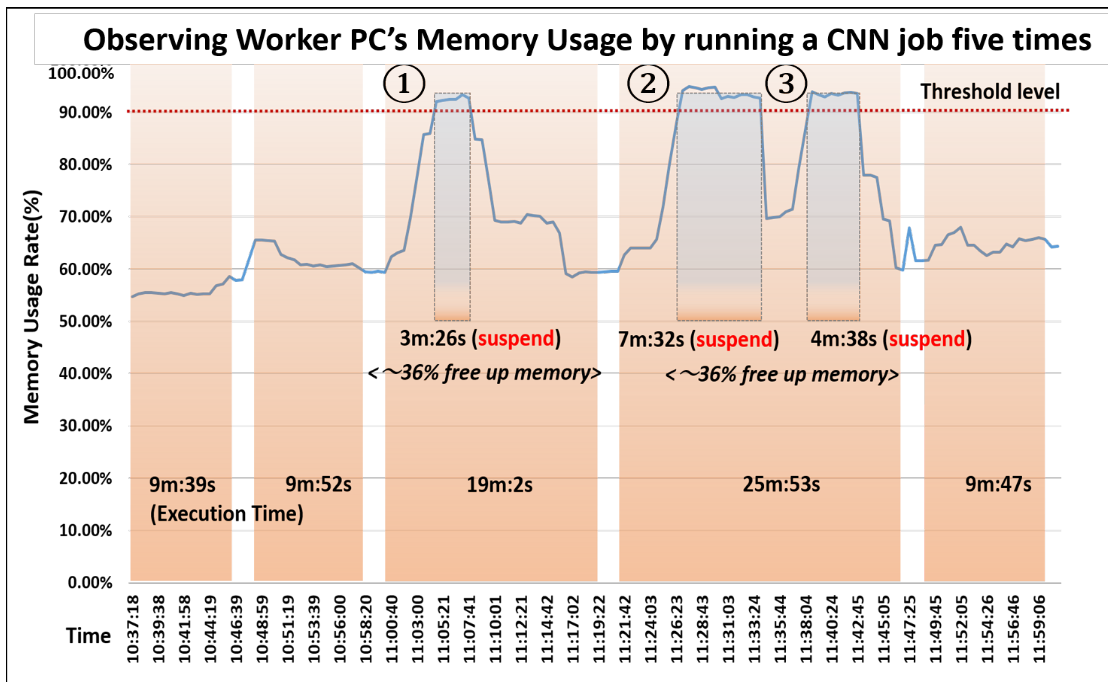

Figure 2.4: Memory usage rate with job control.

2.4 Summary

In this chapter, we presented the design and implementation of UPC system platform using *Docker*. We discussed the job control function and it can be avoided losing the PC owner's processes due to the running of the UPC jobs while the memory usage is high. In the next chapter, we will present the implementation of *UPC web interface* for job submission.

Chapter 3

Implementation of Web Interface Submission

In this section, we present the implementation of the UPC web interface. The users of the UPC system can submit the jobs and download the results by accessing to this web interface through the web browser. The implementation details will be discussed.

3.1 Operation Flow

Figure 3.1 illustrates the overview of the operation flow for submitting the jobs and accessing the results using the UPC web interface.

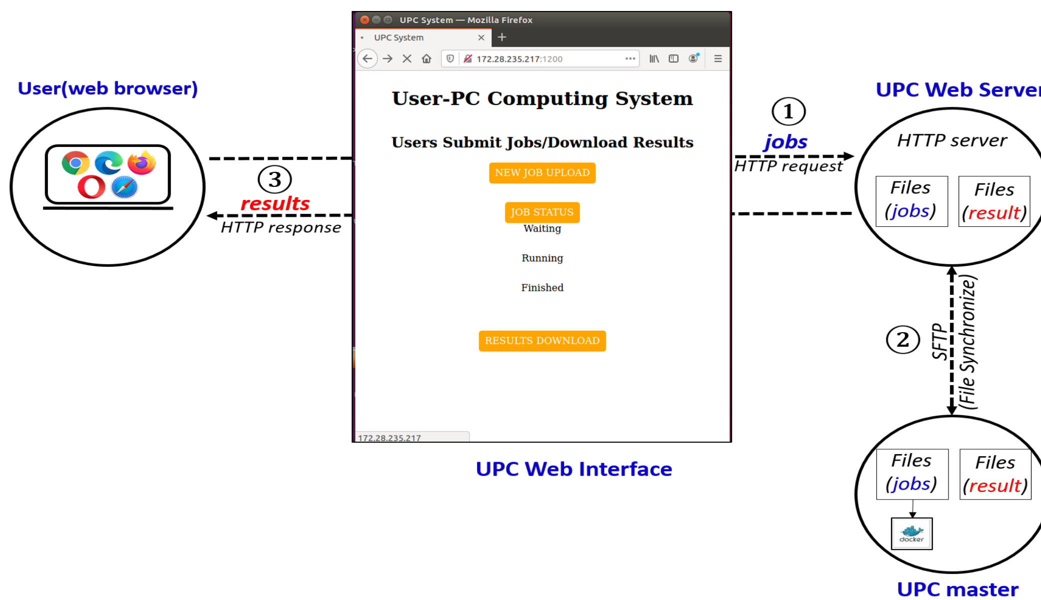

Figure 3.1: Operation flow of job submission and results accessing.

1. *Upload Job:* the user sends the *http* request to the web server for uploading the jobs by calling the web interface. The jobs can be uploaded by clicking the *NEW JOB UPLOAD* button. The *jobs* and *result* directories are prepared at the web server storage for keeping the user uploaded jobs and the results from the master.

2. *Synchronize Job and Result:* the *jobs* and *results* directories at web server are mounted with the directories of the file system at the *UPC master*. Thus, the jobs and result files can be synchronized between the web server and the *UPC master* using the *SFTP* protocol. The UPC master will transform the submitted jobs into the container-based jobs using *Docker* for processing at the workers and keeps the results from them.

3. *Accessing Result:* The *http* response will be sent to the user to download the results when the master receives the results. The result files are synchronized between the UPC master and the web server. The files appear on the web interface under the *RESULTS DOWNLOAD* label. The user can easily download them by clicking the result files.

3.2 Web Server Platform

For the web server platform, we adopted the *Linux* for the operating system, *Node.js* for the framework of implementing the *HTTP* web server and running *JavaScript* programs on the server, *HTML5*, *CSS*, and *JavaScript* for designing the web pages and controlling the file system as shown in the Figure 3.2.

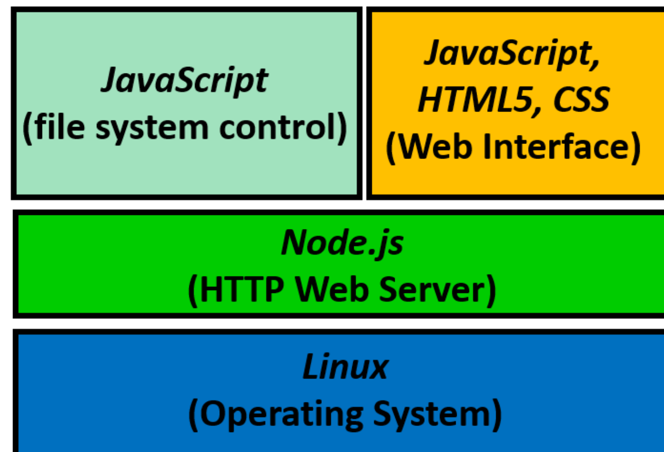

Figure 3.2: UPC web server platform.

3.2.1 HTTP Web Server

Node.js is adopted for implementing the *HTTP web server*. *Node.js* has a built-in module called *HTTP*, to allow transferring data over the *Hyper Text Transfer Protocol (HTTP)*. The *HTTP* module can create the *HTTP* server that listens to the server ports and gives the response back to the client. *Node.js* also offers the running environment of *JavaScript* programs on the server. Due to the cost and security concerns, the UPC web server runs on the private network in our implementation of the UPC system.

3.2.2 Web Interface

The UPC web interface is implemented using *HTML*, *CSS*, and *JavaScript*. *CSS* stands for *Cascading Style Sheets* and it is used to format the layout of a web page. In our implementation, the *internal CSS* is used to define a style for the web interface.

The two functions, *uploadFile()* and *showUploadedFiles()*, are implemented by *JavaScript* to listen the user's requests for uploading the jobs and showing the uploaded jobs and their processing status, as shown in the Figure 3.3.

The *HTTP POST* request method is used in the *uploadFile()* function for carrying the data to be written to the web server storage from an HTML form.

The *HTTP GET* request method is used in the *showUploadedFiles()* function for retrieving the data from the web server storage and showing them on the UPC web interface as the response.

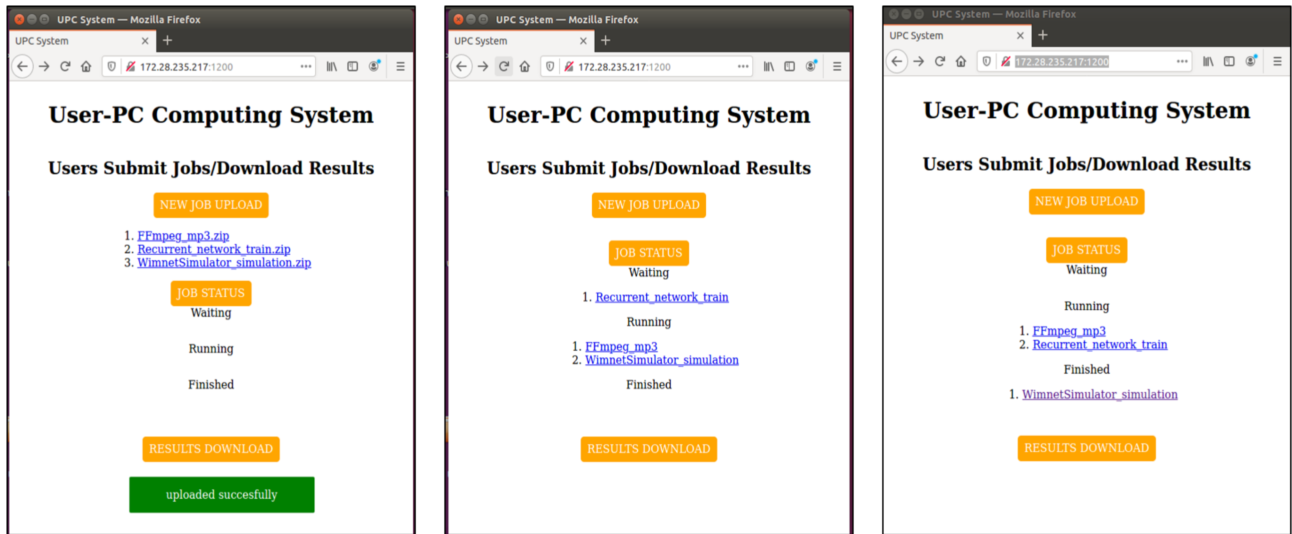

Figure 3.3: Uploaded jobs and their processing status on UPC web interface.

3.2.3 File System Control

Managing the files (*jobs/results*) is one of the most important task for the *Node.js* web server. *Node.js* provides the *file system module (fs)*, for reading, creating, updating, and deleting the files. As shown in Figure 3.4, the user can easily upload the jobs and download the results using the *Node.js file system* module.

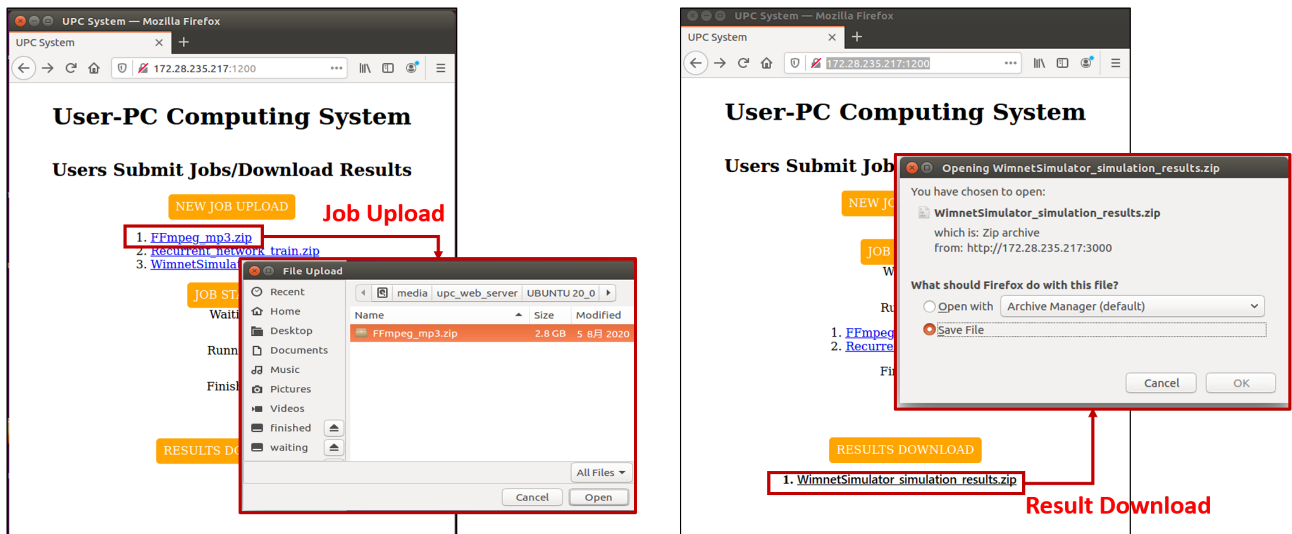

Figure 3.4: Upload and download files on UPC web interface using *fs* module.

The following three *JavaScript* functions are implemented in the web server for managing the files:

1. The *sendListOfUploadedFiles()* function performs sending the list of the files to be uploaded to the UPC web server by the user. The process is initiated when the *POST* request is obtained from the *uploadFile()* function. In this function, the *fs.readdir()* method is used to asynchronously read the contents of the user's directory. The callback of this method returns the list of all the file names in the directory.

2. The *sendUploadedFile()* function sends the files in the list obtained from the *sendListOfUploadedFiles()* function. In this function, the *fs.readFile()* method is used to read the data from the file. It will read all the data and save them into the buffer.

3. The *saveUploadedFile()* function saves the data from the buffer obtained from the *sendUploadedFile()* function. In this function, the *fs.createWriteStream()* method is used to make the writable stream for writing data in the file. This method is much better than *fs.writeFile()*, when it is needed to write very large amounts of data.

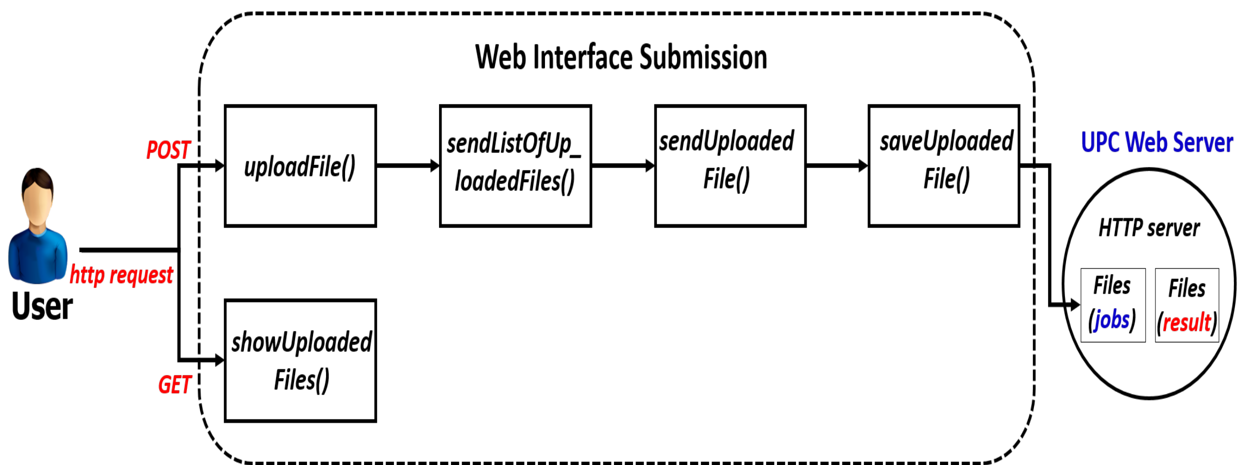

Figure 3.5: Use of *JavaScript* functions for web interface submission.

3.3 Summary

In this chapter, we presented the implementation of the UPC web interface. The users can easily submit the jobs and access to the results through the web interface. In the next chapter, we will present the implementation of *online job acceptance* function.

Chapter 4

Implementation of Online Job Acceptance Function

Previously, the UPC system only accepts jobs from local users manually through the UPC web interface. Besides, the UPC web server runs in private networks due to cost and security concerns. However, some application systems request to submit jobs online to the UPC system to enhance their processing capabilities. In this section, we present the implementation of the two online job acceptance functions. The implementation details will be discussed.

4.1 Online Job Acceptance Function

Some application systems will need to submit the jobs automatically to the UPC system for speeding up their job processing time. To realize it, two approaches using *Secure Shell File Transfer Protocol (SFTP)* and a *cloud storage* are considered and implemented for the online job acceptance functions in the UPC system.

For the first approach, we adopted *SSHFS* as the *filesystem client* [16]. The *UPC web server* interacts with the files (*jobs*) in the application system over the reliable data stream via the *SFTP* data communication protocol.

For the second approach, we adopted *pCloud* as a free cloud storage service [9]. The storage space can be accessed by calling the *pCloud* API for getting the jobs and sending the results to the *pCloud* directory of application systems by the web server.

4.1.1 Online Job Acceptance Function using SFTP

The first approach, online job acceptance function using *SFTP*, is implemented at the web server using *JavaScript*. *SSHFS* is actually adopted, because it can mount and interact with directories and files located on a remote server as the *filesystem client* using:

$ sudo sshfs app_system@public_ip_address:/ upc_web_server@private_ip_address:/

The web server can interact with the remote file system via *SFTP* by providing the file access, file transfer, and file management over any reliable data stream. Then, the application system only needs to prepare the directories for jobs and results at its own storage space and allow *port-22* for SSH with authorized access by the UPC web server. As shown in Figure 4.1, an application system, which has its own public server, can use this approach to send the jobs, and process them at the UPC system.

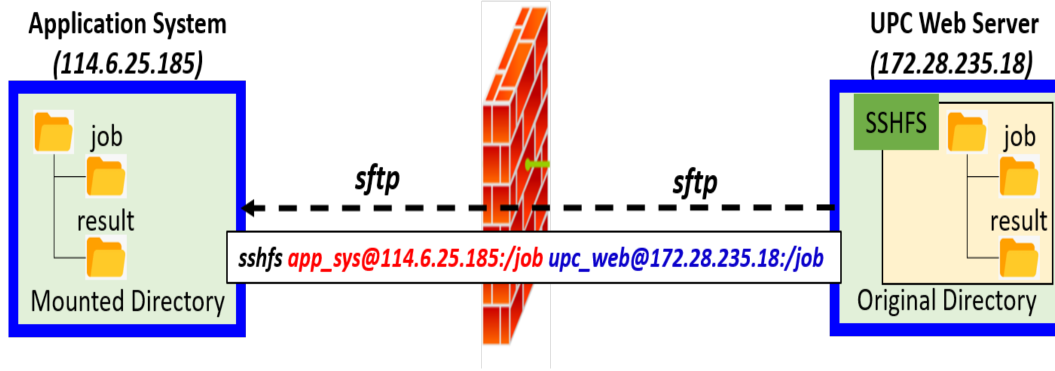

Figure 4.1: UPC online job acceptance using SFTP.

4.1.2 Online Job Acceptance Function Using Cloud Storage

The second approach, online job acceptance function using a *cloud storage*, is implemented at the web server using *JavaScript*. *pCloud* is adopted as the free secure cloud storage. For file access to the pCloud API, ***pcloudCLient.downloadFile(fileLink, file.name)*** and ***pcloudCLient.uploadFile (localResultsFile.name, localResultsFile.path, cloudFilePath)*** are used by the web server. The directories for adding jobs and accessing results are prepared in *pCloud*. As shown in Figure 4.2, an application system, which runs on the private server, can use this approach to send the jobs and process them at the UPC system.

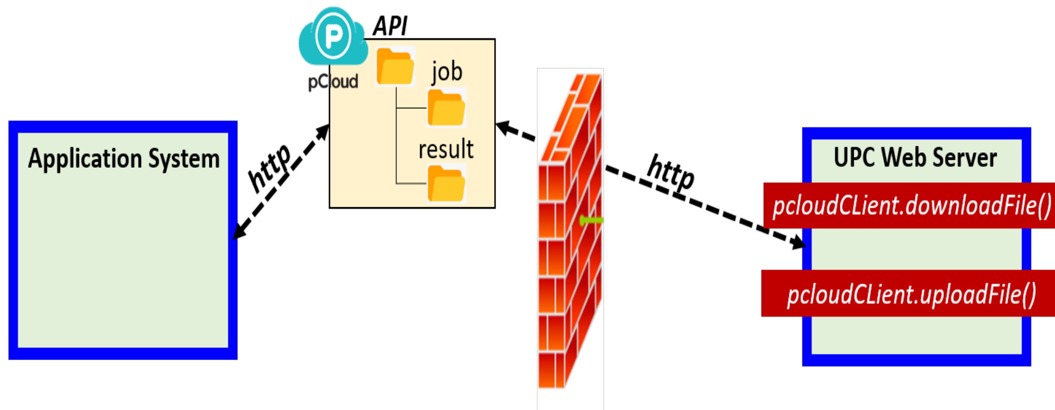

Figure 4.2: UPC online job acceptance using cloud storage.

4.2 Evaluation of Online Job Acceptance Function

For evaluations, we adopt *Android programming learning assistance system (APLAS)* [17] and *Exercise and performance learning assistant system (EPLAS)* [18] as the application systems, which require the high CPU capabilities and have been developed in our group.

4.2.1 Evaluation Setup

Table 4.1 shows the specifications of the master PC and the six worker PCs that are connected with the master through the 100Mbps wired in our experiments, and the specification of the servers for

APLAS and *EPLAS*.

For processing the *APLAS* jobs, we apply the multi-threading technique to increase the efficiency of worker PCs. Extensive measurements were conducted to find the best number of threads for each worker PCs. Table 4.1 also shows the different number of threads usage on each PCs for executing the *APLAS* jobs.

Table 4.1: PC specifications in experiments.

PC	CPU	# of core	cloc. freq.	mem.(GB)		disk (GB)		# of Thread
				avl.	total	avl.	total	
PC-1	corei3	4	1.7	2	4	64	500	1
PC-2	corei5	4	2.6	2	4	64	500	1
PC-3	corei5	4	2.6	2	4	64	500	1
PC-4	corei7	8	3.4	4	8	64	500	4
PC-5	corei9	16	3.6	8	16	64	500	5
PC-6	corei9	20	3.7	8	16	64	500	6
Master	corei5	4	3.2	8	8	225	225	
APLAS server	AMD Opteron 4core	4	2.6	11	11	100	100	
EPLAS server	corei5	4	2.2	8	8	1000	1000	

4.2.2 Results for APLAS Jobs with SFTP

Currently, *APLAS* [17] offers six assignments of Android applications to the students, called *BasicAppX1*, *BasicAppX2*, *ColorGame*, *SoccerMatch*, *AnimalTour*, and *MyLibrary*. Table 4.2 shows the number of jobs, the CPU time at using the *APLAS* server only, and the CPU time at using the UPC system together for each assignment. The total CPU time can be reduced by 90.5% by using the UPC system together through the online job acceptance function with *SFTP*.

Table 4.2: CPU time for *APLAS*.

Type of job	Assign APLAS jobs in UPC						#. of job	CPU time	
	PC-1	PC-2	PC-3	PC-4	PC-5	PC-6		APLAS	APLAS with UPC
BasicAppX1	2	4	4	6	10	24	50	06:02:19	00:24:39
BasicAppX2	2	3	3	8	15	19	50	04:04:29	00:26:54
ColorGame	1	3	3	8	15	20	50	04:19:06	00:27:06
SoccerMatch	2	3	3	11	20	24	50	04:32:29	00:28:11
AnimalTour	0	4	4	8	15	27	50	03:52:19	00:20:34
MyLibrary	1	4	4	11	20	24	50	03:16:48	00:20:44
Total							**335**	**26:07:30**	**02:28:08**

4.2.3 Results for EPLAS Jobs with Cloud Storage

Currently, *EPLAS* [18] offers contents to practice five *Yoga* poses, namely, *Mountain pose*, *Side-bend pose*, *Warrior pose*, *Seated 1 pose*, and *Seated 2 pose*.Table 4.3 shows the number of jobs, the CPU time at using the *EPLAS* server only, and the CPU time at using the UPC system together for each pose. The total CPU time can be reduced by 55.1% by using the UPC system together through the online job acceptance function with *pCloud*. It is noted that in the UPC system, the jobs cannot run on three workers, PC-1, PC-2, and PC-3 due to the memory shortage. By installing additional memories or new worker PCs with sufficient memories, it is expected to further reduce the CPU time by the UPC system.

Table 4.3: CPU time for *EPLAS*.

Type of job	**Assign EPLAS jobs in UPC**			**#. of job**	**CPU time**	
	PC-4	**PC-5**	**PC-6**		**EPLAS**	**EPLAS with UPC**
Mountain pose	12	14	15	41	00:20:37	00:09:17
Side-bend pose	12	14	15	41	00:11:22	00:09:06
Warrior pose	12	14	15	41	00:27:43	00:09:26
Seated 1 pose	12	14	15	41	00:24:47	00:09:13
Seated 2 pose	12	14	15	41	00:18:42	00:09:15
Total				**205**	**01:43:11**	**00:46:17**

4.3 Summary

In this chapter, we presented the implementations of the online job acceptance functions using the *SFTP* and a *cloud storage* in the UPC system. The experiment results using *APLAS* jobs and *EPLAS* jobs showed that the jobs were successfully accepted from the application systems, and the total CPU time of completing the jobs was reduced by 90.5% for *APLAS* jobs and 55.1% for *EPLAS* jobs respectively. In the next chapter, we will present the implementation of the *job migration* function.

Chapter 5

Implementation of Job Migration Function

In this chapter, we present the implementation of the *dynamic job migration function* in the UPC system. It uses two open-source software named *Checkpoint-Restore in Userspace (CRIU)* and *Podman*. *CRIU* can save all the data related to the running job into image files, which is called *Checkpointing*. *Podman* can manage the *Docker containers*. Besides, since these tools have been developed for *Linux OS*, *Windows Subsystem for Linux (WSL)* is added at the additional layer to cover for the PCs running on *Windows OS*. *WSL* enables the PC to run *Linux* distributions directly on *Windows10* alongside *Windows* applications, without the overhead of a traditional virtual machine.

5.1 Job Migration Function

In the UPC system, any job is processing inside the *Docker container*, which allows the dynamic migration of a job that is currently running on one worker, to another one. After the job is migrated, it can continue executing the remaining part of the job. By adopting this *job migration function*, the UPC system can accelerate the job completion by dynamically migrating it to a faster PC when it becomes idling. Besides, the system can avoid job completion failures due to the memory shortage, the full occupation of the CPU cores, or the hardware trouble.

5.1.1 Job Migration Process

The job migration is the process of moving all the resources associated with the running job on a worker to another one. The migration process actually consists of: (a) saving the states of the memory and CPU including the *registers* and the data processing elements (*pipeline*) in the current PC, and (b) initiating them in the new PC.

For this function, *Linux* allows the user to access the ***/proc*** file system which stores the information related to the running processes. Using *CRIU (Checkpoint and Restore in User space)* tool [19], the *process* state can be captured and initiated at the specified points.

5.1.1.1 State Capturing

To capture the state of the job on a PC, *CRIU* collects the following information of the process and dumps them into the image files, as shown in Figure 5.1:

1. Process's threads under */proc/$pid/task/* directory,

2. Children threads under */proc/$pid/task/$tid/children* directory,

3. Memory contents under */proc/$pid/smaps* and */proc/ $pid/map_files* directories,

4. File descriptors under */proc/$pid/fd* directory, and

5. Core(*pipes*) parameters for the pipeline from */proc/$pid/stat* file.

In the UPC system, any job is executing inside the *Docker container*, where *CRIU* cannot correctly create checkpoint and restore the jobs. To overcome this limitation, we adopt the *Podman* container management tool [20]. It can manage the entire container ecosystem and support multiple container image formats, including *Docker images*. *Podman* does not require a daemon running as root to launch and manage containers. It can run with the same permissions as the user who launched the containers. Therefore, it enables *CRIU* to create the checkpoint and restore the running containers (*jobs*).

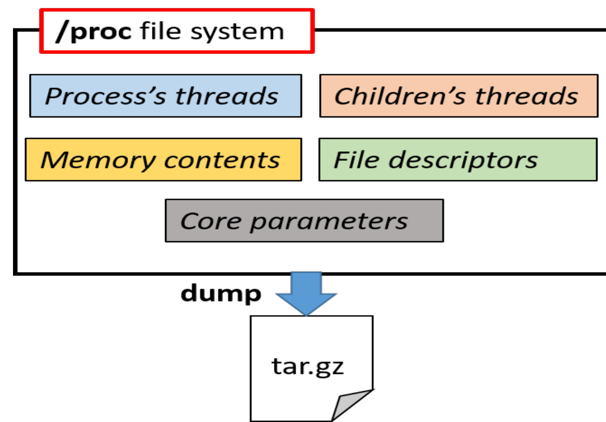

Figure 5.1: State capturing at PC.

5.1.1.2 State Restoring

To restore the captured state on a new PC, *CRIU* reads the image files, and restores the shared files and the memory areas by creating the corresponding processes at the PC. Then, it recreates the processes by repeatedly calling the *fork()* process creation function until the basic task resources such as the memory mappings for exact location, the timers, the credentials, and the threads are successfully restored.

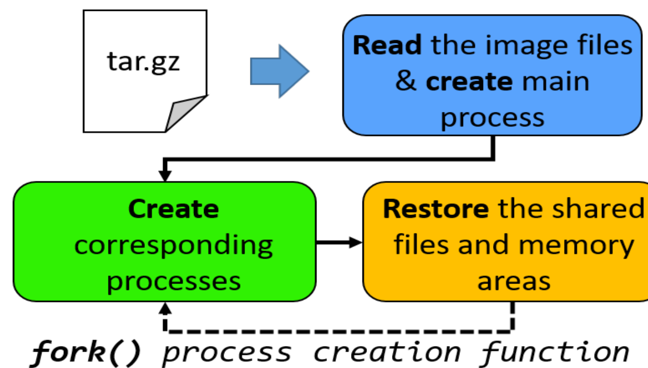

Figure 5.2: State restoring at new PC.

5.1.2 Software Tools

The *job migration function* at a UPC worker is implemented using *Podman 1.6.2* and *CRIU 3.14*, and *Python* codes to control them.

Podman 1.6.2 is used to manage the Docker container jobs during the job migration process at the UPC worker.

CRIU 3.14 is used to create the checkpoint and restore the jobs in the *Docker container* during the job migration process at the UPC worker. It restores the job programs into one or multiple image files.

5.1.3 Worker Search for Migration

The following steps are performed to search for an available worker to be migrated.

1. The jobs are requested by the relevant worker one after the other until the queue becomes empty.
2. The master changes the worker status into *'busy'* while job is processing at the worker.
3. The master changes the worker status into *'free'* when all jobs in the queue are finished.
4. The master finds the *'busy'* worker, where the job is running that has maximum CPU time reduction.
5. The master sends the *'checkpoint'* alert to the selected *'busy'* worker and *'restore'* alert to the *'free'* worker.

5.1.4 Procedure of Job Migration Function

The job migration function can migrate the currently running job on a worker to another one by the following steps. Any command is issued at UPC master through SSH. The migration function is called after source (*'busy'*) and destination (*'free'*) workers are selected for the job migration.

1. Suspend the running container in the current worker when *checkpoint* alert is received by:
   $ sudo podman container stop job:latest
2. Make the checkpoint image file to save the job state by:
   $ sudo podman container checkpoint -l –export=job.tar.gz
3. Remove the job from the job queue in the worker by:
   $ sudo os.remove(upc@upcworker1:/worker1/ queue/job)
4. Send the checkpoint image file to the UPC master by:
   $ sudo scp upc@upcworker1:/job.tar.gz upc@upcmaster :/checkpoints/worker1/
5. Send the checkpoint image file to the worker that receives the *restore* alert by:
   $ sudo scp upc@ upcmaster:/checkpoints/worker1/job.tar.gz upc@upcworker2:/
6. Restore the running state of the job from the checkpoint image file in the worker by:
   $ sudo podman container restore –import=job.tar.gz
7. Run the restored job in the worker by:
   $ sudo podman run -d -it –name jobrestore job:latest

5.1.5 WSL Layer in Windows PC

Windows Subsystem for Linux (WSL) allows users to run the applications on *Windows* without the overhead of a traditional virtual machine. By adding *WSL* layer on the *Windows* as shown in Figure 5.3, CRIU and Podman can be installed on the *Linux kernel* of the host OS (*Windows*). In the UPC system, the jobs are running as the *Docker container* at the workers using the full resources of the host OS. The running container can be managed from the *WSL Linux terminal* for carrying out the migration process.

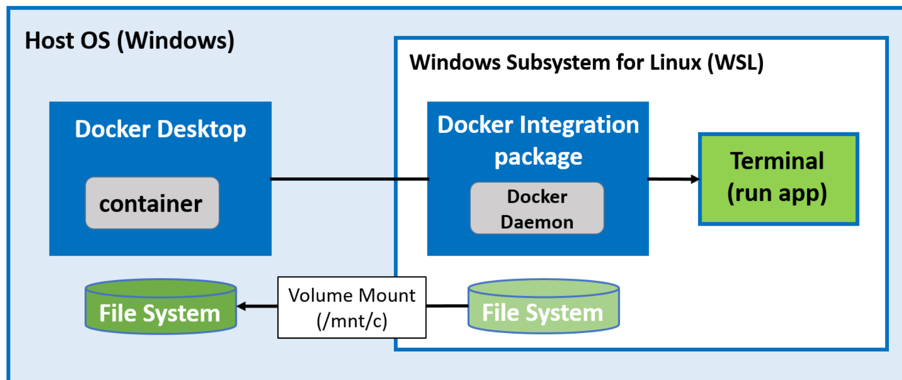

Figure 5.3: Run Docker container on Windows with WSL.

5.1.6 Job Migration Limitation

To migrate a job (process), it is necessary to save the several resources such as the CPU state, the memory state, the network state, and the disk state as the image files. The resource formats are different among different OS. As shown in Figure 5.4, a job can be successfully migrated (checkpoint/restore) from one PC to another which is running the same OS.

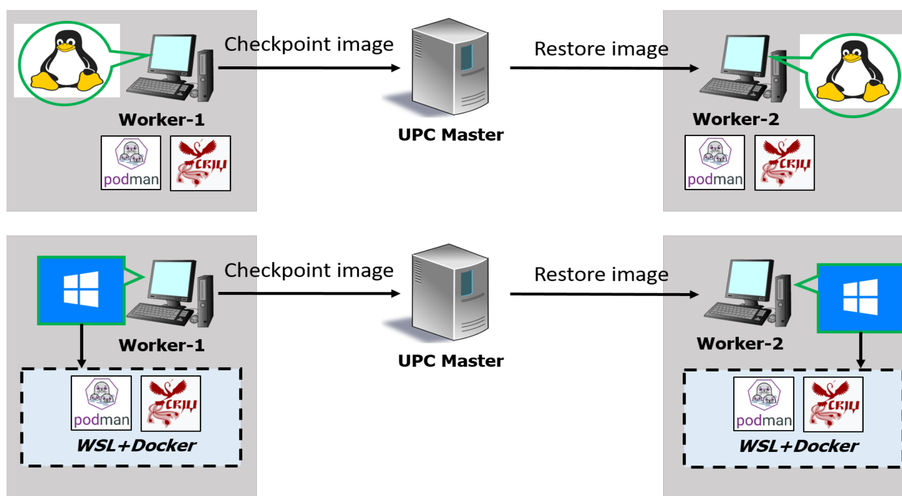

Figure 5.4: Job migration between same OS worker PCs.

5.2 Evaluation of job migration function

In this section, we evaluate the implemented *job migration function* in the UPC system through extensive experiments using the testbed system.

5.2.1 Evaluation Setup

Table 5.1 shows the specifications of the *UPC master* and the six *UPC workers*. The workers are connected with the master through the 100$Mbps$ wired in our experiments. These workers can be classified into two groups by the number of CPU cores. *PC-1*, *PC-2*, and *PC-3* are into the first group. *PC-4*, *PC-5*, and *PC-6* are into the second group. As the running jobs with the different number of threads usage in the UPC system, five C, two C++ and seven Python programs are selected as shown in Table 5.2.

Table 5.1: Worker specifications.

PC #	CPU	# of core	clock freq.	memory(GB)		disk(GB)	
				aval.	total	aval.	total
PC1	i3	4	1.7	2	4	64	500
PC2	i5	4	2.6	2	4	64	500
PC3	i5	4	2.6	2	4	64	500
PC4	i7	8	3.4	4	8	64	500
PC5	i9	16	3.6	8	16	64	500
PC6	i9	20	3.7	8	16	64	500
Master	i5	4	3.2	8	8	225	225

Table 5.3 shows the measured CPU time when each job runs on each worker. The two C programs, *Network Simulator*, *Optimization Algorithm*, are single-thread programs for wireless networks [21]. They have been developed in our group. The CPU time is not much different among the workers except *PC-1*, because it does not have the maximum turbo frequency feature. The remaining C programs [22], *FFmpeg*, *multimedia content resizing*, and *multimedia format changing*, use multi-thread. The CPU time is much different among the workers.

The two C++ physics simulation programs, *Palabos* [23] and *Flow* [24], use two or four threads. The CPU time is not much different among the workers. The next four neural-network [25] related jobs, Deep Convolutional Generative Adversarial Networks (*DCGAN*), Recurrent Neural Network (*RNN*), Convolutional Neural Network (*CNN*), and *Covid-19 detection* are Python programs using multi-thread. The CPU time is much different between the two worker groups, because the number of cores in the CPU is different among them. The remaining three Python programs [26], *Converter*, *Blockchain mining*, *Covid-19 outbreak prediction*, use one or four threads. The CPU time is not much different between the two groups.

5.2.2 Validity of Migration Function

First, we verify the implemented function by migrating the 14 jobs between the six worker PCs at three different job completion rates of 25%, 50%, and 75% for the migration checkpoint. We checked the validity of the migration function by the scenarios in Figure 5.5. A total of 1260 combinations are executed.

Table 5.2: Job specifications.

Job #	job name	# of threads	disk usage (GB)
job1	Network Simulator	1	0.392
job2	Optimization Algorithm	1	1.5
job3	DCGAN	17	1.9
job4	RNN	17	1.9
job5	CNN	17	1.9
job6	FFmpeg	18	2.8
job7	Converter	1	1.1
job8	Palabos	2	6.7
job9	Flow	4	0.438
job10	Blockchain mining	1	0.92
job11	Covid-19 detection	23	2.95
job12	Covid-19 outbreak prediction	4	1.84
job13	multimedia content resizing	18	2.8
job14	multimedia format changing	18	2.8

Table 5.3: Jobs standard processing time.

job #	PC1	PC2	PC3	PC4	PC5	PC6
job1	2:14:46	1:06:44	1:06:41	0:54:21	0:39:40	0:36:09
job2	0:41:43	0:26:38	0:26:35	0:20:27	0:13:53	0:13:10
job3	1:37:14	1:09:28	1:09:29	0:22:44	0:12:45	0:11:15
job4	0:17:43	0:12:01	0:12:04	0:07:03	0:05:37	0:04:49
job5	0:26:04	0:22:22	0:22:17	0:07:07	0:05:24	0:04:47
job6	0:46:37	0:31:49	0:31:51	0:13:23	0:07:59	0:06:54
job7	0:17:09	0:11:34	0:11:30	0:05:41	0:04:53	0:04:15
job8	0:12:51	0:08:38	0:08:37	0:05:24	0:04:29	0:03:19
job9	0:25:20	0:14:45	0:14:44	0:11:08	0:08:32	0:07:55
job10	0:36:28	0:09:09	0:09:07	0:07:53	0:05:59	0:04:14
job11	0:39:35	0:24:53	0:24:57	0:10:42	0:04:08	0:03:16
job12	0:13:12	0:06:35	0:06:38	0:05:09	0:03:55	0:03:01
job13	0:34:50	0:19:47	0:19:45	0:12:33	0:07:48	0:07:10
job14	0:36:33	0:24:56	0:24:53	0:10:55	0:05:45	0:04:19

Figures 5.6, 5.7, 5.8, 5.9, and 5.10 illustrate the migration of *job3* (*DCGAN*) from one PC to the others at three different job completion rates of 25%, 50%, and 75% for the migration checkpoint and show the total job execution time. The results show that this job was successfully migrated between different workers, and the migration from the slow worker to the faster one reduced the total CPU time. It is also observed that the faster migrating the slower worker's job, the shorter the total CPU time. It is an important factor to be considered for designing the scheduling algorithm combining with the job migration. In some cases, the total CPU time is longer due to migration of the job from the fast worker to the slower one. As one thing, PC2 and PC3 have the same specifications, therefore, it is only showed the PC2's validation results of migration function. The

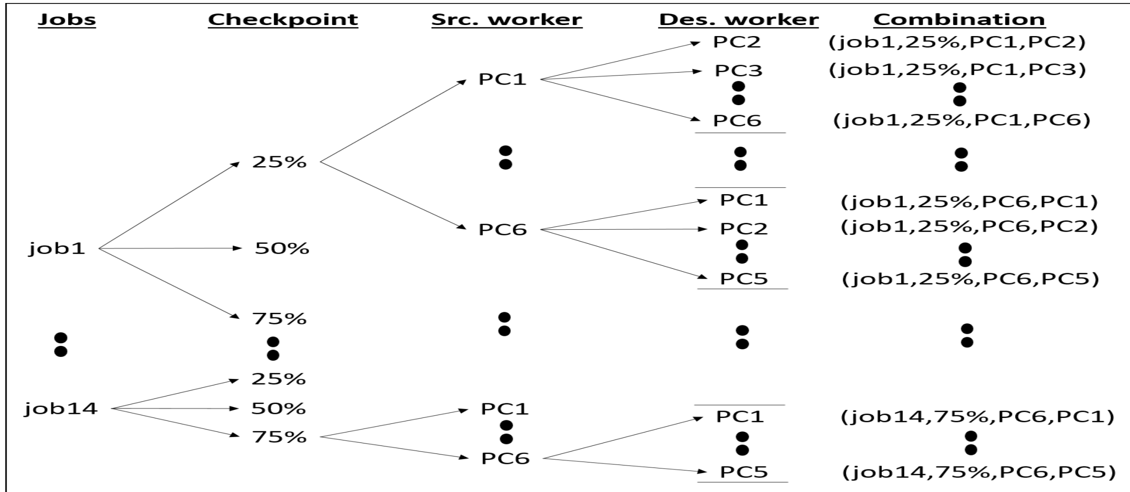

Figure 5.5: Scenarios for validating migration function.

CPU time of job migration from the PC3 is considered as same as the PC2.

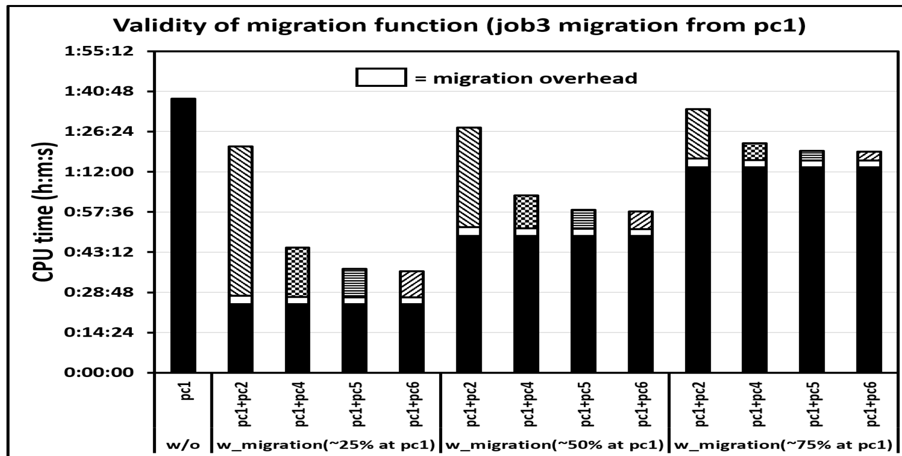

Figure 5.6: CPU time of job migration from PC1.

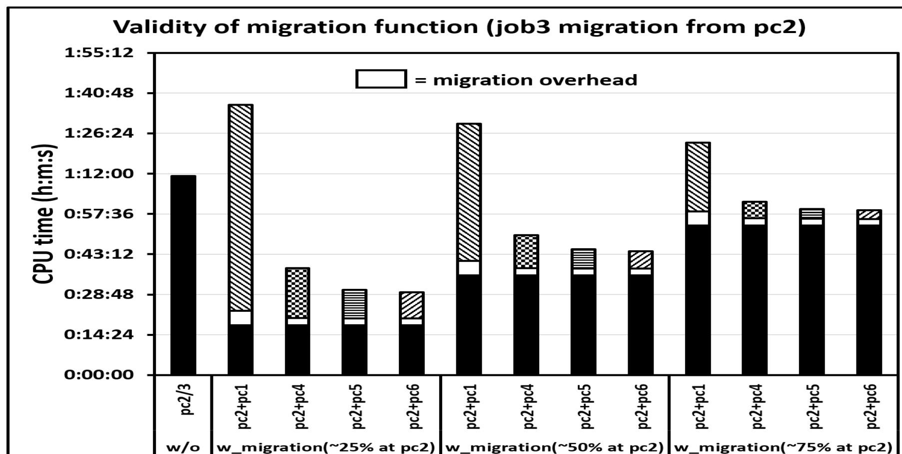

Figure 5.7: CPU time of job migration from PC2/PC3.

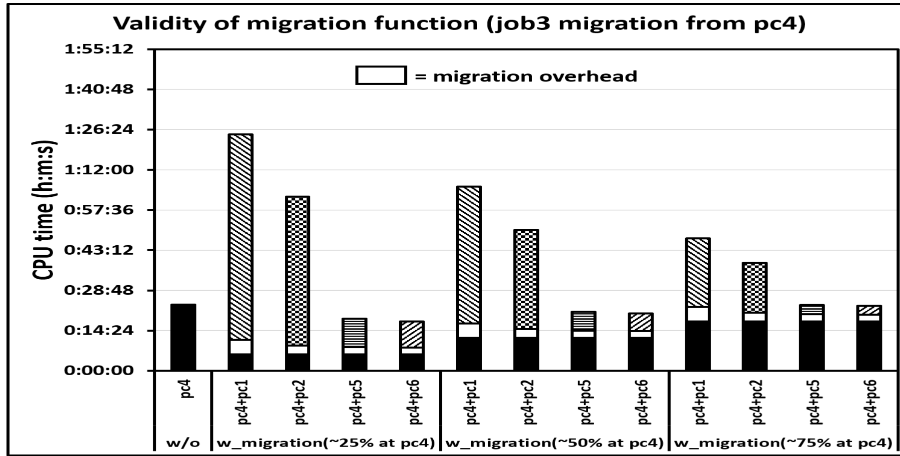

Figure 5.8: CPU time of job migration from PC4.

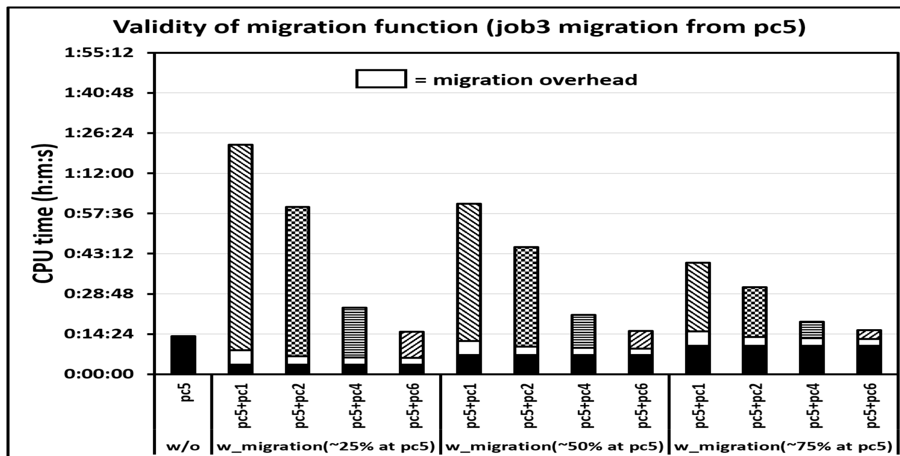

Figure 5.9: CPU time of job migration from PC5.

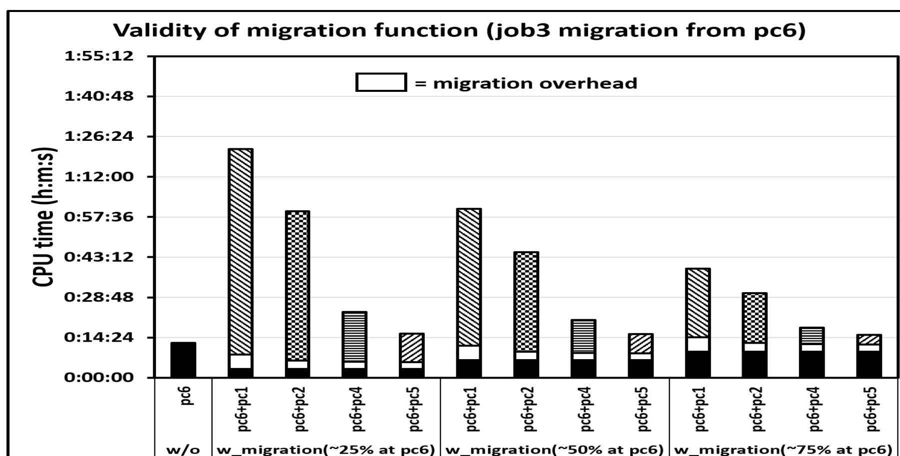

Figure 5.10: CPU time of job migration from PC6.

To demonstrate the effectiveness of the implemented function, Table 5.4 shows the improved percentage obtained by Eq.4 based on the data in Figures 5.6, 5.7, 5.8, 5.9, and 5.10. It is noted that the longer running at the slow worker will not be greatly improved to the total CPU time

Table 5.4: Total CPU time improvement rates by migration.

migrate from slow to faster workers	Improvement by migration of job-3 on three different migration checkpoints (*ImprovedPercentage*)		
	25% migration	50% migration	75% migration
PC-1 + (PC-2/3/4/5/6)	49.17%	31.89%	14.60%
PC-2/3 + (PC-4/5/6)	53.97%	34.81%	15.64%
PC4 + (PC-5/6)	23.08%	11.98%	0.82%
PC5 + PC6	* no improvement ($T_{mig_overhead} > T_{i+1,after}$, and $PC5 \approx PC6$)		

by applying the job migration function. It can be clearly seen by comparing the total CPU time improvement rates upon the three different migration checkpoints in Table 5.4.

$$N_i = COUNT(PC_{i+1} : PC_n) \quad (1)$$

$$T_{i,avg} = \frac{1}{N_i} \sum_{j=i+1}^{n} (T_{i,before} + T_{mig_overhead} + T_{i+1,after}), (j < n) \quad (2)$$

$$T_{improve} = T_{w/o} - T_{i,avg} \quad (3)$$

$$ImprovedPercentage = \frac{T_{improve}}{T_{w/o}} * 100\% \quad (4)$$

where,

- n represents the total number of PCs and it is arranged in ascending order (PC_1, PC_2,...PC_n,),
- N_i represents the total number of faster PCs than PC_i,
- $T_{i,avg}$ represents the average total CPU time with migration (from PC_i to the faster PCs),
- $T_{w/o}$ represents the total CPU time at PC_i without migration,
- $T_{i,before}$ represents the total CPU time at PC_i (before migration),
- $T_{mig_overhead}$ represents the migration overhead from PC_i to PC_{i+1},
- $T_{i+1,after}$ represents the total CPU time at PC_{i+1} (after migration),
- $T_{improve}$ represents the total CPU time improvement by migration.

Next, we proved the effectiveness of the migration function by conducting the extensive experiments. We randomly select six jobs among 14 jobs and randomly assign to the six workers. Table 5.5 shows the total CPU time and *makespan* comparison of with and without migration for 10 random cases. For migration, when a faster worker becomes idle, the job that has the maximum CPU time reduction will be migrated to that idle worker. The maximum CPU time reduction is considered by deducting the remaining CPU time on current worker to the addition of migration overhead and CPU time on new worker. Table 5.6 shows the total CPU time and *makespan* can be reduced 34.62%, and 49.72%, respectively, by migration.

Table 5.5: Comparison of with and without migration for 10 random cases.

Random cases	without migration (w/o)		with migration (w)		
	total CPU time	makespan	total CPU time	makespan	#. of migration time
1	1:51:53	0:39:57	1:06:20	0:17:04	3
2	1:06:20	0:17:04	0:55:25	0:14:32	4
3	1:38:51	0:25:32	1:14:18	0:18:39	3
4	1:45:31	0:34:55	1:06:27	0:16:16	2
5	1:53:07	0:36:34	1:05:34	0:14:01	3
6	1:23:15	0:22:44	1:04:08	0:16:48	2
7	1:39:52	0:41:48	1:03:29	0:17:05	3
8	1:29:59	0:34:56	0:46:07	0:12:32	2
9	1:27:21	0:24:58	1:02:32	0:15:10	2
10	1:09:22	0:24:49	0:40:53	0:10:17	3
Average	**1:32:33**	**0:30:20**	**1:00:31**	**0:15:14**	**3**

Table 5.6: Analysis of CPU time and makespan results.

	w/o migration	w migration	Reduction time	Reduction rate
Average total CPU time	1:32:33	1:00:31	0:32:02	34.62%
Average makespan	0:30:20	0:15:14	0:15:05	49.72%

5.2.3 Job Scheduling Algorithms with and without migration

Next, we evaluate the CPU time reduction by the migration. Three scheduling algorithms called *FCFS_L2H*, *FCFS_H2L*, and *Hjob_L2H*, are implemented and applied. The first two algorithms follow the *First Come First Serve* approach. *FCFS_L2H* assigns the first arriving job to the slowest available worker PC. *FCFS_H2L* assigns the first arriving job starting from the highest available worker PC. *Hjob_L2H* assigns the longest CPU time job starting from the slowest worker PC. Figures 5.11, 5.12, and 5.13 show the total CPU time comparison of each PC with and without applying the job migration upon three scheduling algorithms respectively.

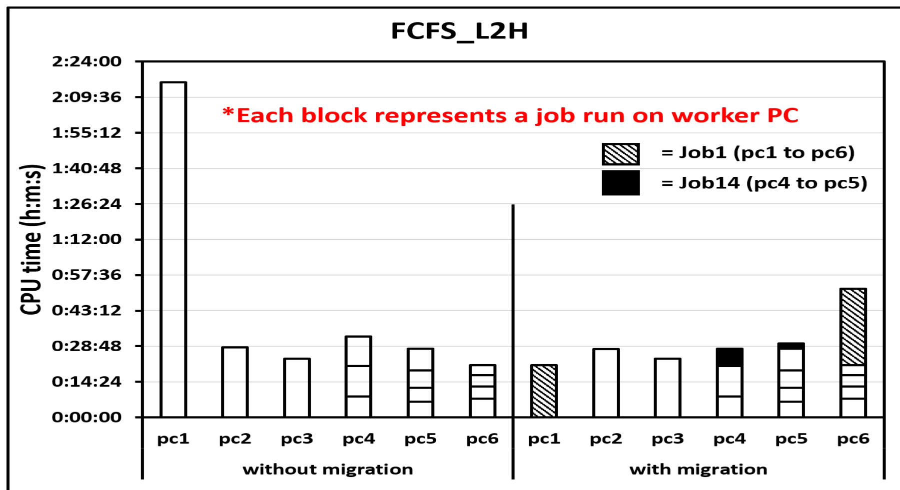

Figure 5.11: FCFS_L2H algorithm with and without migration.

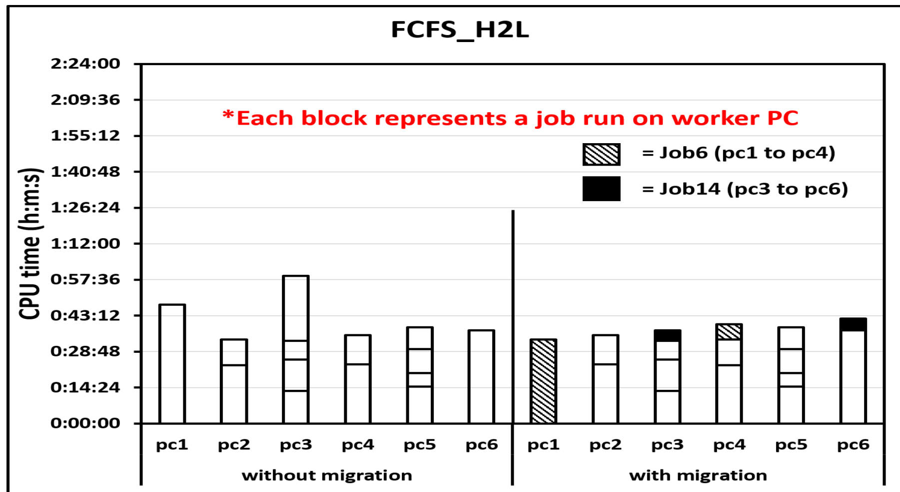

Figure 5.12: FCFS_H2L algorithm with and without migration.

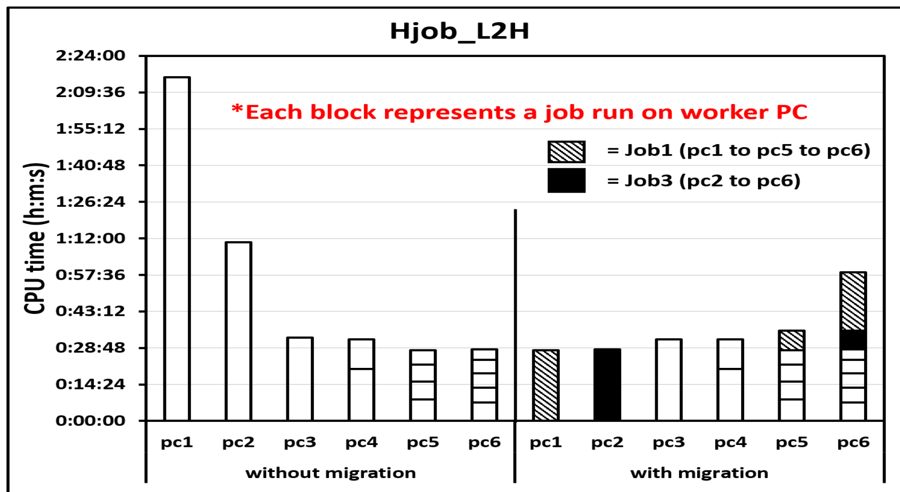

Figure 5.13: Hjob_L2H algorithm with and without migration.

5.2.3.1 Analysis of Migrated Jobs

Table 5.7 shows the migrated jobs in each algorithm. In *FCFS_L2H* algorithm, the two jobs, *job1* and *job14*, are migrated from PC1 to PC6 and from PC4 to PC5, respectively. The CPU time is greatly improved. In *FCFS_H2L* algorithms, *job6* is migrated from PC1 to PC4 and *job14* is from PC2 to PC5. In *Hjob_L2H* algorithm, *job1* is migrated into two times from PC1 to PC5 and PC5 to PC6. *Job3* is migrated from PC2 to PC6 and the total CPU time is reduced significantly. The workers are efficiently used by migrating running jobs.

5.2.3.2 Makespan Results

Table 5.8 compares the makespan of three scheduling algorithms with and without using migration. The makespan was reduced by 53.73% on average. "with migration" outperforms "without migration" in terms of *makespan*.

Table 5.7: Migrated job conditions.

	migrated job	from			to		
		Source worker	**partially finished percentage**	**CPU time**	**Destination worker**	**Remaining executed percentage**	**CPU time**
FCFS_L2H	job1	PC1	14.69%	00:21:14	PC6	85.31%	00:30:50
	job14	PC4	63.18%	00:06:57	PC5	36.82%	00:02:07
FCFS_H2L	job6	PC1	54.45%	00:33:35	PC4	45.55%	00:06:06
	job14	PC3	16.71%	00:04:10	PC6	83.29%	00:03:35
Hjob_L2H	job1	PC1	18.25%	00:27:49	PC5	18.78%	00:07:51
	job3	PC2	33.85%	00:28:14	PC6	66.15%	00:07:26
	job1	PC5	18.78%	00:07:51	PC6	62.57%	00:22:59

Table 5.9 compares the total CPU time for 14 jobs by three scheduling algorithms with and without the migration. The total CPU time was reduced by 26.45% on average. The total CPU time can be reduced significantly, especially, in *Hjob_L2H* with the migration.

Table 5.8: Makespan results (H:M:S).

	without migration	with migration	Reduction time	Reduction rate
FCFS_L2H	02:15:35	00:52:04	01:23:31	61.59%
FCFS_H2L	00:59:07	00:42:07	00:17:00	28.76%
Hjob_L2H	02:15:35	00:58:39	01:16:56	56.75%
average	**01:50:06**	**00:50:57**	**00:59:09**	**53.73%**

Table 5.9: Total CPU time results (H:M:S).

	without migration	with migration	Reduction time	Reduction rate
FCFS_L2H	04:29:30	03:02:27	01:27:03	32.31%
FCFS_H2L	04:11:42	03:46:47	00:24:55	9.89%
Hjob_L2H	05:27:09	03:34:47	01:52:22	34.35%
average	**04:42:47**	**03:28:00**	**01:14:47**	**26.45%**

5.3 Summary

In this chapter, we presented the implementation of the *dynamic job migration function* using *CRIU* and *Podman*. This function can speed up the job completion and is useful to overcome the memory overuse, or avoid hardware trouble. The experimental results confirmed the validity of the implemented function, and the effectiveness in improving the resource utilizations of workers and reducing the total CPU time and *makespan*. In the next chapter, we will present the implementation of the running job backup function.

Chapter 6

Implementation of Job Running Backup Function

In this section, we present the implementation of the *job running backup function* in the UPC system, by extending the *dynamic job migration function*. It periodically check-points the running job in a worker, and automatically migrates it to another healthy worker when the current worker meets a trouble.

6.1 Job Running Backup Function

In the UPC system, some jobs, such as *physics simulations* and *neural networks*, require the long CPU time to be completed. Then, the probability of causing a failure of the running worker will be increased due to the memory shortage, the full occupations of the CPU cores, the power outage, or some hardware troubles. They may cause restarting or shutdown of the worker. This runtime failure can delay the completion of the job.

To avoid it as much as possible, the current state of running the job on the worker should be automatically backed up, and the job should run from the backed up state on another healthy worker. Therefore, the job running backup function is implemented by extending the *dynamic job migration function*. *Checkpoint-Restore in Userspace (CRIU)* is periodically applied to capture the job running state of the running job at a worker.

6.1.1 Job Check-Pointing Process

The *job check-pointing process* is implemented to save the resources that are associated with the currently running job on the worker. As shown in Figure 6.1, first, it captures the states of the memory and the CPU, including the *registers* and the data processing elements (*pipeline*) in the current worker, which will be dumped and compressed into the *tar* file. Next, it sends the backup compressed file to the master that stores the latest checkpoint image to recover from job failure.

Linux allows the user to access the ***/proc*** file system that stores the information related with the running processes. Using the *CRIU* tool [19], the *process* state can be captured and initiated at the specified points.

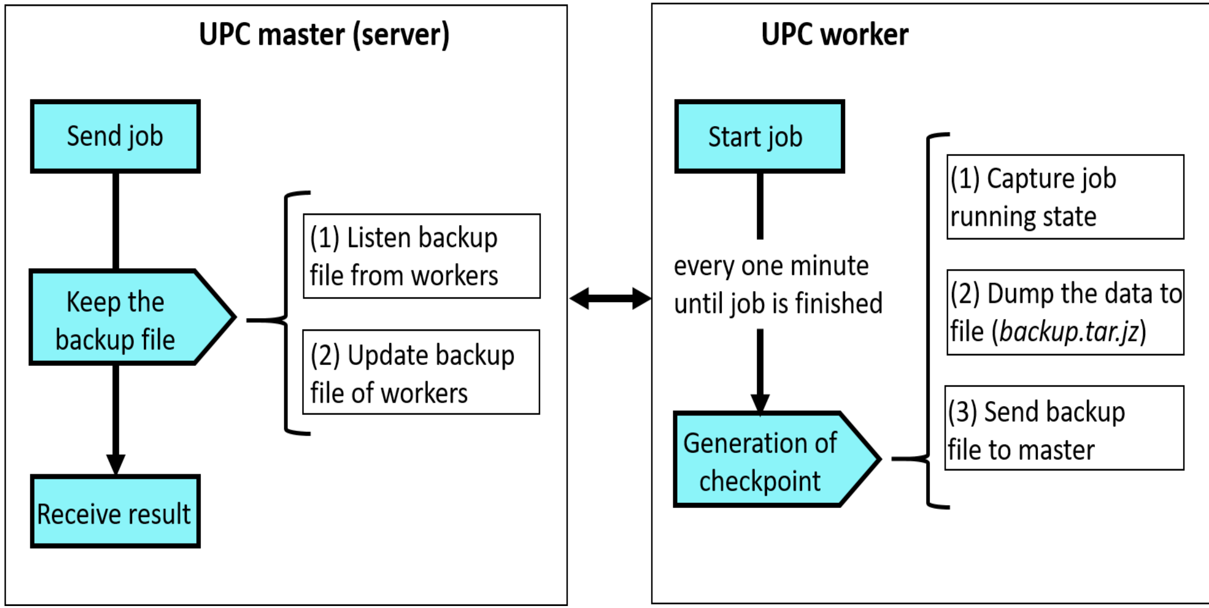

Figure 6.1: Job check-pointing at worker and backup saving at master.

6.1.2 Job Restoring Process

When the master detects the failure of the worker as shown in Figure 6.2, it will find the healthy worker among the available workers that has the higher performance than the failed one, and send the original *job* and the *backup* file to this selected worker. To restore the captured states at a new worker, *CRIU* reads the image files, and restores the shared files and the memory areas by creating the corresponding processes at the PC. It recreates the processes by repeatedly calling the *fork()* process creation function until the basic task resources such as the memory mappings for the exact locations, the timers, the credentials, and the threads are successfully restored.

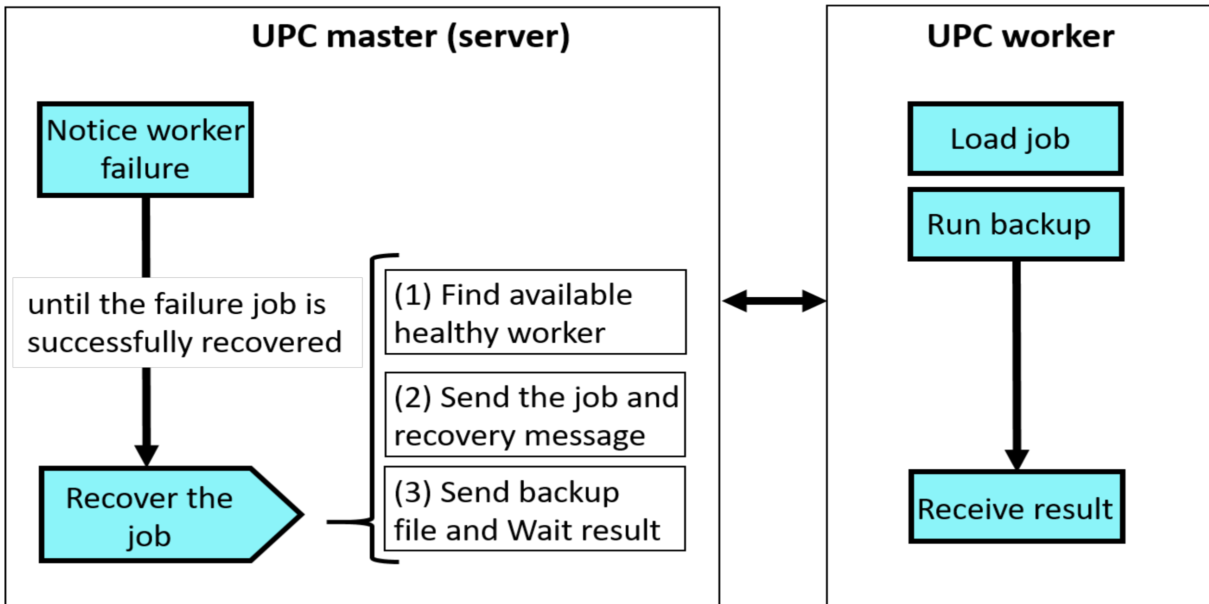

Figure 6.2: Noticing worker failure and job restoration at healthy worker.

6.1.3 Software Tools

Job running backup function is the extension of the *dynamic job migration function*, and so, *Podman 1.6.2* and *CRIU 3.14*, are re-applied for the implementation of backup function. In the *dynamic job migration function*, *CRIU* is used to generate the checkpoint, when the faster worker is free, and available to accept the job from the slow worker. However, in the *job running backup function*, *CRIU* is used to periodically generate the checkpoint of running job, and it is saved as the backup file at the master.

Podman 1.6.2 manages the Docker container jobs during the check-pointing process at the UPC worker.

CRIU 3.14 creates the checkpoint and restores the jobs in the *Docker container*. It saves the running job's state into one or multiple image files and restores the job from the saved state.

These tools are controlled by the *Python* programs. They correctly perform the backup process when the current worker meets a trouble during executing the job.

6.1.4 Worker Search for Checkpoint Restoration

The following steps are performed to search a new worker to restore the interrupted job.

1. The worker receives the jobs that are existing in the master's job queue, one after another.

2. The master changes the worker status to *'busy'* when the job is processed at the worker.

3. The worker sends the checkpoint file of the current job every one minute.

4. The master keeps the latest checkpoint file of the job from each worker.

5. The master changes the worker status to *'corrupt'* when the connection is lost.

6. The master changes the worker status to *'free'* when all the assigned jobs are completed.

7. The master sends the *'restore'* alert and the checkpoint file of the *'corrupt'* worker to the *'free'* worker.

6.1.5 Procedure of Job Running Backup Function

The job running backup function saves all the data related to the running job and restores the crashed job. The function is called when the jobs are processing at the workers.

1. Generate the checkpoint of the running job every one minute by:
   $ sudo podman container checkpoint -l –ignore-rootfs -R –export=./job.tar.gz

2. Send the checkpoint image file to the UPC master by:
   $ sudo scp upc@upcworker1:./job.tar.gz upc@upcmaster :/checkpoints/worker1/

3. Update the latest checkpoint image file of each worker at the master by:
   $ sudo rm -r upc@upcmaster :/checkpoints/worker1/<old>job.tar.gz

4. Send the original image of interrupted job to the worker that receives the *'restore'* alert by:
   $ sudo scp upc@upcmaster:/worker1/ original-job.xtar upc@upcworker2:/

5. Send the checkpoint file of interrupted job to the worker that receives the *'restore'* alert by:
   $ sudo scp upc@upcmaster:/checkpoints/worker1/ job.tar.gz upc@upcworker2:/

6. Load the original image of the interrupted job at new worker by:
   $ sudo podman load -i original-job.xtar

7. Restore the interrupted worker job from the checkpoint image file in the *free* worker by:
   $ sudo podman container restore –import=job.tar.gz

6.2 Evaluation of job running backup function

As the setup, we used the same experimental setup that was applied in the dynamic job migration function. To verify the effectiveness of the implemented function, we conducted extensive experiments with 14 jobs, in Table 5.2, on the testbed UPC system with six workers that have various specifications, as shown in Table 5.1. In the experiments, first, we verified the validity of the implemented job running backup function. Then, we confirmed the effectiveness of the function of reducing the CPU time loss when the worker is shutdown unexpectedly while running a job.

6.2.1 Correctness of Checkpoint Execution

First, we verify the correctness of the checkpoint execution in the *job running backup function* by applying it to 14 jobs on the six worker PCs. The checkpoint of the running job was generated every one minute, considering the required time and file size. The latest checkpoint was kept in the master. Table 6.1 shows the number of generated checkpoints and their average CPU time and file size. Any checkpoint was successfully generated and saved in the files. The average time for one checkpoint is 2.62*sec* and the average file size is 42.7*MB*, which does not increase the load of the UPC system.

6.2.2 Correctness of Job Running Backup

Next, we verify the correctness of the job running backup function implementation, when an unexpected shutdown will occur at the worker. *job-1* (*Network simulator*) and *job-3* (*DCGAN neural network*) were selected in this experiment. They ran at *PC-1* and *PC-2* respectively, while the states were check-pointed every one minute. After running the jobs for sufficiently long time, we manually shut-downed the workers, and observed that they were automatically migrated to *PC-5* and *PC-6*, and resumed to run there from the last check-pointed states.

Table 6.2 shows the CPU time to complete the two jobs. For *job-1*, the original worker *PC-1* was shut-downed after *49min 5sec* passed since it started running. For *job-3*, the first worker *PC-2* was shut-downed after *32min 42sec* passed since it started running. With the job running backup function, the job could continue running at the new worker from the check-pointed state when the original worker was shut-downed. The total CPU time could be reduced by 16.61% and 15.17%, respectively, from the cases of rerunning the whole job at the new workers from the beginning.

Table 6.1: Number of checkpoints and average loads.

job #	PC1	PC2	PC3	PC4	PC5	PC6	checkpoint loads	
							time (sec)	size (MB)
job1	132	64	64	52	37	34	1.6	0.83
job2	39	24	24	18	11	9	1.4	0.074
job3	95	66	66	20	10	9	2.0	38.7
job4	15	10	10	6	4	3	1.7	34.4
job5	24	20	20	7	5	3	1.5	27.6
job6	44	29	29	11	6	5	2.1	39.9
job7	15	9	9	5	4	3	4.1	6.0
job8	10	8	8	5	4	3	2.8	2.1
job9	24	13	13	9	7	5	3.6	155.6
job10	35	9	9	7	5	4	2.2	44.7
job11	37	24	24	10	4	3	4.3	191.3
job12	13	6	6	5	3	3	1.3	9.4
job13	32	18	18	12	7	5	4.5	36.4
job14	36	22	22	10	5	4	3.6	10.8
average	39	23	23	13	8	7	2.62	42.7

Table 6.2: CPU time for two jobs (H:M:S).

	job1		job3	
	Worker	CPU time	Worker	CPU time
before shutdown	PC-1	00:49:05	PC-2	00:32:42
after shutdown by backup	PC-5	00:24:56	PC-6	00:04:35
after shutdown by rerunning	PC-5	00:39:40	PC-6	00:11:15
Improvement	00:14:44 (16.61%)		00:06:40 (15.17%)	

6.3 Summary

In this chapter, we presented the implementation of the *job running backup function* in the UPC system. *Checkpoint-Restore in Userspace (CRIU)* is periodically applied to capture the job running state at a worker. When the master detects the failure of the worker, it automatically migrates the job running at another worker. The results confirmed that the proposal successfully resumed the job running from the interrupted point at another worker after the original worker was shutdown suddenly, and could reduce the total CPU time by migrating the job to a faster worker.

Chapter 7

Related Works in Literature

In this section, we introduce related works to this thesis. Several works have discussed the effectiveness of containerization, using *Docker container technology*, rather than virtualization, for processing resource intensive applications in computational environments. Moreover, a significant amount of research works has addressed checkpoint creations for migrations of *Virtual Machines (VMs)* and the *container* workloads to recover from node failures and balance the resource utilizations in parallel and distributed environments, *High Performance Computing (HPC)* environments, and *cloud computing* environments.

In [27], Benjamin et al. presented comparisons of the behaviors of four virtualization tools in grid computing environments. The authors measured the CPU, memory, disk, and network usages by executing the micro benchmark programs in each VM tool, and evaluated the linearity, overhead, and performance isolation. This work helps the user to select the suitable tool according to the application's nature.

In [28], Xavier et al. presented performance evaluations between the containerization and the virtualization for HPC applications. The authors found that the containerization is the lightweight alternative to the virtualization for HPC applications.

In [29], Park et al. presented a container-based cluster management platform to provide dynamic distributed computing environments desired by users. The authors compare the performance between the Docker-based execution and the native one by using two benchmark tools implemented with C, Java, Python, and R, and showed that Docker offers almost the native performance.

In [30], Jaikar et al. focused on executions of scientific jobs that require intensive resources in cloud computing environments. They proved that the Docker container outperforms the OpenStack virtual machine to execute the CPU and memory intensive jobs. Besides, the Docker container consumes the less power while executing scientific jobs.

In [31], Sindi et al. presented migrations of HPC workloads from the faulty node to the spare node using *CRIU* for *OpenVZ* containers. The authors showed that HPC applications written with C++, C, and FORTRAN were successfully migrated without modifying the application programs.

In [32], Zeynep et al. focused on data security during live migrations of containers. The authors proposed a secure model and proved the efficiency of the migration system both for stateless and stateful sample applications upon *LXC (Linux Containers)* using *CRIU* and complementary tools.

In [33], Ansel et al. demonstrated the check pointing and restarting of over 20 well known applications using *DMTCP (Distributed Multithreaded Check pointing)*. The authors showed that the applications were successfully checked and restarted on both centralized and distributed computation nodes.

In [34], Alrajeh et al. discussed the methodology of the responsive migration model in the High-Throughput Computing (HTC) system, and introduced six migration policies to determine the selection of the target computer when a migration is needed. It is demonstrated that the proposal could reduce the number of job evictions approximately by 92% when it is used as a fault-tolerance mechanism.

In [35], Yu et al. presented a logging and replay approach for live migrations of *Docker containers*. The authors observed that the container based technique is more portable, efficient, and easier to management. Both the down time and total migration time can be greatly reduced for live migrations of *Docker containers* compared with the approach for traditional virtual machines.

In [36], Fan et al. proposed locality live migrations of *Docker containers*. The simulation results showed that the proposed method improvesd the utilization of resources of servers, and balanced all kinds of resources on the physical machine.

In [37], Stoyanov et al. reported the causes of significant delays in live migration due to the large amount of memory use by the applications inside the container, especially when checkpointing the process using *CRIU*. This issue is addressed using the *CRIU* feature, the so-called *"image cache/proxy"*, which keeps all the memory pages in a cache buffer rather than storing them on disk.

In [38], Junior et al. investigated container migrations in large-scale geo-distributed platforms. Long-distance migrations have significant impacts on the migration downtime. The authors proposed a model that will transfer *Docker container* volumes' contents that are not being actively modified prior to the actual container migration. Thus, only a small number of *"hot"* files must be transferred during the downtime.

In [39], Fukai et al. introduced *BLMVisor* as an OS-independent and lightweight live migration scheme for bare-metal clouds. The live migration is not supported in bare-metal clouds. *BLMVisor* utilizes a very thin hypervisor to allow pass-through accesses to physical devices from the guest OS.

In [40], Alshahrani et al. discussed existing solutions to secure the virtual machine live migration. To use the live migration in cloud computing might lead to a lot of attacks. Thus, it is necessary to implement the security requirements in the virtual machine live migration.

In [41], Nagarajan et al. adopted *Xen's* live migration mechanism for a guest operating system (OS) to migrate a task from the unhealthy node to the healthy one. They proved that the live migration reduced the cost of relocating the guest OS with the task.

In [42], Wang et al. designed the novel process-level live migration mechanism for the jobs by adopting the *Berkeley Lab Checkpoint/Restart (BLCR)* tool. The authors showed that the process level migration is significantly less expensive than migrating the entire OS images under the *Xen* virtualization.

In [43], Cores et al. presented a checkpoint-based approach to proactively migrate parallel applications when impending failures are notified. The authors adopted the *CPPC* framework, which is the portable and transparent checkpointing infrastructure. This approach is implemented at the application level, and thus, it is independent upon the operating system or a particular implementation.

In [44], Polze et al. proposed an architectural blueprint for the proactive virtual machine migration before a failure occurs. This architecture comprises the multi-level online failure prediction. The system is monitored ranging from the hardware to the application level for detecting errors that might lead to a failure. It is also observed that the virtual machine live migration is supported in popular hypervisors such as *VMWare ESX*, *Citrix XenServer*, and *KVM*.

In [45], Purushotham et al. proposed a new method for computing segmented backup for recovering from node and link failures in real-time IP communications. The proposed scheme

gives more number of backup segments computed by the *Min_SegBak algorithm*, and it gives better results in terms of the faster failure recovery and the efficient reuse of primary path. The authors showed that their proposal can recover the delay from component failures and guarantees on the message delivery latency for the distributed real-time applications, such as medical imaging, traffic control, and video conferencing, and so on.

In [46], Helmy and Rasheed presented an intelligent scheduling algorithm in the grid environment, which uses the *Fuzzy c-means (FCM) clustering technique* for predicting three classifications of job workloads and the *Ant Colony Optimization (ACO) algorithm* for allocating them to different grid nodes. For each part on the completion time of a node and the waiting time of each job, the experimental results showed that the scheduling system using the proposed algorithm outperforms all other algorithms and gives the optimal results.

In [47], Rawas et al. presented an *Energy-Efficient and Bandwidth-Aware workload allocation (EEBA)* method for data-intensive applications in geo-distributed data centers (DCs). The authors formulated the allocation problem as a multi-objective optimization problem, and proposed a meta-heuristic genetic algorithm to find a near-optimal solution. They proved that the energy consumption of the cloud DCs can be reduced due to load balancing the workload.

In [48], Bindu et al. proposed the *Optimized Scheduling mechanism using ACO algorithm (OSACO)* to schedule the task to the resources based on the minimization of the cost, execution time, and energy consumption. The simulation results proved that the client tasks are efficiently allocated to the available resources due to the ability of resource adjustment at run time.

In [49], Singhal et al. presented the algorithm based on the *Mutative ACO algorithm (MACO)* that adjusts the loads to be balanced in the cloud environment. *MACO* reduces the makespan time and further develops the fitness function, while keeping up with other QoS parameters. The algorithm deals with the resources successfully distributes over the data centers and shows the instances can be shifted between the servers depending on the available resources.

Chapter 8

Conclusion

In this thesis, I presented the study of the *user-PC Computing System (UPC)*. The UPC system allows various application programs to run on various PC environments for the *UPC workers* using the *Docker container technology*. Several useful functions are implemented. The effectiveness of each function is verified by conducting the extensive experiments using the testbed UPC system.

Firstly, I presented the design and implementation of the UPC system platform using *Docker*. By adopting *Docker*, the UPC system can accept various jobs or applications to run on user-PCs as the UPC workers with different platforms and environments. The user submitted jobs are transformed into the container based jobs, so called, *Docker image* at the *UPC master*. The *Docker* image consists of all the necessary software and it is distributed to the *UPC workers* for carrying out the execution process.

Secondly, I presented the implementation of the *web interface* in the UPC system for job submissions by the users. The *web server* is implemented using *Node.js*. It runs in a private network due to cost and security concerns. The users can easily submit the job and download the results by accessing to the *web interface* through the web browser.

Thirdly, I presented the implementation of the two *online job acceptance functions* for accepting the jobs from the application systems to enhance their processing capabilities. The application systems can submit their jobs to the UPC master through the shared storage or the FTP service. For evaluations, I adopt *APLAS* and *EPLAS* that have been developed in our group as the application systems. They require the high CPU capabilities. The experimental results showed that the jobs were successfully accepted from the application systems, and the total CPU time of completing the jobs was reduced by collaborating with the UPC system.

Fourthly, I presented the implementation of the *job migration function*, using the two open-source software, *CRIU* and *Podman*. This function will speed up the job completion and will be useful to overcome the memory overuse, or avoid hardware trouble. In the evaluations, I verified the validity of the implemented function and confirmed the effectiveness of the function by comparing the job completion performances when three scheduling algorithms were adopted. The results show that the total CPU time and *makespan* by the algorithms are significantly reduced by improving the resource utilizations and balancing the workloads of the workers through the dynamic migration.

Lastly, by extending the *dynamic job migration function*, I presented the implementation of the *job running backup function* in the UPC system. It periodically check-points the running job in a worker, and automatically migrates it to another healthy worker when the current worker meets a trouble. In the experiments, first, I verified the validity of the implemented job running backup function. Then, I confirmed the effectiveness of the function in reducing the CPU time loss when

the worker is shutdown unexpectedly while running a job.

In future studies, I will study the automatic *Docker* image generation for a newly submitted job, the use of GPU devices for *workers*, the automatic join/leave of *workers*, and the collaboration of multiple *masters* for the scalable UPC system.

Bibliography

[1] H. Htet, N. Funabiki, A. Kamoyedji, M. Kuribayashi, F. Akhter, and W.-C. Kao, "An implementation of user-PC computing system using Docker container," Int. J. Future Comput. Commun. (IJFCC), vol. 9, no. 4, pp. 66-73, Dec. 2020.

[2] N. Funabiki, K. S. Lwin, Y. Aoyagi, M. Kuribayashi, and W.-C. Kao, "A user-PC computing system as ultralow-cost computation platform for small groups," Appl. Theo. Comput. Tech., vol. 2, no. 3, pp. 10-24, Mar. 2017.

[3] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer com-puting system," Future Gen. Com. Sys., vol. 18, no. 4, pp. 561-572, Aug. 2002.

[4] H. Htet, N. Funabiki, A. Kamoyedji, and M. Kuribayashi, "Design and implementation of improved user-PC computing system," IEICE Tech. Report, NS2020-28, vol. 120, no. 69, pp. 37-42, Jun. 2020.

[5] A. Mouat, *Using Docker: developing and deploying software with containers*, O'Reilly Media, Inc., Dec. 2015.

[6] D. Herron, *Node.js web development*, 5th Ed., Packt Pub., Jul. 2020.

[7] H. Htet, N. Funabiki, A. Kamoyedji, X. Zhou, Y. W. Syaifudin, I. T. Anggraini, and M. Kuribayashi, "Implementations of online job acceptance functions in user-PC computing system," in Proc. IEEE 4th Global Conf. Life Sci. Tech. (LifeTech), pp. 121-122, Mar. 2022.

[8] "SFTP," Internet: `https://www.ssh.com/ssh/sftp`, (Accessed 13 Apr., 2022).

[9] "pCloud," Internet: `https://docs.pcloud.com/`, (Accessed 13 Apr., 2022).

[10] H. Htet, N. Funabiki, A. Kamoyedji, X. Zhou, and M. Kuribayashi, "An implementation of job migration function using CRIU and Podman in Docker-based user-PC computing system," in Proc. Int. Conf. Com. Commun. Manag. (ICCCM), pp. 92-97, Jul. 2021.

[11] H. Htet, N. Funabiki, A. Kamoyedji, X. Zhou, S. Sugawara, and W.-C. Kao, "An implementation of job running backup function in user-PC computing system," in Proc. IEEE 4th Int. Conf. Comput. Commu. Internet (ICCCI), pp. 156-161, Jul. 2022.

[12] A. Ratan, E. Chou, P. Kathiravelu, and M. O. Faruque Sarker, *Python network programming: conquer all your networking challenges with the powerful python language*, Packt Pub., Jan. 2019.

[13] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High performance MySQL: optimization, backups, and replication*, 3rd Ed., O'Reilly Media, Mar. 2012.

[14] R. McKendrick, *Monitoring Docker*, Packt Pub., Dec. 2015.

[15] G. Rodola, "Efficient I/O with zero-copy & psutil," Internet: https://gmpy.dev/static/efficient-io-with-zerocopy-syscalls.pdf, (Accessed 13 Apr., 2022).

[16] "SSHFS," Internet: https://github.com/libfuse/sshfs, (Accessed 13 Apr., 2022).

[17] Y. W. Syaifudin, N. Funabiki, M. Mentari, H. E. Dien, I. Mu'aasyiqiin, M. Kuribayashi, and W.-C. Kao, "A web-based online platform of distribution, collection, and validation for assignments in Android programming learning assistance system," Eng. Letter., vol. 29, no. 3, pp. 1178-1193, 2021.

[18] I. T. Anggraini, A. Basuki, N. Funabiki, X. Lu, C.-P. Fan, Y.-C. Hsu, and C.-H. Lin, "A proposal of exercise and performance learning assistant system for self-practice at home," Adv. Sci. Tech. Eng. Syst. J. (ASTESJ), vol. 5, no. 5, pp. 1196-1203, 2020.

[19] "CRIU," Internet: https://www.criu.org/Main_Page, (Accessed 13 Apr., 2022).

[20] H. Gantikow, S. Walter, and C. Reich, "Rootless containers with Podman for HPC," in Proc. Int. Conf. High Perform. Comput. (HiPC), pp. 343-354, Oct. 2020.

[21] M. M. Islam, N. Funabiki, M. Kuribayashi, S. K. Debnath, K. I. Munene, K. S. Lwin, R. W. Sudibyo, and M. S. A. Mamun, "Dynamic access-point configuration approach for elastic wireless local-area network system and its implementation using Raspberry Pi," Int. J. Netw. Comput. (IJNC), vol. 8, no. 2, pp. 254-281, Jul. 2018.

[22] "FFmpeg," Internet: https://github.com/FFmpeg/FFmpeg, (Accessed 13 Apr., 2022).

[23] J. Latt, et al., "Palabos: parallel lattice Boltzmann solver," Comput. Math. Appli., vol. 81, pp. 334-350, Apr. 2021.

[24] "Open Porous Media (OPM) flow," Internet: https://opm-project.org/?p=1412, (Accessed 13 Apr., 2022).

[25] "Neural network using TensorFlow," Internet: https://github.com/aymericdamien/TensorFlow-Examples/tree/master/examples/3_NeuralNetworks, (Accessed 13 Apr., 2022).

[26] "Converter," Internet: https://github.com/andyp123/mp4_to_mp3, (Accessed 13 Apr., 2022).

[27] B. Quetier, V. Neri, and F. Cappello, "Selecting a virtualization system for grid/p2p large scale emulation," in Proc. EXPGRID Workshop, pp. 19-23, Jun. 2006.

[28] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in Proc. 16th Euro Micro Conf. Paral. Dist. Netw-Based Process. (PDP), pp. 233-240, Feb. 2013.

[29] P. J. Won and H. Jaegyoon, "Container-based cluster management system for user-driven distributed computing," KIISE Trans. Comput. Pract., vol. 21, no. 9, pp. 587-595, Sep. 2015.

[30] A. Jaikar, S. Bae, H. Han, B. Kong, SA. Shah, and SY. Noh, "OpenStack and docker comparison for scientific workflow w.r.t. execution and energy," in Proc. 4th Int. Workshop Effi. Data Cent. Sys. (EDCS), pp. 1-5, Jun. 2016.

[31] M. Sindi and J. R. Williams, "Using container migration for HPC workloads resilience," in Proc. IEEE Conf. High Perf. Ext. Comput. (HPEC), pp. 1-10, Sep. 2019.

[32] M. Zeynep and P. Angin., "A secure model for efficient live migration of containers," J. Wirel. Mob. Netw. Ubiqui. Comput. Depend. Appl. (JoWUA), vol. 10, no. 3, pp. 21-44, Sep. 2019.

[33] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: transparent check pointing for cluster computations and the desktop," in Proc. Int. Symp. Paral. Dist. Process. (IPDPS), pp. 1-12, May 2009.

[34] O. Alrajeh, M. Forshaw, and N. Thomas, "Using virtual machine live migration in trace-driven energy-aware simulation of High-Throughput computing systems," Sustainable Comput.: Info. Sys., vol. 29, p. 100468, Mar. 2021.

[35] C. Yu and F. Huan, "Live migration of Docker containers through logging and replay," in Proc. Int. Conf. Mech. Indus. Info., pp. 623-626, Oct. 2015.

[36] W. Fan, Z. Han, P. Li, J. Zhou, J. Fan, and R. Wang, "A live migration algorithm for containers based on resource locality," J. Signal Process. Sys., vol. 91, no. 10, pp. 1077–1089, Oct. 2019.

[37] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of Linux containers," in Proc. High Perf. Comput. (HiPC), pp. 184-193, Jun. 2018.

[38] P. S. Junior, D. Miorandi, and G. Pierre, "Stateful container migration in geo-distributed environments," in Proc. IEEE Int. Conf. Cloud Comput. Tech. Sci., pp. 49-56, Dec. 2020.

[39] T. Fukai, T. Shinagawa, and K. Kato, "Live migration in bare-metal clouds," IEEE Tran. Cloud Comput., vol. 9, no. 1, pp. 226-239, Jan. 2021.

[40] H. Alshahrani, A. Alshehri, R. Alharthi, A. Alzahrani, D. Debnath, and H. Fu, "Live migration of virtual machine in cloud: survey of issues and solutions," in Proc. Int. Conf. Secur. Manag. (SAM), pp. 280-285, 2016.

[41] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive fault tolerance for HPC with Xen virtualization," in Proc. Int. Conf. Supercomput. (ICS), pp. 23-32, Jun. 2007.

[42] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in HPC environments," in Proc. IEEE Supercomput. Conf. (SC), pp. 1-12, Nov. 2008.

[43] I. Cores, G. Rodriguez, P. Gonzalez, and M. J. Martin, "An application level approach for proactive process migration in MPI applications," in Proc. IEEE Int. Conf. Paral. Distri. Comput., Appl. Tech. (PDCAT), pp. 400-405, Oct. 2011.

[44] A. Polze, P. Troger, and F. Salfner, "Timely virtual machine migration for pro-active fault tolerance," in Proc. IEEE Int. Symp. Object/Comp./Service-Orient. Real-Time Distrib. Comput. Works. (ISORCW), pp. 234-243, Mar. 2011.

[45] B. Purushotham, C.D. Rao, and N. Padmaja, “An efficient recovery scheme for node and link failures in real-time IP communications,” IAENG Int. J. Comput. Sci., vol. 37, no. 2, p. 156-163, Jun. 2010.

[46] T. Helmy and Z. Rasheed, “Independent job scheduling by Fuzzy C-mean clustering and an Ant optimization algorithm in a computation grid,” IAENG Int. J. Comput. Sci., vol. 37, no. 2, p. 136-145, Jun. 2010.

[47] S. Rawas and A. Zekri, “EEBA: Energy-Efficient and Bandwidth-Aware workload allocation method for data-intensive applications in cloud data centers,” IAENG Int. J. Comput. Sci., vol. 48, no. 3, pp. 703-715, Sep. 2021.

[48] G. Bindu, K. Ramani, and C. S. Bindu, “Optimized resource scheduling using the meta heuristic algorithm in cloud computing,” IAENG Int. J. Comput. Sci., vol. 47, no. 3, pp. 360– 366, Aug. 2020.

[49] S. Singhal and A. Sharma, “Mutative ACO based load balancing in cloud computing,” Eng. Letter., vol. 29, no. 4, pp. 546-555, Dec. 2021.