# DEVELOPMENT OF AN ENHANCED CHECKPOINTING TECHNIQUE IN GRID COMPUTING USING PROGRAMMER LEVEL CONTROLS

**BY**

**Garba ALIYU**
**P14SCMT8062**

**Department of Computer Science**
**Ahmadu Bello University, Zaria**
**Nigeria**

**JANUARY, 2017**

# DEVELOPMENT OF AN ENHANCED CHECKPOINTING TECHNIQUE IN GRID COMPUTING USING PROGRAMMER LEVEL CONTROLS

By

Garba ALIYU, B.Sc. Computer Science (ABU) 2009
P14SCMT8062

A DISSERTATION SUBMITTED TO THE SCHOOL OF POSTGRADUATE STUDIES,
AHMADU BELLO UNIVERSITY, ZARIA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF A MASTER OF SCIENCE DEGREE IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF PHYSICAL SCIENCE
AHMADU BELLO UNIVERSITY, ZARIA
NIGERIA

JANUARY, 2017

## Declaration

I Garba Aliyu declare that work in this Dissertation entitled "**Development of An Enhanced Checkpointing Technique in Grid Computing Using Programmer Level Controls**" has been carried out by me in the Department of Computer Science. The information derived from the literature has been duly acknowledged in the text and a list of references provided. No part of this dissertation was previously presented for another degree or diploma at this or any other institution.

| _____ | _____ | _____ |
|---|---|---|
| Name of Student | Signature | Date |

## Certification

This dissertationentitledDEVELOPMENT OF AN ENHANCED CHECKPOINTING TECHNIQUE IN GRID COMPUTING USING PROGRAMMER LEVEL CONTROLS by ALIYU Garbameets the regulations governing the award of the degree of M.Sc. in Computer Science of the Ahmadu Bello University, and isapproved for its contribution to knowledge and literary presentation.

(Signature) _____

_____ Date _____

Chairman, Supervisory Committee

(Signature) _____

_____ Date _____

Member, Supervisory Committee

(Signature) _____

_____ Date _____

Member, Supervisory Committee

(Signature) _____

_____ Date _____

Head of Department

(Signature) _____

_____ Date _____

Dean, School of Postgraduate Studies

## Dedication

All thanks be to Almighty Allah for making impossible to be possible. I dedicate my dissertation work to my family and many friends. A special feeling of gratitude to my loving parents, Alhaji Aliyu Bello and Hajiya Zainab Aliyu whose words of encouragement and push for tenacity remains sticky in my ears. My sisters and brothers have never left my side and are very special. I also dedicate this dissertation to my many friends who have supported me throughout the process. I will always appreciate all they have done, especially my colleagues for helping me develop my technical skills, Malcolm Forsyth for the many hours of proofreading and helping me to achieve this work.

**Acknowledgment**

My deepest gratitude goes to Almighty Allah who has provided all that was needed to complete this dissertation and the program for which it was undertaken. There was never lack or want.

A special thanks to theChairman of my Supervisory Committee, Prof. Sahalu B. Junaidu, for his countless hours of reflecting, reading, encouraging, and most of all patience throughout the entire process and Member, Supervisory Committee Dr. A. F. Donfack Kana who have generously given his time and expertise to better my work.

I would like to acknowledge and thank my department and Institute of Computing & ICT for allowing me to conduct my research and providing any assistance requested. Special thanks go to the members of staff of Department of Computer Science and Institute of Computing & ICT for their continued support especially Mr. Malcolm Forsyth who has given me his time.

I also from the depth of my heart appreciate my lovely wife Zainab Garba and my children Othman Garba Aliyu, Aisha Garba Aliyu and Hajara Garba Aliyu(truly, you inspired me) who even at such tender ages have had to endure so much stress and discomfort just for me. This page can't tell it all.

Finally, I thank my family and friends for their prayers, words of motivation and words of comfort that come in just in time. May Allah bless you all.

**Abstract**

Grid computing is a collection of computer resources from multiple locations assembled to provide computational services, storage, data or application services. Grid computing users gain access to computing resources with little or no knowledge of where those resources are located or what the underlying technologies, hardware, operating system, and so on are. Reliability and performance are among the key challenges to deal with in grid computing environments. Accordingly, grid scheduling algorithms have been proposed to reduce the likelihood of resource failure and to reduce the overhead of recovering from resource failure. Checkpointing is one of the faulttolerance techniques when resources fail. This technique reduces the work lost due to resource faults but can introduce significant runtime overhead. This research provided an enhanced checkpointing technique that extends a recent research and aims at lowering the runtime overhead of checkpoints. The results of the simulation using GridSim showed that keeping the number of resources constant and varying the number of gridlets, improvements of up to 9%, 11%, and 11% on throughput, makespan and turnaround time, respectively, were achieved while varying the number of resources and keeping the number of gridlets constant, improvements of up to 8%, 11%, and 9% on throughput, makespan and turnaround time, respectively, were achieved. These results indicate the potential usefulness of our research contribution to applications in practical grid computing environments.

## Table of Contents

## List of Figures

## List of Tables

## List of Abbreviations

| | |
|---|---|
| ACO | Ant Colony Optimization |
| ALC | Application-Level Checkpointing |
| ALC-P | Programmer Level Checkpointing |
| ALC-U | User Level Checkpointing |
| ATAT | Average Turnaround Time |
| BCS | Buffered Co-Scheduling |
| BLCR | Berkeley Lab Checkpoint/Restart |
| CFSM | Extended Finite State Machine |
| CIC | Communication induced checkpointing |
| CPPC | Controller/Pre-compiler for Portable Checkpointing |
| CPR | Checkpoint Recovery |
| CPU | Central Processing Unit |
| FIBR | Fault Index Based Rescheduling |
| FR | Resource Failure Rate |
| FSM | Finite State Machine |
| GIS | Grid Information System |
| GRB | Grid Resource Broker |
| I/O | Input/Output |
| ICT | Information and Communication Technology |
| IDE | Integrated Development Environment |
| MIPS | Millions Instruction Per Second |
| MLC | Mixed Level Checkpointing |
| MTTR | Minimum Total Time to Release |
| OLP | On Line Protocol |
| OS | Operating System |
| Pod | Process Domain |
| PVM | Parallel Virtual Machine |
| QoS | Quality of Service |
| SLC | System Level Checkpointing |
| SLC-H | Hardware Level |
| SLC-K | Kernel (Operating System) Level |
| TICK | Transparent Incremental Checkpointer at Kernel level |
| TTR | Total Time to Response |
| XDR | External Data Representation |

## Definition of Terms

**Bandwidth**: The amount of data that can be transmitted in a fixed amount of time.

**Checkpoint latency**: The duration of time required to save the checkpoint.

**Checkpoint overhead**: The increase in the execution time of an application caused by the checkpointing.

**Computing resources**: Resources which allow users to execute the required application.

**Error:**The part of the system's state that may lead to a failure.

**Fail tendency:** The percentage of the tendency of grid resources to fail.

**Failure:**An event that occurs when the delivered service deviates from its correct behavior. A service is said to have failed if it could not comply with the promised functional specifications.

**Fault:** The adjudged or hypothesized cause of an error. A fault is an unwanted but possible constellation of system states.

**Gridlet:** A package that contains all the information related to the job and its execution management details such as job length expressed in MIPS, disk I/O operations, the size of input and output length, and the job originator.

**Grid Information System (GIS**): Contains information about the available grid resources with their computing capacity and cost at which they offer services to grid users.

**Grid Resource Broker (GRB)**: Maps jobs received from users to the grid resources.

**Grid Scheduler**: Schedules jobs to grid resources.

**Grid scheduling:** Process of making scheduling decisions involving resources over multiple administrative domains.

**Global scheduling:** The problem of deciding where to execute the job. Global scheduling is not responsible for the final mapping of the job's processes onto CPUs, this is done by the machine's local scheduler, i.e., operating system.

**Heterogeneity:** Resources are said to be heterogeneous when they do not have all the exact same characteristics. For instance, the set of processors in a grid is heterogeneous if they do not all have the same processing capabilities or the same amount of memory.

**Job:** The user's program or application which is executed in the Grid. The job is also subject of the scheduling process.

**Job Migration:** Sometimes it happens that due to some reasons a job cannot be completely executed on a particular machine. At the time of failure of any task, task can be migrated to another machine.

**Local scheduling:** Is used to assign the process (es) of the job to the CPU(s) of the single machine.

**Makespan**: The time at which the application execution is completed. It is measured from the time the first job is sent to the grid, until the last job comes out of the grid.

**Process:** Referred as a task, is an instance of a program in execution.

**Quality of Service (QoS):** The ability to provide different priority to different jobs and users, or to guarantee a certain level of performance to a job.

**Re-execution**: A task whose execution is not reliable enough (for instance, because its execution speed is below a threshold) is executed twice (or more) to enhance its reliability.

**Resource:** Any hardware or software component such as a cluster, a supercomputer or a storage repository. Resources perform an operation, such as memory for data storage, processor for data processing, bandwidth for data transfer and so on.

**Resource Characteristics:** Object that stores the properties of a Grid resource: architecture, OS, list of Machines, allocation policy: time- or space-shared, time zone.

**Resource Scheduling**: The process of mapping resources based on the requirements, characteristics which are specified in a query.

**Resource selection**: The problem of deciding which, among all available resources, should be used to solve a given problem.

**Schedulers**: Are responsible for scheduling, and they manage various tasks – resource discovery and resource selection - required for effective application execution.

**Scheduling**: The problem of deciding at what time (when), and on which resource (where), to execute each of the (atomic) tasks that must be executed on the given platform.

**Sites**: Are autonomous units composed of one or multiple resources. They also called nodes.

**Task**: The smallest unit that can be scheduled by the scheduler and allotted to a resource. Task has parameters such as priority, memory requirement, deadline, etc.

**Task scheduling**: Assigning of tasks to a group of resources that are distributed in multiple domains.

**Throughput**: Thenumber of jobs completed in a certain period of time

**Turnaround Time**: The interval of time elapsed from submission of a job to the time of its completion

# CHAPTER ONE

# INTRODUCTION

## 1.1 Background of the Study

Grid computing uses a computer network in which each computer's resources are shared with every other computer in the system. In view of this, computing becomes pervasive and individual users (or client applications) gain access to computing resources (processors, storage, data, applications, and so on) as needed with little or no knowledge of where those resources are located or what the underlying technologies, hardware, operating system, and so on are. The main objective in grid scheduling is to finish a job or application as soon as possible(Harshadkumar and Vipul, 2014). Fault tolerance is an important property for large scale computational grid systems, where geographically distributed nodes cooperate to execute a task in order to achieve a high level of reliability and availability. A common approach to guarantee an acceptable level of fault tolerance in scientific computing is to use checkpointing. When a task fails it can be restarted from its most recently checkpointed state rather than from the beginning, which reduces the system loss and ensures reliability (Bakhta and Ghalem, 2014).

## 1.2 Motivation

The ability to checkpoint a running application and restart it later can provide many useful benefits like fault recovery, advanced resource sharing, dynamic load balancing and improved service availability. A fault-tolerant service is essential to satisfy QoS requirements in grid computing. However, excessive checkpointing results in performance degradation. Thus there is the need to improve the performance by reducing the number of times that checkpointing is invoked. The research on Grid computing can be helpful and applicable to some industries that have successfully adopted grid computing technology such as:

1. The financial services industry uses it for derivatives analysis, statistical analysts and portfolio risk analysis.
2. The insurance industry uses it for certain tasks.
3. Life sciences use grid technology to carry out cancer research and protein sequencing and folding (Hossein, 2014)

### 1.3 Research Problem

Fault tolerance is the technique to give the required services in the presence of fault or error within the system. The aim is to avoid failures in the presence of faults and provide services as per requirement. In fault tolerance, the fault is detected first and recovers them without participation of any external agents. The main issue in fault tolerance is how, where, and which technique is used to tolerate fault in distributed systems(Bashir and Kumar, 2013).The checkpoint is one of the most popular techniques to provide fault-tolerance on unreliable systems. It is a record of the snapshot of the entire system state in order to restart the application after the occurrence of some failure. The checkpoint can be stored on temporary as well as stable storage. However, the efficiency of the mechanism is strongly dependent on the length of the checkpointing interval. Frequent checkpointing may worsen the overhead. It is necessary to reduce the number of checkpointing occurrences to an optimal number in order to minimise the overhead.

### 1.4 Research Aim and Objectives

The aim of this research is to develop an enhanced checkpointing-based fault-tolerance system by reducing its runtime overhead using programmer-level checkpointing controls.

The specific objectives of the research are:

1. Develop a mechanism that provides users the flexibility and control to insert checkpointing code as desired.
2. Provide space efficiency by saving only the data necessary to recover an application.
3. Implement and simulate the enhanced checkpointing algorithm using Grid Sim Toolkit – version 5.2.
4. Evaluate performance of the enhanced checkpointing algorithm side-by-side that ofIdris(2015)

### 1.5 Research Methodology

The following procedures are to be adopted for this research work:

1. Specify the number of checkpoints prior to the execution of the user's job which can be determined based on the number of failure rate, response time tendency of failures of the failed resource. This will be decided dynamically and programmatically.
2. Insert the checkpoints after the job has been scheduled. The number of checkpoints specified will be evenly distributed. The distribution of the checkpoints (checkpoint interval) will be implemented based on the number of checkpoints and response time of the failed resource.

   $\left\lfloor \frac{rtime}{n} \right\rfloor$, n $> 0$, where n is the number of checkpoints , rtime is the response time and the ratio is the checkpoints interval
3. Save the state of the process after every checkpoint to assure coherence and replicate the checkpoint files at many nodes to ensure their availability. Purge the previous checkpoint data and leave the last checkpoint (restore point) for recovery in case of fault.

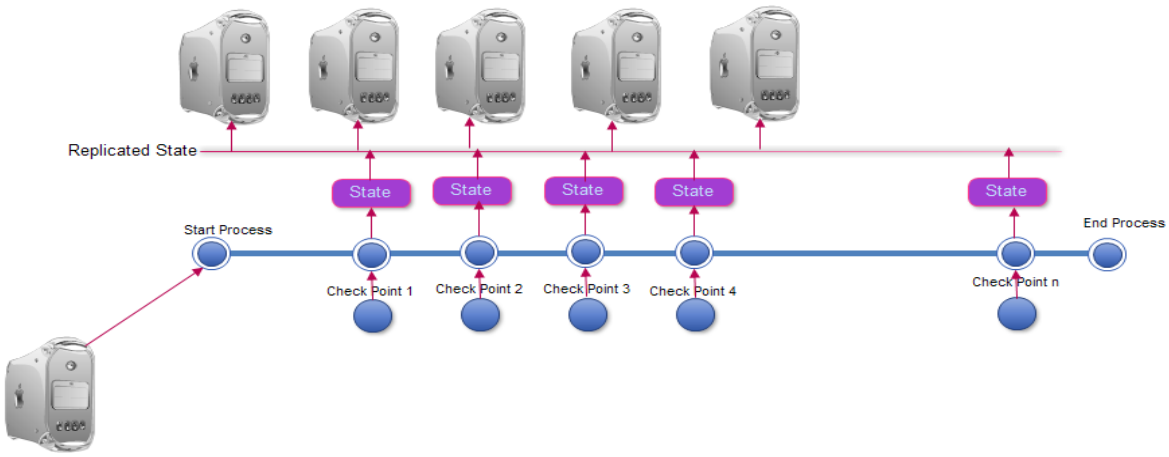


Figure 1.1: Check Pointing Techniques

4. Implement and simulate the proposed improved checkpointing algorithmusing GridSim Toolkit – version 5.2. Netbeans is used as an IDE (Integrated Development Environment) where a grid simulation toolkit for resource modelling and application scheduling for parallel and distributed computing containing Java classes will be imported into the environment.
5. Reschedule the job execution to the active nodes using the same scheduling algorithm proposed by Idris(2015).
6. Evaluate performance of our algorithm in comparison to that of Idris (2015) with respect to Makespan, Turnaround Time and Throughput.

## 1.6 Limitations/Challenges

The purpose of this research work is to reduce the checkpointing proposed and implemented by Idris(2015). The basic idea behind checkpoint-recovery is the saving and restoration of the system state. By saving the current state of the system periodically or before critical code sections, the baseline information needed for the restoration of a state lost as a result of a system failure, is provided; while the cost caused by checkpointing should be reduced to a minimum.

The research work is limited because of the fact that the cost of checkpoint-recovery (i.e. the amount of computation lost due to rollback in case of a fault) can be high and that the checkpointing is done at programmer level.

# CHAPTER TWO

# LITERATURE REVIEW

## 2.1 Introduction

Checkpointing is an indispensable fault tolerance technique used in many applications that take a long time to complete their tasks. These applications periodically write large volumes of snapshot data to persistent storage, in an attempt to capture their current state. In the event of a failure, applications recover by rolling back their execution state to a previously saved checkpoint (Samer *et al.*, 2013). Checkpointing/rollback-recovery strategies have been an attractive approach for providing fault tolerance to distributed applications (Siva *et al.,* 2010). Checkpoints are periodically saved on stable storage servers and the recovery from a processor failure is done by restoring the system to the last saved state (Ritu *et al.,* 2011). The system can thus avoid the total loss of computations in case of failure. One of the popular systems that use checkpointing to ensure fault tolerance is distributed discrete event simulation. In this type of environment the simulated system is partitioned into a set of sub-systems that are simulated by a set of processes that communicate by sending and receiving time-stamped messages (Lin, 2002). The state of each process in distributed discrete event simulation must be saved regularly to ensure a correct rollback in case of failure todecrease the system loss. However, it is proved that the performance of this system is dominated by the efficiency of the checkpointing strategy used. Thus, it is important to analyse and know more about this fault tolerance technique (Perrone et *al*., 2010).

## 2.2 Fault Tolerance

Fault tolerance is the ability of a system to perform its function correctly even in the presence of faults. Fault tolerance makes the system more dependable. A

complementary but separate approach to increase dependability is fault prevention. This consists of techniques, such as inspection, whose intent is to eliminate the circumstances by which faults arise (Ritu *et al.,* 2011). Fault tolerance Latchoumy *et al.,*(2011) assured that there is a specification of what constitutes correct behaviour of resources. A failure occurs when an actual running system such as in the grid environment deviates from this specified behaviour. The cause of a failure is termed an error. An error represents an invalid system state, one that is not allowed by the system behaviour specification. The error itself is the result of a defect in the system or a fault. In other words, a fault is the root cause of a failure which means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors at later stages. Similarly, a single error may lead to multiple failures (Gupta, 2011).

### 2.2.1 Fault Detection

Amoon (2012) Detection of faults is limited to the types of faults that a system may handle. The potentially exhaustive list of faults should generally avoid including those that are arbitrary or Byzantine. This will simplify the system design. The remaining are omission, crash, and timing failures which should be detected by network monitors, or in some other way. It is the responsibility of the monitoring process to notify the group members so that a decision can be made. In the case of crash failures, a crashed process may be detected when the process fails to respond to a message within a timeout period. The sending process then has the responsibility to notify all other members of the group that the process failed.

### 2.2.2 Fault Rectification

The key aspect of fault rectification in this work is based on implementing the check point mechanism (Gupta, 2011). This mechanism introduces two components namely

the checkpoint manager and the check point server. The focus of the checkpoint manager is to manage the rate of failure and the total number of failures occurring in the system as such in the computational grid environment.

**The Checkpoint Manager**: It receives the scheduled tasks from the scheduler and sets checkpoints dynamically Antony *et al.*,(2010) based on the failure rate of the resource on which it is scheduled. Then it submits the job to the resource. The checkpoint manager receives a job completion message or a job failure message from the grid resource and responds to that accordingly. During execution, if job failure occurs, the job is rescheduled from the last checkpoint instead of running from the scratch. The checkpoint manager implements the checkpoint setter algorithm to set job checkpoints.

**The Checkpoint Server**: On each checkpoint set by the checkpoint manager, the job status is reported to the checkpoint server. The checkpoint server saves the job status and returns it on demand i.e., during a job or resource failure. For a particular job, the checkpoint server discards the result of the previous checkpoint when a new value of a checkpoint result is received (Gokuldev *et al.*, 2014).

Faults may be classified based on several factors.

1. Network Faults: Faults like packet loss and packet duplication or packet corruption.
2. Physical faults: faults that occur in memory, processor or storage media.
3. Media faults: disk head crashes.
4. Resource faults: unavailability of resources, lack of resources.
5. Communication faults: eg, protocol incompatibility.
6. Service time faults: eg, expiry of service time.
7. System performance degradation(Shahana, 2013)

Commonly utilized techniques for providing fault tolerance are job checkpointing and replication. Both techniques mitigate the amount of work lost due to changing system

availability but can introduce significant runtime overhead. The latter largely depends on the length of checkpointing interval and the chosen number of replicas, respectively. In case of complex scientific workflows where tasks can execute in well-defined order reliability is another biggest challenge because of the unreliable nature of the grid resources (Ritu and Awadhesh, 2011)**.**

### 2.2.3 Checkpointing

Checkpointing is one of the most popular techniques to provide fault-tolerance on unreliable systems. It is a record of the snapshot of the entire system state in order to restart the application after the occurrence of some failure. The checkpoint data can be stored on temporary as well as stable storage (Ritu and Awadhesh, 2011). However, the efficiency of the mechanism is strongly dependent on the length of the checkpointing interval. Frequent checkpointing may enhance the overhead, while lazy checkpointing may lead to loss of significant computation. Hence, the decision about the size of the checkpointing interval and the checkpointing technique is a complicated task and should be based upon the knowledge about the application as well as the system. Therefore, various types of checkpointing optimization have been considered by the researchers, e.g., (i) full checkpointing or Incremental checkpointing, (ii) unconditional periodic checkpointing or optimal (dynamic) checkpointing, (iii) synchronous (coordinated) or asysnchronous (uncoordinated) checkpointing, and (iv) kernel, application or user-level checkpointing (Ritu and Awadhesh, 2011).

### 2.2.4 Full Checkpoint or Incremental Checkpoint

A full checkpoint is a traditional checkpoint mechanism which occasionally saves the total state of the application to local storage. However, the time consumed in reforming the checkpoint and the storage required to save it is very large (Ritu and Awadhesh, 2011).

The incremental checkpoint mechanism was introduced to reduce the checkpoint overhead by saving the pages that have been changed instead of saving everything (Agarwal *et al.*, 2004). In the incremental checkpoint scheme, the first checkpoint is typically a full checkpoint. After that, only modified pages are checkpointed after some predefined interval. When large numbers of pages have been modified another full checkpoint is taken. In order to recover the application, a saved state will be loaded from the last full checkpoint and the changed pages from each incremental checkpoint that follow the last full checkpoint. This results in a more expensive recovery cost than the recovery cost of the full checkpoint mechanism (Ritu and Awadhesh, 2011).

### 2.2.5 Uncoordinated or Coordinated Checkpointing

In uncoordinated checkpointing each process takes its checkpoint independently of the other processes though it may lead to a domino effect (processes may be forced to rollback up to the execution beginning). Since there is a chance of losing the whole computation, these protocols are not popular in practice (Ritu and Awadhesh, 2011).

Coordinated checkpoint protocols produce consistent checkpoints; hence, the recovery process is simple to implement. Communication induced checkpointing (CIC) tries to take advantage of uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it piggybacks causality dependencies in all messages and detects risks of inconsistent states. When such a risk is detected, some processes are forced to checkpoint. A detailed survey of checkpointing protocols may be found in (Alvisi , *et al.*, 1999).

The checkpointing strategies can be classified according to their synchronization type (Siva and Syam, 2010). Coordinated and uncoordinated synchronization are two fundamental approaches to checkpointing and recovery. There is another popular

classification based on the abstraction level in which the state of a process is saved. There are a large number of design choices of abstraction level (Fontes *et al.*,2004).To understand this, consider the typical system stack shown in Figure. 2.1.

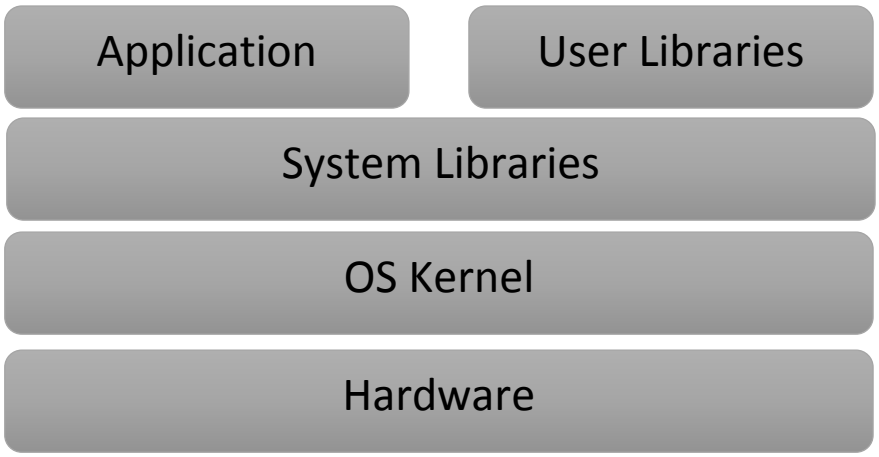


Figure 2.1: System Stack(Bakhta and Ghalem, 2014)

It contains the original user application, which may be compiled and linked with user-level libraries. It may use system libraries, which reside on top of the OS kernel, which executes directly on top of hardware. Any of these levels may be modified with checkpointing functionality, so it is possible to save and restore the stack levels above it. Furthermore, it is possible to insert new layers between these standard layers that enable checkpointing of the layers above, such as the work proposed in Bronevetsky *et al.*,(2003) where a coordination layer is created between the system library and the user library to ensure more transparency and to ensure also that system library will not be modified. The insertion of a new library can be necessary if the code of the existing library is not available or to ensure more transparency (Bronevetsky *et al.*, 2003). There are many survey papers proposed in the literature that study the abstraction level of checkpointing (Sancho *et al.*, 2005). However,toour knowledge, none of those surveyed studied the solutions to the problems and the limitations of each level. Our present work summarizes the majority of existing solutions of abstraction levels and proposes a comparative study between several papers.

## 2.3 Checkpointing Levels

There are many types of abstraction level classifications in the literature. In (Sancho *at al.*,2005), the classification is based on the implementation techniques. However, the authors in (Siva *at al.*, 2010) and (Silva and Silva, 1998) propose another classification that uses transparency as a criterion. In both previous works and in many other papers the application-level is referred to as user-level. For our discussion we will distinguish them by their transparency with regard to the application program. Our classification categorizes the checkpointing levels into three different types: application level, system level and mixed(hybrid) level. Figure 2.2 illustrates the proposed classification.

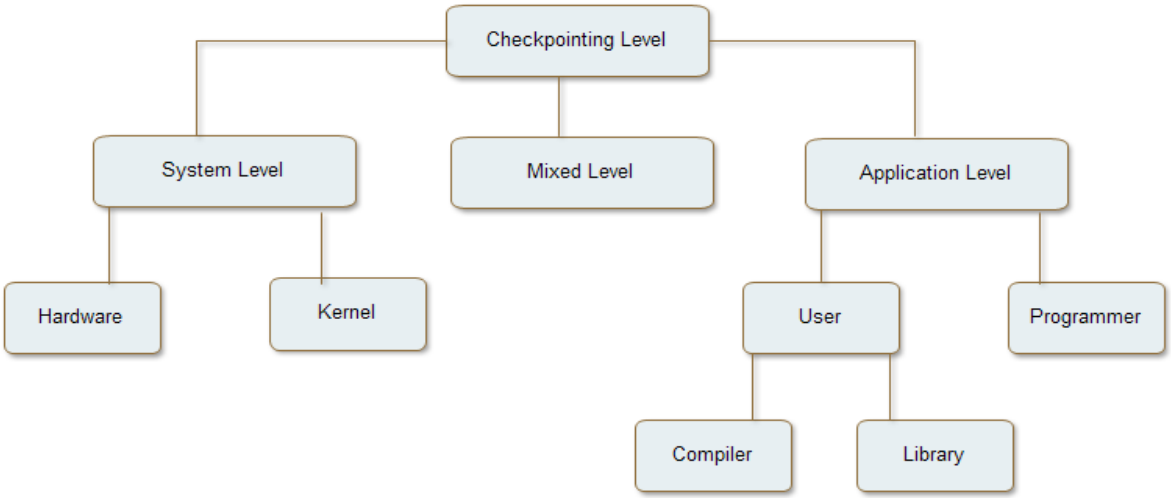


Figure 2.2: Checkpointing Level Classification(Bakhta and Ghalem, 2014)

### 2.3.1 System Level (SLC)

System-level checkpointing is a technique which provides automatic, transparent checkpointing of applications at the operating system or middleware level. The application is seen as a black box, and the checkpointing mechanism has no knowledge about any of its characteristics. Typically, this involves capturing the complete process image of the application. There are two main approaches to checkpointing at a system level: kernel (operating system) implementation and hardware implementation (Sivaand

Syam, 2010). The system-level checkpointing can be activated by a system call: a kernel-mode signal handler or a kernel thread.

### 2.3.2 Kernel (Operating System) Level (SLC-K)

In kernel space every data structure relevant to a process's state is readily accessible: these include registers, memory regions, file descriptors, signal state, and more. This accessibility enormously simplifies the implementation of checkpoint/restart operations, though it requires somewhat more knowledge of kernel internals. The Berkeley Lab Checkpoint/Restart (BLCR) method proposed in Hargrove and Duell(2006) is kernel level checkpointing in distributed systems. It uses coordinated checkpointing activated by a special *callback* thread. To reduce the overhead caused by the checkpointing, BLCR focuses on the management of I/O strategies.

Transparent Incremental Checkpointer at Kernel level (TICK) Gioiosa *et al.*,(2005) is another system that uses kernel level checkpointing. TICK considers transparency as the most important criterion in scalable systems so the system-level is the perfect checkpointing to ensure transparency in grid calculations. TICK uses buffered co-scheduling (BCS) Petrini *et al.,* (2004) to ensure checkpointing consistency. In BCS the messages are buffered and scheduled before transmission to exclude the late and transit messages. ZAP Osman *at al.*,(2002) uses kernel-level checkpointing formigration. It provides a virtualization mechanism called Pod (Process domain) to cope with resource consistency, resource conflicts, and resource dependencies that arise when migrating processes between machines with different persistent states. AlKiswany *et al.*,(2008) proposed Stdchk which uses kernel-level checkpointing and it focuses on reducing the storage time by reducing the checkpointing size.

### 2.3.3 Hardware Level (SLC-H)

Checkpointing may be supported by purpose-designed hardware. As with operating system level implementations, this approach can be entirely transparent to users. But hardware-level checkpointing is of limited importance precisely because it relies on custom hardware. The work proposed in Koch *at al.*,(2007) uses checkpointing at the hardware level to prevent faults in reconfigurable systems. It assumes that each hardware module can be modelled by a finite state machine (FSM). This FSM will be extended to CFSM (checkpointed FSM) by adding a new module related to the FSM that controls the checkpointing interval and the placement of the checkpoint file. ReVive Koch *et al.*,(2007) requires modifications to the directory controllers of the machine that intercepts the I/O to perform memory based distributed parity protection and logging in the background. Parity protection is used to protect the checkpoint file since it will be transferred via a network to other nodes where it will be stored. Logging is used to ensure the atomicity of transitions and ignoring the modifications in memory in case of failure. In logging buffering the checkpoint value is copied to a log, while the original location is modified and remains part of the working state. SafeNet Sorin, *et al.*,(2002) uses the same idea proposed in ReVive except that in SafeNet the parity protection is not used and the checkpoint files are stored only in main memory. SafeNet uses uncoordinated checkpointing with pipelined validation rather than the coordinated checkpointing used in ReVive. These differences justify why SafeNet requires more hardware resources than Revive.

### 2.3.4 Application-Level (ALC)

A typical approach to avoid many of the complexities of checkpointing is based on taking an application-centred point of view, and exploiting knowledge of the structure

and behaviour of a given application. In this approach the checkpointing is initiated, and to some degree managed, from within the application (Silva and Silva, 1998). The application programmer identifies program points at which all essential states can be captured from within the application (Walters and Chaudhary, 2006). A common scheme of implementation is to install a signal handler for a default signal offered by the kernel to automatically initiate the checkpoint operations. The signal handlers are defined at user level and are invoked by the kernel. This signal can be triggered by a timer that periodically interrupts the application (Sancho *at al.*, 2005). The application level can be classified according to the transparency tothe user. In other words, how much the user is involved in the checkpointing process (Siva and Syam, 2010).

### 2.3.5 Programmer Level (ALC-P)

This is also known as manual code insertion. Atthis level the programmer manually inserts the checkpointing code into the application code in order to save its state and to recover after a fail-stop failure. The programmer inserts code at points in the application where he wants checkpointing to occur. The work introduced in (Li and Yeung, 2008) determines when the checkpoints are taken by identifying the main controlling loops in the benchmarks (usually the outer loops associated with major program phases), and inserts the checkpointing calls at the top of each iteration. The main advantage of this approach is that semantic information about memory contents is available when saving and recovering checkpoint data. Using this approach, only the important data necessary to recover the application are saved. The main drawback is that the programmer has to manually insert CPR (checkpoint recovery) code to save and recover an application state, which is a very error prone process. Another drawback of this approach is the need to have access to the application source code.

### 2.3.6 User Level (ALC-U)

User-level checkpointing is implemented in user-space and typically provides transparency by virtualizing all system calls into the kernel. Within this virtualized environment the checkpointing approach is able to capture the state of the entire process without being tied to the kernel and without modifying the application code. It just inserts the checkpointing call in the code using a library that will be activated at each execution or by using a special process named pre-compiler or compiler (Bakhta and Ghalem, 2014).

### 2.3.7 Library Checkpointing

This technique provides support for checkpointing through a run-time library. The approach is not transparent to the user: the checkpoint contents and the places where checkpoints should be taken have to be defined by the application programmer. Its implementation is based on the LD_PRELOAD environment variable Sancho *at el.*,(2005) which installs the signal handlers and loads the checkpoint library without recompiling the application. It can be implemented also by the signal handler. A fail-safe PVM (Parallel Virtual Machine) proposed in Leon *et al.*,(1993) implements a checkpointing library on top of Unix to support fault tolerance (userlevel). It uses coordinated checkpointing activated by the daemon process to assure the coherence and replicates the checkpoint files in many nodes to ensure their availability.

DejaVu provides user-level checkpointing by implementing a new library in the system. DejaVu is a coordinated checkpointing system, but unlike Distributed Snapshots it uses a novel runtime mechanism called OLP (On Line Protocol) to capture the state of communication channels as part of the checkpoint and does not incur the overheads associated with flushing the network. The OLP is used to implement loosely

coordinated checkpointing. In contract to classic approaches of user-level checkpointing where the kernel state is recreated during the rollback of the system in the case of failure, the paper Tullmann *at al.*,(1996) proposes to encapsulate the kernel state and to store it in order to recreate the same kernel state in the rollback phase, so the overhead will be decreased. The paper Tullmann *at al.*,(1996) proposes also a new system where the processparent can control its children and ensures the checkpointing service for them. This reduces the size of checkpointing files.

Dieter *at al.*, (1999) describes a user-level checkpointing library to checkpoint multithreaded programs that use the POSIX threads library. It solves the problem of the inter-blockage of processes that can occur in the checkpointing process. MTCP introduced in Rieker *at al*., (2006) and Dieter *at al*., (2001) focuses also on the user-level checkpointing in multithreading systems and it uses coordinated checkpoints in shared memory systems. DTMC proposed in Ansel *at al*., (2009) extends MTCP by focusing on the management of sockets and it proposes to use a single started thread in the rollback process to minimize the checkpoint size. In the work Abdel Shafi *at al*., (1999), user-level library checkpointing is used for the migration of threads. In order to reduce the time of migration, the paper proposes to synchronize between the source and the destination before migration. It uses also the incremental checkpoint.

### 2.3.8 Pre-Compiler Checkpointing

To overcome the problem of transparency to the user, the pre-compiler checkpointing approach is introduced. The basic idea for a program transformation tool or pre-compiler is to analyse the application source code and determine what program variables must be saved at each checkpoint. It also adds the appropriate code to the

source code to write checkpoints and to restart the application from these checkpoints (Rodriguez *at al.*, 2010).

Compiler based approaches to checkpoint/restart fault tolerance are typically composed of two components: a pre-compiler, and a runtime support library. The pre-compiler is a source-to-source compiler that augments an existing application with calls to the associated runtime support library in order to provide transparent checkpoint/restart capabilities. This approach is independent of the MPI implementation. It permits the implementation of the coordination protocol without modifying the underlying MPI library.This promotes modularity and eliminates the need to access the MPI library code, which is proprietary on some systems. The additional requirement for the programmer is the need to insert calls to checkpointing functions at points in the application where checkpointing has to occur. Figure 2.3 shows the precompiler architecture.

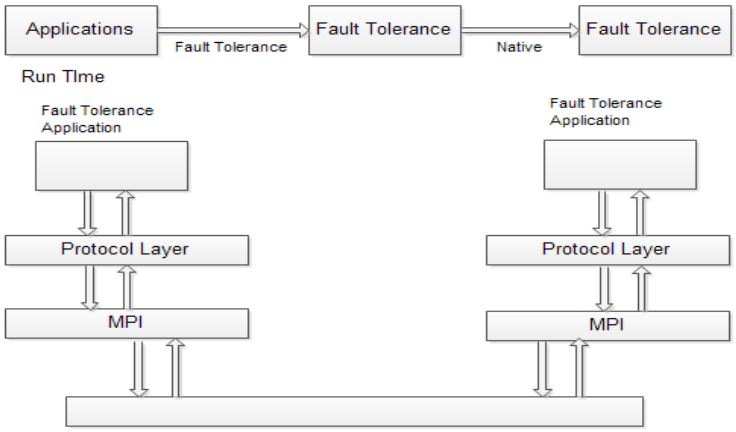


Figure 2.3: Pre-compiler Architecture (Bakhta and Ghalem, 2014)

Many compilers are proposed in the literature such as in Li *at al.*,(1994). In Rodriguez *at al.*,(2010), the authors propose a CPPC (Controller/Pre-compiler for Portable Checkpointing). It focuses on the automatic insertion of fault tolerance into long-running message-passing applications. It is designed to allow the execution restart

on different architectures or operating systems. CPPC supports checkpointing over heterogeneous systems, such as a grid. It uses portable code and protocols, and generates portable checkpoint files while avoiding traditional solutions (such as process coordination or message-logging) by using safe points.

Safe-point checkpoints are taken at the same relative code locations by all processes, without performing inter-process communication or runtime synchronization. To avoid problems caused by messages between processes, checkpoints must be inserted at points where it is guaranteed that there are neitherin-transit, nor orphan messages.

### 2.3.9 Mixed Level Checkpointing (MLC)

It is clear that neither application-level checkpointing nor system-level checkpointing is always an optimal solution to the system in terms of performance. Efficiency and correctness are difficult issues for both approaches. So, Mixed-Level Checkpointing (MLC) combines aspects of both application and system level checkpointing. It can be used to develop checkpointing systems that are able to omit all the problems of each type (Bakhta and Ghalem, 2014). The great difficulty in building MLC systems is separating the state of an application into logically consistent sets that can be checkpointed using either system or application level approaches. This approach was noted in Siva and Syam (2010) without any suggestion or implementation. However the paper Bronevetsky *at al*. (2003) implements MLC as a system that balances between SLC and ALC in order to get the best possible performance. But it does not resolve the problem of distinguishing between the states that must be stored with SLC and the states that must be stored with ALC. Since MLC in Bronevetsky *at al.*,(2003) balances between SLC and ALC so at any time the MLC can use only ALC so the programmer

will modify the application code to support the checkpointing. In this case even the MLC is not transparent. For this reason our comparison is limited to system-level checkpointing and application-level checkpointing.

## 2.4 SLC versus ALC Checkpointing: Problems and Solutions

Obviously, there are problems and advantages in the use of each approach. Try to quantify them using a list of metrics. And for each problem, indicates some solutions cited in the literature(Bakhta and Ghalem, 2014).

## 2.5 Programmer Effort (Transparency)

This property Bakhta and Ghalem, (2014) refers to how the user perceives the fault tolerance solution. In application-level checkpointing the programmers will have to specify what data should be included in the checkpoint, and where checkpoints should be taken within the application code. So the SLC is more transparent then the ALC. Even in the ALC there are different transparency degrees for each type (ALC-P, ALC-U (Lib), ALC-U (Comp)). This is illustrated in Figure 2.4.

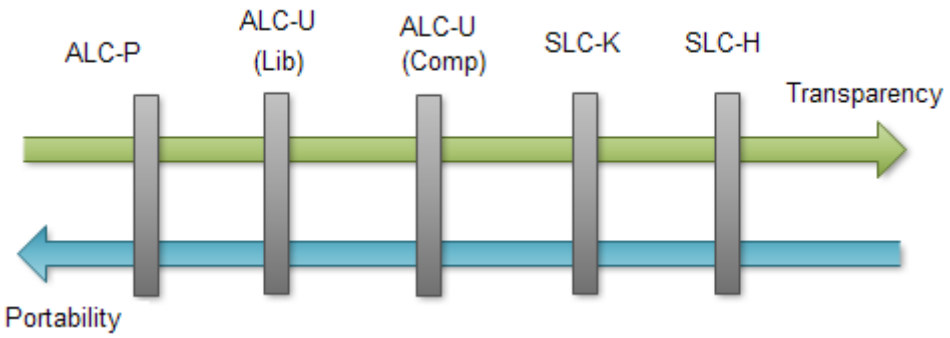


Figure 2.4: Transparency vs Portability in Abstraction Levels(Bakhta & Ghalem, 2014)

## 2.6 Portability

Another important aim is to provide portability of the checkpoint files and a portable checkpointing scheme. A checkpointing technique is portable if it allows the use of state

files to recover the state of a failed process on a different machine. This attribute reduces the transparency as shown in Figure 2.4. To solve the problem of portability at the system level and even to improve it at theapplication level some works use virtual machines to ensure the consistency of process states (Leon *at al.*, 1993), (Rieker *at al.*, 2006), (Hargrove and Duell, 2006). The second condition for a checkpointer to be portable is that all data is stored in a portable format so conversions can be carried out when necessary for recovering the process state on a binary incompatible machine. Since different architectures represent data types in different sizes, a technique to convert data from one architecture to another is needed. We can cite three strategies.

1. Machine independent presentation techniques: the classic example is XDR (External Data Representation) (Lyon, 1984). Some newer application level schemes have used an XML-based format as well Krishnan and Gannon(2004) or based on HDF5 (Hierarchical data forma 5) in CPPC (Rodriguez *at al.*, 2010).
2. Lowest/Highest precision in the group Ramkumar and Strumpen(1997): means to save all checkpoint data in a precision that is higher/lower than that of any of the machines within the group. Both of these techniques suffer from the disadvantage that a conversion is required, even if the checkpoint is being restored to the machine on which it was taken. In addition, the technique of saving in the lowest precision of the group requires knowledge of the group members before checkpointing takes place.
3. Receiver makes right technique Zhou and Geist(1995): the originator of the data simply checkpoints the data in its own precision. This technique has been used in (Jiang *at al.*, 2003). In this case, the receiver is charged to ensure that the data conversion takes place when necessary. However, data conversion issues arise in the case of architecture differences.

## 2.7 Checkpoint Size

The size of the full checkpoint file generated by SLC is larger than that generated by ALC since the SLC is based on a global snapshot of the processor address state, including all the dynamic data of the operating system. In the case of ALC, the programmer can precisely determine the data required for the recovery (Koch *at al.*, 2007) (Hargrove and Duell, 2006). In Silva and Silva(1998) the authors prove that the ALC can reduce the checkpoint's size by 50.7%.

However there are many techniques used to reduce this size of checkpoint file such as restricting the checkpointed data necessary for the rollback (Rieker *at al.*, 2006) (Osman *at al.*, 2002). Incremental checkpointing is also an efficacious strategy to decrease the checkpoint size. It identifies the dirty pages (those that have been modified since the last checkpoint): only these are saved to the checkpoint file. Many approaches use incremental checkpoints (Abdel-Shafi *at al.*, 1999). The authors in Gioiosa *at al.*,(2005) propose different strategies for incremental checkpointing based on dirty pages, such as Bookkeeping and Bookkeeping saving. The Word-level memory exclusion proposed in CAME Plank *at al.*,(1995) tracks every Read/Write operation so that both clean and dead memory can be excluded from the checkpoint file. This leads to near optimally small files. However the overhead of word-level memory exclusion is too large.

## 2.8 Flexibility

In particular cases, users may find it convenient to perform a data-driven or iteration-based checkpoint, rather than a blind time-triggered checkpoint. For the sake of flexibility and functionality, checkpoint/restart mechanisms should be accessible to the application programmer. This sort of ALC checkpoint can be used for other purposes rather than onlyforfault tolerance. For instance, ALC can be used to perform job swapping across different systems, for post-processing analysis or for data visualization.

To increase the flexibility in system-level checkpointing many techniques for determining the checkpointing interval are proposed in the literature (Siva and Syam, 2010). To improve the flexibility of ALC-U, many potential checkpoints are placed in the code and actual checkponts are chosen from amongst them according to the interval or the performance desired (Li and Yeung, 2008).

## 2.9 Efficiency

The efficiency is evaluated by the overhead added to the application's execution by the checkpointing system. In practice, there are two important sources of overhead. The first overhead type is incurred in order to maintain information about the execution of the application that would be used if a checkpoint were taken. This overhead, which is call the checkpoint-free overhead, is paid whether or not any checkpoints are taken during execution and should be kept as small as possible using techniques cited in Checkpoint size.

The checkpointing protocols used (coordinated, uncoordinated and communication induced) can also affect the checkpoint-free overhead. The technique that causes most overhead is the coordinated, followed by the communication induced, and then the uncoordinated. The last technique for minimizing the checkpoint-free overhead is used in the case of external checkpointing. This is when the checkpoint initiator does not participate in the application, as against internal checkpointing, where the checkpoint initiator is one of processes of the executing application(Gioiosa *at al.*, 2005). The second type of overhead is the cost of writing checkpoint data to stable storage whenever a checkpoint is taken, which is call the checkpointing cost. This overhead is proportional to the size of the checkpoint data. ALC techniques tend to incur a checkpoint-free overhead, whereas the SLC techniques generally do not. The

checkpointing cost can be reduced by reducing the amount of data in each checkpoint. It can be also reduced by controlling the I/O system (Wang *at al.*, 2014). In BLCR proposed in Hargrove and Duell(2006) the I/O manger collects the writes in a buffer in order to minimize the I/O calls, this strategy reduces the overhead cost by 12%. Storage type described in Prvulovic *at al.*,(2002) infects the overhead cost: it groups schemes into three classes based on where and how the checkpoint storage is protected from errors.

The first type is Safe External Storage where the checkpoint is stored in external storage that is assumed to be safe. The second storage type is Safe Internal Storage where the checkpoint is stored in main memory or other internal storage and made safe through redundancy across the nodes. And the last storage type is Specialized Fault Class where the checkpoint storage is not protected with redundancy across nodes. However, the system is not expected to recover from faults that can damage that storage. It is clear that the first type of storage increases the overhead compared with the other two types. There is another technique used to decrease the checkpoint cost called "Copy on write" where the checkpointing write is executed in the background of the application execution (the checkpoint service elect a process to do the checkpointing write and the application continue the execution at the same time (Rieker *at al.,* 2006).

### 2.10Restart Ability

Operating systems like UNIX provide a virtual and uniform memory layout in homogeneous machines, making it easier to restart a process on a different processor. However, there are some state attributes that are kernel dependent. They cannot be saved and carried across different processors in a sensible fashion (Abdel-Shafi *at al.*, 1999). To assure the restart ability of the checkpointing, the application program should

not make use of kernel dependent attributes. The restart ability is related to the portability so ALC is more easily portable and can be restarted on different systems. SLC is usually restricted to homogeneous hosts. To facilitate this restart ability, the Zap Osman *at al.*,(2002) uses the domain file as a virtualization that ensures the portability of the process and its resources. In Tullmann (1996) the kernel state is encapsulated in a list of references and information to ensure the restart ability. For the restart ability in case of migration, the work proposed in Abdel-Shafi *at al.*,(1999) proposes synchronization between the source and the destination of the thread; and both the source and the destination store the thread state at the virtual address.

### 2.11Forced Checkpointing Generation

ALC cannot be generated forcefully, because in application level checkpointing, the process state can only be saved when checkpoint generation code is reached during execution, whereas in SLC, it can be saved at any moment, since the state is obtained directly from the main memory by a separate thread or process. The technique of potential checkpoints insertion in ALC cannot make the forced checkpointing possible but it reduces the time between the desired time of forced checkpoints call and the execution of the real checkpointing (Bakhta and Ghalem, 2014).

### 2.12Correctness

Correctness is the ability of a checkpointing system to ensure that the application produces a correct result. Of all of the properties that have listed here, correctness is arguably the most difficult to ensure. To solve the problem of correctness in SLC and ALC, the Mixed checkpointing level MLC is proposed in (Bronevetsky *at al.*, 2003).

The paper (Li and Yeung, 2008) uses the correctness at the application level to reduce the checkpoint's size. This paper investigates definitions of program correctness that

view correctness from the application's standpoint rather than the architecture's standpoint. Under application- level correctness, a program's execution is deemed correct as long as the result it produces is acceptable to the user. So in case of soft computing (programs that produce inexact and/or approximate outputs) it is not necessary to store some states, called soft states, because they will be modified at every re-execution without touching the correctness of the application.

### 2.13Comparison between ALC and SLC

To summarize the characteristics of both ALC and SLC using all the points cited in the previous section we present Table 2.1,whichcompares the differences between application and system level checkpointing.

Table 2.1: ALC versus SLC(Bakhta and Ghalem, 2014)

| Level | SLC | ALC |
|---|---|---|
| Transparency | Yes | No |
| Portability | No | Yes |
| CP Size | High | Low |
| Correctness | No | No |
| Flexibility | No | Yes |
| Forced CP | Yes | No |
| Efficiency | No | Yes |
| Restart-ability | Difficult | Easy |
| Scalability | No | Yes |

The system signal is a general purpose signal provided by the system but called only by the user to perform checkpointing (automatic initiation) which minimizes the transparency and the flexibility. A kernel-mode signal handler is based on the signalling mechanism offered by the kernel, it is a new specific signal added to the kernel for the purpose of checkpointing. The advantage of using kernel mode is that the checkpoint is initiated by the system; but as the previous signal (system call), it is hard to control the checkpointing so the flexibility is not assured. Applications may be flexibly

checkpointed by adding a specific thread to the application's process called a kernel thread. Here a kernel thread is created to perform the checkpoint/restart activities so that flexibility is assured (Bakhta and Ghalem, 2014).

Since the kernel thread is also a different process and, especially in a multiprocessor system, it might run in parallel with the application that can change some data while the kernel thread is saving them. In this case a mechanism to stop the application is necessary in order to guarantee data consistency.

The System Call and the Kernel Mode signal handler approaches have the advantages of being executed behind the process that has to be checkpointed. In this way the actual process address space is still the same as the process running in user mode. Unlike the system and kernel signal that required a modification in the code, the kernel thread is completely transparent.

### 2.14 Related Work

Malarvizhi and Uthariaraj(2010)identified two major problems that are critical to the effective utilization of computational resources;those are efficient scheduling of jobs and providing fault tolerance in a reliable manner. This paper addresses these problems by combining the checkpoint replication based fault tolerance mechanisms with Minimum Total Time to Release (MTTR) job scheduling algorithm. Total Time to Response (TTR ) includes the service time of the job, waiting time in the queue, transfer of input and output data to and from the resource.

Amoon (2011)proposed an architecture that employs job replication as an effective approach for achieving efficient fault-tolerant and scheduling system. Most of work done using the replication-based approach assumes using a fixed number of replications

for each job which consumes more grid resources. He propose an algorithm to determine adaptively the number of job replicas according to the grid failure history.

Gokuldev and Valarmathi (2013) used a separate server for storage purpose in order to increase the execution time. The main goal of checkpoint approach is to minimize the overall execution time in grid system. In this work fault tolerant scheduling is achieved using kernel-level checkpoints. In case of resource failure, the Fault Index Based Rescheduling (FIBR) algorithm is used to reschedule the jobs to some other available resources. This ensures that the job is executed with minimized execution time.

Gokuldev & Radhakrishnan(2014) proposed adaptive job scheduling in addition to the fault tolerance strategy with checkpointing approach which were able to show the improvement in performance of the overall computation time even in worst scenario under the heterogeneous grid environment. In this research work, there will be an overhead in the check points because the authors did not consider the size and the number of checkpoints in their implementetion.

Idris(2015) Proposed a Job Scheduling Algorithm in Grid Computing Environment Using Fault Tolerance Mechanisms. The technique employed here, is the use of resource failure rate, as well as checkpoint-based roll back recovery strategy. Checkpointing aims at reducing the amount of work that is lost upon failure of the system by intermediately saving the state of the system.

## 2.15 Gap in the literature

A large amount of research has been done on fault tolerance in the area of distributed computing. However, little work has been done on fault tolerance in grid environments that take into cognisance the checkpoint interval.

Replication is based on the assumption that the probability of a single resource failure is much higher than of a simultaneous failure of multiple resources. It avoids job recomputation by starting several copies of the same job on different resources. With redundant copies of a job, the grid can continue to provide a service in spite of failure of some grid resources carrying out job copies without affecting the performance. But then it requires huge amount of momery to do the replication and again in the process of replication the execution of user's job may be affected and henced the performance will suffer.

Checkpointing is the ability to save the state of a running job to secondary storage so that it can resume its execution later than the time at which it was last stored. The purpose of checkpointing is to increase fault-tolerance and to speedup application execution on unreliable systems. The major chellenge in checkpointing is the creteria for determining the number of checkpoints and in what interval should it be invoked. Some researchers used either response time or failure rate or Fault Index or a combination of any two to determine the number of checkpoints and their size.

# CHAPTER THREE

# MODIFIED ARCHECTURE OF FAULT TOLERANCE IN GRID COMPUTING

## 3.1 Introduction

This chapter shows the enhancement of the existing dissertation and the elaboration on the method to be applied to achieve the proposed checkpointing of the scheduling algorithms.

## 3.2 Architecture of the Proposed Checkpointing Technique

The architecture of the proposed improved checkpointing system is shown in Figure 3.1. The main difference between this architecture and that of Idris (2015) is that Replica module has been added to reduce checkpoint overhead and to enhance reliability.

When a user submits a job to the Scheduler Handler, it interacts with the GIS to get information about available resources of the grid and then requests the resources to provide their information about the most current work load condition as well the fault manager to send its fault rate history. The Checkpoint Handler will determine the checkpoint interval based on the failure rate, response time and tendency of failure of the failed resource. At every successive checkpoint the Replica is responsible for replicating the states of the executing machines to the other machines of the grid to ensure availability in case it does not come to life again. If failure occurs, the scheduler collects the set of appropriate machines along with their work load and fault rate and it invokes the ACO Scheduling techniques for decision making. The ACO algorithm is responsible for searching the resource that meets the user requirement using the fault rate of the resource and the work load on the resource before appropriate decision

making. The criteria for selecting a resource are based on the current work load and fault rate of the resources. The Grid Dispatcher sends the jobs to the Checkpoint Handler before forwarding to the job of the selected resource. When user submits the job to the Grid Scheduler Handler, the Checkpoint Handler contacts the Fault Manager to get the number of successful jobs completed and number of failed jobs of the resource on which it is scheduled and sets the number of time to the checkpoint and the checkpoint interval based on the failure rate of the resource. If the failure rate of a resource is higher than the average number of failure rate of resources, there is high tendency that the resource may not be considered for scheduling except that the work load of the all other ones doubled that of the failed one. Then the Checkpoint Handler submits the job along with the checkpoints to the selected resource. The Checkpoint Handler provides the checkpoint interval to the Checkpoint Repository alongside the uncompleted executed result of the jobs of the grid resources which is available in the checkpoint table. For a particular job, the Checkpoint Repository purges the replicated state of the previous checkpoint from all grids and considers the last checkpoint for recovery. The failure of a resource is reported to the Checkpoint Handler to increment its fault index. Otherwise, if a particular job is completed, it will be removed from checkpoint result table. The Fault Handler tracks the status of the resources available in the grid at regular intervals.

The Fault Index value gives the rate of resource failure, whereby, the higher the fault index value, the higher the failure rate of the resource. The Checkpoint Handler then queries the Checkpoint Repository to obtain latest checkpoint files of the executed jobs on the failed resource, if available, or get it from the replicated checkpoint files and rescheduled the jobs along with last checkpoint status. On the successful completion of the job, the checkpoint files are removed to save space and the Checkpoint Handler

receives the job completion message from the grid resource and updates the Fault Index

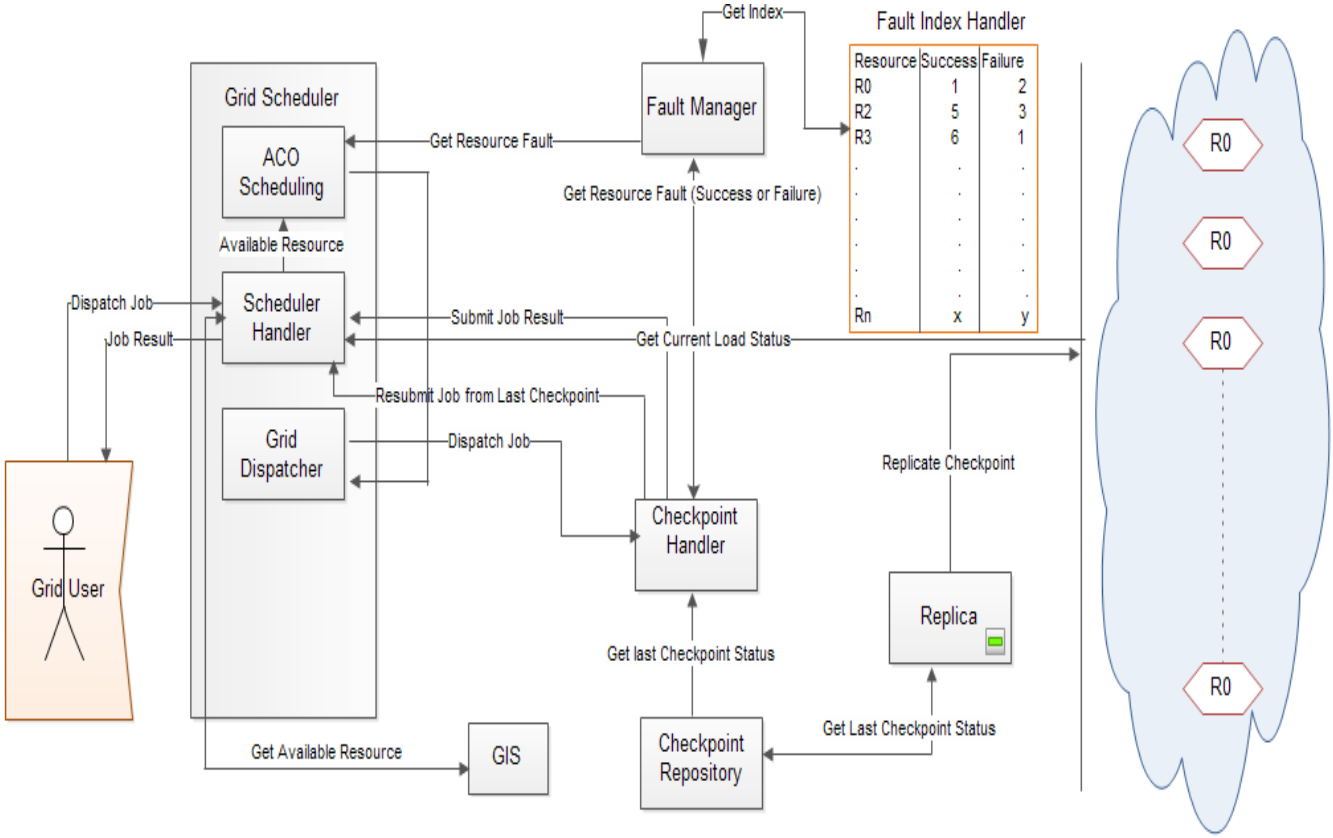


Figure 3.1: Modified Architecture of the Proposed Fault Tolerance in Grid Computing

handler to increment the success rate of the resource. The Fault Index Handler maintains a Fault Index history of the grid resources, which indicates the failure rate of the resource. The decision to be taken by Fault Index Handler on the update and maintenance of the Fault Index of a grid resource is to be determined by the algorithm in Figure 3.2.

- IF the checkpoint handler receives the job completion message from resource THEN
  - Send a message to the fault index handler to increment success index of the resource.
  - Submit finished job to the scheduler.

END IF

- IF the checkpoint handler receives the failure message from the resource monitor THEN
  - Send a message to the fault index handler to increment the failure index of the resource that failed to complete the assigned job.
  - Send a message to the checkpoint repository, to check checkpoint status of this job.
  - Replicate the checkpointing status to the other nodes (To ensure availability)
- IF the checkpoint result of the job exists in the checkpoint repository

THEN

  - Submit the last checkpoint data received, to the scheduler handler for rescheduling.
  - Exit

END IF

- IF the checkpoint result of the job does not exist in the checkpoint repository THEN
  - Get the replicated checkpoint data from any available node.

Figure 3.2: Modified Algorithm of the Fault Index

The Checkpointing algorithm implemented by Idris (2015)as shown in Figure 3.3 assumed static values for the checkpointing time and the interval between the checkpoints while in our proposed checkpointing algorithm they are to be determined based on the failure rate and tendency of failure of a resource as shown in Figure3.4

S1: set checkpoint time and interval for a particular job

S2: submit the job for execution

While job is running till the end

S3: if current Time ≥ checkpoint Time

Then
Update checkpoint interval time
Increment number of time checkpoint
Checkpoint current process state
Else not time for checkpoint

Figure 3.3: Existing Checkpoint Algorithm

```
S0: determine the failure rate of the failed resource
S1: determine the fail tendency of the resource
S2: determine the average number of all failed resources
    IF (S1>S2) THEN
        S3: set checkpoint time and interval for a particular job to be high
    ELSE IF (S0>=S2) THEN // high failure rate
        S3: set checkpoint time and interval for a particular job to be high
S4: submit the job for execution
While job is running till the end
    S5: if current Time ≥ checkpoint Time
        Then
            Update checkpoint interval time
            Increment number of time checkpoint
            Checkpoint current process state
            Replicate the checkpoint state
    Else not time for checkpoint
```

Figure 3.4: Modified Checkpoint Algorithm

### 3.3 Checkpointing Control Implementation

The checkpointing is one of the most popular and widely used techniques to provide fault-tolerance on unreliable systems. It is a record of the snapshot of the entire system state in order to restart the application after the occurrence of some failure. The states and data of a running process include registers containing the address, variables, memory spaces keeping source codes, libraries, data structures, files containing data with a large size.

A programming-level checkpointing control is adopted. The program source code saves and restores the critical section of the program variables to and from stable storage. The object-oriented program is considered in the implementation whereby the state of the program is to be recovered from the contents of the objects' fields.

To identify what is to be checkpointed, the object of the class Gridlet Checkpointing must be created and then visited and recorded the state of a resource. The method checkpoint Repository() is explicitly invoked by the programmer where it's required and when it is time to take a checkpoint, Gridlet Checkpointing object drives the checkpointing process whereby replicateCheckpoint() method replicates the checkpoint status to all other nodes to ensure availability.

The hash function technique used in the checkpoint is the same with that of (Idris, 2015). A hash function takes a group of characters (called a key) and maps it to a value of a certain length (called a hash value or hash). The hash value is representative of the original string of characters, but is normally smaller than the original. Hashing is a technique of storing values and searching for them in tables. Table is a set of table entries, (K; V), each containing a unique key K, and a value (information) V. The assumption made is that checkpoints are written to this hash table. Each checkpoint contains only the pages that have been modified since the previous checkpoint. Here, Hash table keyed is used to keep track of the necessary objects. It used a hash table keyed on identity hash code of the objects. It used a hash table keyed on identity hash code of the objects. This hash table maps from the object to a pair: <object, address>. Object is a reference back to the object; address refers to the address of the object in the output image.

The resource failure rate (*FR*), and tendency of failure are used to determine the checkpoint interval and the number of checkpoints instead of using the resource fault index. Resource failure rate is used to represent failure history of a resource. The formulae used to compute the number of checkpoints and checkpoint interval as given next. Equation (3.1) calculates the fault rates of the resources and Equation (3.2) calculates the number of times to checkpoint a particular job when it is running and

Equation (3.3) calculates the checkpoint interval time when a job should be checkpointed.

$$Failure\ Rate(FR) = \frac{Failed\ \ Job}{Number\ \ of\ Jobs\ \ Submitted} \quad . \quad . \quad . \quad (3.1)$$

Suppose $R_t$ is the response time then the number of checkpoints can be determined by

$$CheckpointNumber\ = R_t * FR. \quad . \quad . \quad (3.2)$$

The interval between one checkpoint and the other (checkpoint interval) is given by

$$checkpointInterval = \frac{R_t}{checkpointNumber} \quad . \quad . \quad . \quad (3.3)$$

$$Failtendency(\mathrm{FT}) = \frac{\sum_{j=1}^{m} p_{fj}}{m} \quad . \quad . \quad . \quad (3.4)$$

Where m is the total number of grid resources and $p_{fj}$ is the failure rate of resource *j*. Through this metric, it is expectedto determine the faulty behaviour of the system(Altameem, 2013).

### 3.4 Job Rollback Recovery System Analysis

Checkpointing is the ability to save the state of a running job to a stable storage. In case of any fault, this saved state can be used to recover the lost computation of the user's job from the point where the checkpoint was last captured and saved instead of restarting the execution from scratch thereby reducing the execution time to a large extent, potentially. This technique is known as rollback recovery. The main purpose of checkpointing is to increase fault-tolerance, improve throughput, lower average turnaround time and Makespan. The checkpoint mechanism periodically saves current and removes the previous state of executing application to and from stable storage respectively. Hence, whenever a failure occurs, the job execution can be resumed from its most recent successful state, thereby improving the QoS requirement of the users. In

this work, the checkpoint recovery state is replicated to all grid sites despite being already stored in the checkpoint repository system, in case of failure. This ensures reliability and availability of the saved checkpointing files. The checkpoint Replica gets information about failure rate of each resource and decides where to recover the saved checkpointing files based on the low failure rate history from Fault Manager and then passes it to Checkpoint Handler where it will be submitted to Grid Scheduler for scheduling.

### 3.5 System Model

Grid environments are heterogeneous and dynamic, with components joining and leaving the system all the time. Faults are likely to occur in grid environments. Also, the likelihood of error occurrences is exacerbated by the fact that many grid applications will perform long tasks that may require long time of computation. This will allow us to make efficient comparison of the proposed scheduling algorithm with the algorithm proposed by Idris (2015) and ensure reliability and availability in order to meet the user's Quality of Service (QoS) in existence of resource faults. The comparison is based on the assumption that the grid consists of one computing node (machine) per resource and 2 Processing Elements (PE) as well as the Network speed among the computing nodes is also the same as used by Idris (2015) as shown in Table 3.1

Table 3.1: Grid Resource Characteristics

| Grid Resource Characteristics | |
|---|---|
| Number of machines per resource | 1 |
| Number of PE per machine | 2 |
| PE ratings | 50 MIPS |
| Bandwidth | 5000 B/S |

### 3.6 Application Model

The user submits the job with its requirements (deadline, estimated execution time, etc) to the grid without knowing how the computations will be done because of the heterogeneous nature of grid. During simulation, different numbers of Gridlet objects are created to evaluate these approaches. A Gridlet is defined in terms of length (in Million Instruction), input file size (in byte), output file size, and computational size. The same characteristics of the Gridlet used in the simulation as used by (Idris, 2015) sent to the grid to compare the performance of different algorithms as shown in Table 3.2

Table 3.2: Gridlet Characteristics

:

| | |
|---|---|
| Length | 0 – 50000 MI |
| Input file size | 100 + (10% to 40% ) |
| Output file size | 250 + (10% to 50% ) |

### 3.7 Performance Evaluation Criteria

The criteria for evaluation of performance of the proposed algorithm include makespan, average turnaround time and throughput. Thegridlets and topologies are randomly generated, number of checkpoints and checkpoint interval are dynamically determined,

and the replicated states of the failed resources are to be recovered. The batch file in order to simulate and compare the results were used to plot graphs

$$Makespan\ (MS) = FTL - STF\ .\quad .\quad .\quad (3.4)$$

where $FTL$ is thefinish time of the last task and $STF$ is thestart time of the first task

$$Throughput(TP) = \frac{NGRD}{TGRD}\quad .\quad .\quad .\quad (3.5)$$

where $NGRD$ is thefinish time of the last job and $TGRD$ is thestart time of the first job

$$Turn\ Around\ Time(TAT) = GRDET - GRDST\ .\quad .\quad .\quad (3.6)$$

where$GRDET\ is\ thegridlet\ Execution\ time$ and$GRDST$is the$gridlet\ start\ time$

$$Average\ Turn\ Around\ Time(ATAT) = \frac{\sum_{i=0}^{n} TAT_i}{n}, \text{n } = \text{ NGRD}\ .\quad .\quad .\quad (3.7)$$

Suppose that when a job submitted and starts its execution at time *JBST* and finishes its execution at time *JBET*, failure occurs at time *JBF* (Job failure notification time) and the last created checkpoint is at time *JBCP* which is before *JBET*, then the job can be restarted from the last checkpoint at time *JBRT*(Job restart time from last checkpoint)

$$Total\ Execution\ Time\ (TET) = (JBF - JBST) + (JBF - JBRT) + (JBCP - JBRT).\quad .\ .\ (3.9)$$

$$Execution\ Time\ (ET) = (JBET - JBST)\ .\quad .\quad .\quad (3.8)$$

# CHAPTER FOUR

# RESULT ANALYSIS

## 4.1 Introduction

The comparison of the proposed checkpoint scheduling algorithm and the scheduling algorithm implemented by Idris (2015) were performed under the same heterogeneous grid environment and the simulation was done using GridSim toolkit to evaluate the improvement of the proposed approach. Results of the experiments are analyzed based on the standard performance metrics (makespan, throughput, and turnaround time) and the reliability of the resources.

## 4.2 Results and Discussion

In the simulation, the scheduling experiments were performed by keeping the resource constant and setting different values to the number of jobs; the number of Gridlets was varied from 1,00 to 3,700 and the length of each gridlet was 200,000 MIPs(Million Instructions Per Second) at each step, as similarly implemented by Idris (2015) for comparison analysis.

In order to evaluate and compare the performance of the proposed checkpoint algorithm with Idris (2015), throughput is one of the most important standard metrics that is used to measure the performance of fault tolerant systems as mathematically defined as:

$$Throughput(T) = \frac{N_j}{T_n}. \quad . \quad . \quad (4.1)$$

where$N_j$ is the number of jobs submitted to grid by user and $T_n$ is the total amount of time it requires to complete the execution of n jobs.

Table 4:1: Average Throughput table for varied Gridlets

| Number of Resource | Number of Gridlet | Existing Throughput(ms) | Improved Throughput(ms) | % Improvement |
|---|---|---|---|---|
| **100** | 100 | 0.001296 | 0.001322 | 2.00% |
| **100** | 500 | 0.00322 | 0.003381 | 5.00% |
| **100** | 900 | 0.003909 | 0.004104 | 5.00% |
| **100** | 1300 | 0.004431 | 0.004697 | 6.00% |
| **100** | 1700 | 0.004955 | 0.005253 | 6.00% |
| **100** | 2100 | 0.005526 | 0.005912 | 7.00% |
| **100** | 2500 | 0.005947 | 0.006363 | 7.00% |
| **100** | 2900 | 0.006234 | 0.006671 | 7.00% |
| **100** | 3300 | 0.006553 | 0.007077 | 8.00% |
| **100** | 3700 | 0.006762 | 0.00737 | 9.00% |

Throughput measures the number of batch jobs completed in a certain period of time, Increase in throughput in the proposed checkpoint algorithm suggested the improvement as compared with checkpoint algorithm implemented by Idris (2015) by about 9%. Table 4.1 shows the improvement achieved based on throughput. The tendency of failure observed during simulation shows that the number of jobs failure increases as the number of Gridlets increased. This is because resources have more jobs in execution at the time of failure. Consequently, improvement in throughput by our proposed system increases as the number of gridlets increases.

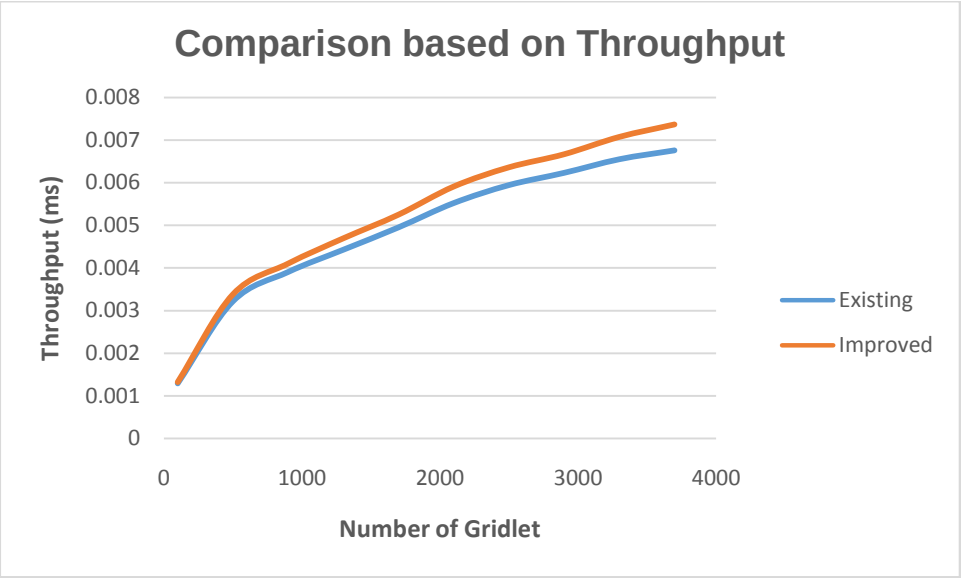


Figure 4.1 Graph of Average Throughput for varied Gridlets

Figure 4.1 shows the aggregate comparative performance on throughput between the two systems over varying numbers of gridlets. Note, however, that associated with this throughput improvement is the increase in overhead of checkpoint management in our proposed model.

Table 4.2: Average Makespan for varied Gridlets

| Number of Resource | Number of Gridlet | Existing Make span | Improved Make span | % Improvement |
|---|---|---|---|---|
| 100 | 100 | 82581.19 | 78452.56 | 5.00% |
| 100 | 500 | 159281.1 | 148131.42 | 7.00% |
| 100 | 900 | 235238.1 | 218771.46 | 7.00% |
| 100 | 1300 | 298183.8 | 274329.11 | 8.00% |
| 100 | 1700 | 347092.4 | 315854.06 | 9.00% |
| 100 | 2100 | 383579.5 | 352893.12 | 8.00% |
| 100 | 2500 | 423205.2 | 385116.73 | 9.00% |
| 100 | 2900 | 467180.2 | 420462.18 | 10.00% |
| 100 | 3300 | 505965 | 470547.41 | 7.00% |
| 100 | 3700 | 548524.3 | 488186.66 | 11.00% |

Results of the simulation as shown in the Table 4.2 below and Figure 4.2 indicated that there is decrease in makes pan as compared with (Idirs, 2015) which indicated an

improvement for about 11%. This is simply because the checkpoint is not frequently invoked that can bring an overhead which increases the job execution time.

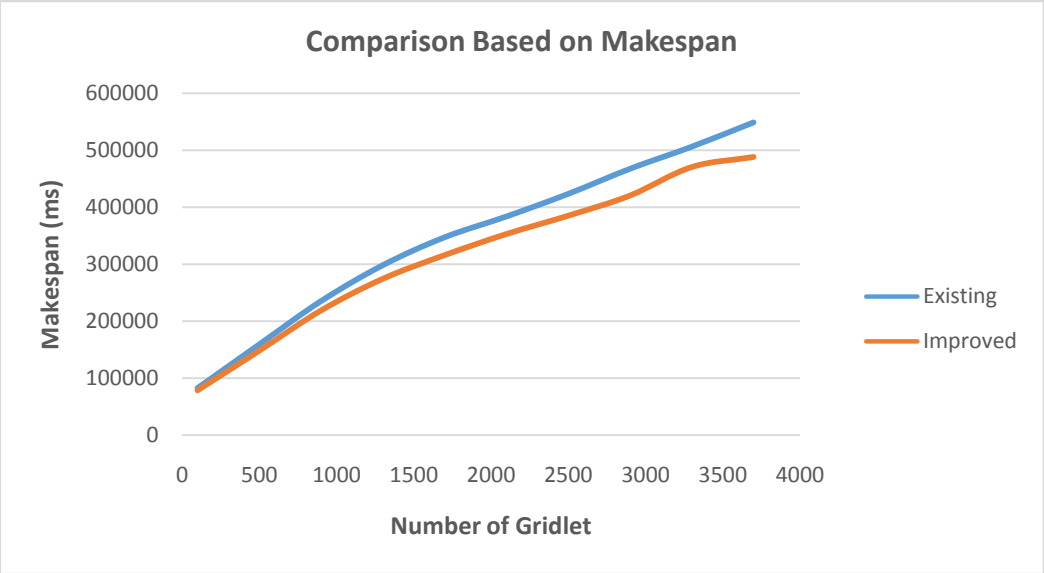


Figure 4.2: Graph of average Makespan for varied Gridlets

Figure 4.2 shows the aggregate comparative performance on makespan between the two systems over varying numbers of gridlets.

Table 4.3: Average Turnaround time for varied Gridlets

| Number of Resource | Number of Gridlet | Existing ATAT | Improved ATAT | % Improvement |
|---|---|---|---|---|
| 100 | 100 | 32851.02 | 31865.49 | 3.00% |
| 100 | 500 | 79757.59 | 75769.71 | 5.00% |
| 100 | 900 | 134389.9 | 126326.51 | 6.00% |
| 100 | 1300 | 186865.8 | 171916.58 | 8.00% |
| 100 | 1700 | 228997.5 | 215257.66 | 6.00% |
| 100 | 2100 | 262358.6 | 241369.92 | 8.00% |
| 100 | 2500 | 298553.1 | 271683.33 | 9.00% |
| 100 | 2900 | 339709.5 | 315929.85 | 7.00% |
| 100 | 3300 | 383153.9 | 341007.01 | 11.00% |
| 100 | 3700 | 427488.4 | 384739.57 | 10.00% |

Size of the Gridlets improved the average turnaround time by 11% as shown in Table 4.3. The improvement was a result of infrequent invocations of checkpoint and the selection of resources by considering the calculated low failure tendency.

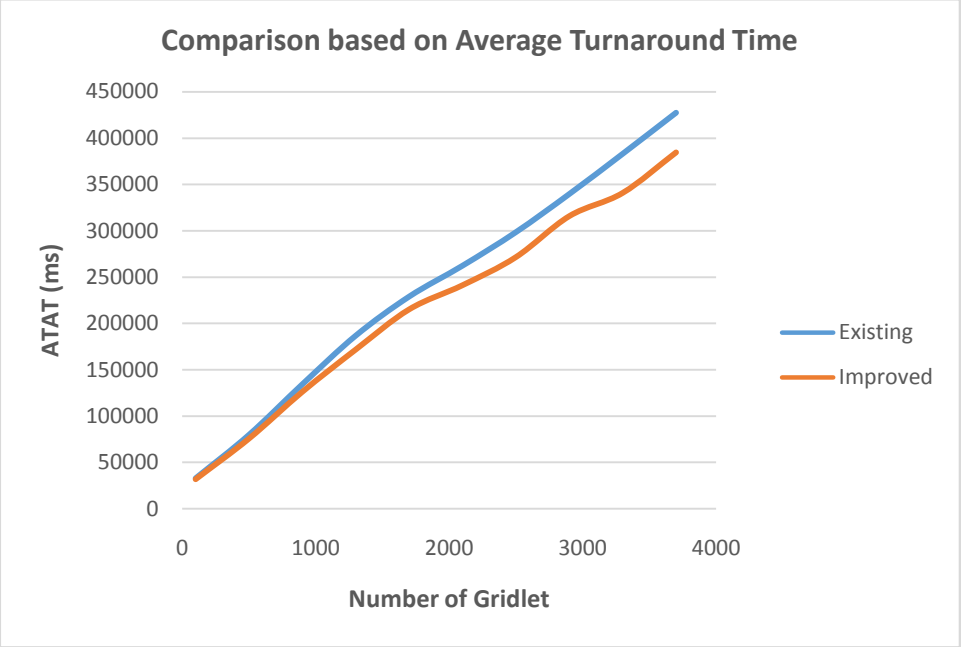


*Figure 4.3 Graph of Average Turnaround Time for varied Gridlets*

Figure 4.3 shows the aggregate comparative performance on average turnaround time (ATAT) between the two systems over varying numbers of gridlets.

Now keeping the number of Gridlets constant and varying the number of resources, the results show that there were improvements in terms of throughput, make span and average turnaround time by up to 8%, up to 11% and up to 9% as shown in Tables 4.5, 4.6 and 4.7 respectively.

Table 4.4: Average Throughput time for varied Resources

| Number of Gridlets | Number of Resources | Existing Throughput | Improved Throughput | % Improvement |
|---|---|---|---|---|
| 3000 | 50 | 0.000723 | 0.0007594 | 5.01% |
| 3000 | 250 | 0.003297 | 0.0034573 | 4.88% |
| 3000 | 450 | 0.00477 | 0.0050561 | 6.00% |
| 3000 | 650 | 0.005824 | 0.0061155 | 5.00% |
| 3000 | 850 | 0.006571 | 0.0069653 | 6.00% |
| 3000 | 1050 | 0.007189 | 0.0075485 | 5.00% |
| 3000 | 1250 | 0.007919 | 0.0083937 | 6.00% |
| 3000 | 1450 | 0.008372 | 0.00879 | 5.00% |
| 3000 | 1650 | 0.008859 | 0.0095678 | 8.00% |
| 3000 | 1850 | 0.009327 | 0.0099796 | 7.00% |
| 3000 | 2050 | 0.009683 | 0.0104571 | 8.00% |
| 3000 | 2250 | 0.009845 | 0.0102387 | 4.00% |
| 3000 | 2450 | 0.010471 | 0.0108894 | 4.00% |
| 3000 | 2650 | 0.010837 | 0.0114867 | 6.00% |
| 3000 | 2850 | 0.010826 | 0.0111512 | 3.00% |
| 3000 | 3050 | 0.011195 | 0.0120915 | 8.01% |

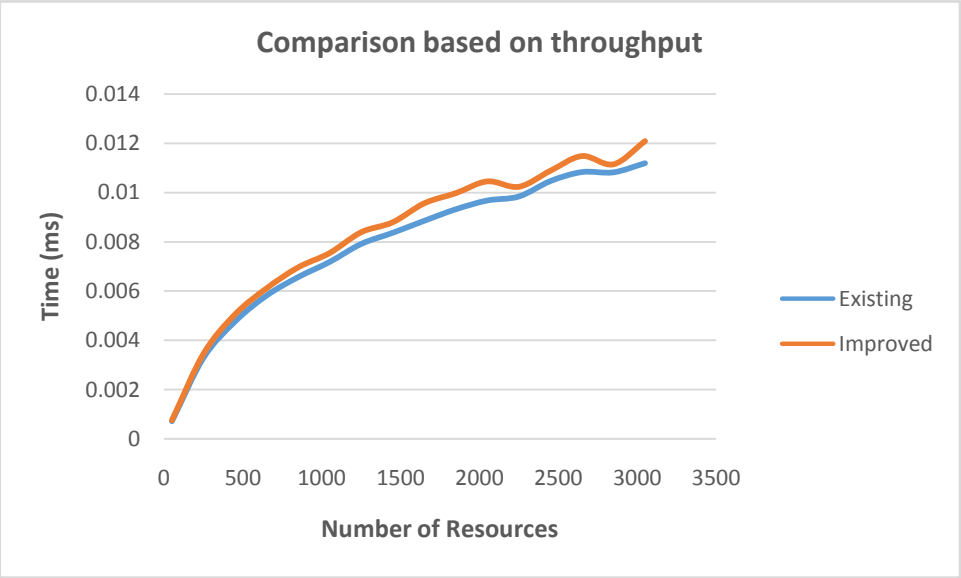


Figure 4.4 Graph Throughput for varied Gridlets

Figure 4.4 shows the aggregate comparative performance on throughput between the two systems over varying numbers of resources.

Table 4.5: Average Make span time for varied Resources

| Number of Gridlets | Number of Resources | Existing Makespan | Improved Makespan | % Improvement |
|---|---|---|---|---|
| 3000 | 50 | 4193551.12 | 3858067.03 | 8.00% |
| 3000 | 250 | 915672.954 | 833262.39 | 9.00% |
| 3000 | 450 | 635026.91 | 584224.76 | 8.00% |
| 3000 | 650 | 522718.722 | 486128.41 | 7.00% |
| 3000 | 850 | 462654.445 | 430268.63 | 7.00% |
| 3000 | 1050 | 425486.675 | 391447.74 | 8.00% |
| 3000 | 1250 | 384175.14 | 349599.38 | 9.00% |
| 3000 | 1450 | 366323.478 | 337017.61 | 8.00% |
| 3000 | 1650 | 324779.388 | 292301.45 | 10.00% |
| 3000 | 1850 | 330142.321 | 300429.51 | 9.00% |
| 3000 | 2050 | 317569.953 | 292164.36 | 8.00% |
| 3000 | 2250 | 311514.311 | 293478.02 | 5.79% |
| 3000 | 2450 | 293752.861 | 264377.57 | 10.00% |
| 3000 | 2650 | 282355.198 | 251296.13 | 11.00% |
| 3000 | 2850 | 284203.798 | 261467.51 | 8.00% |
| 3000 | 3050 | 272435.062 | 247915.92 | 9.00% |

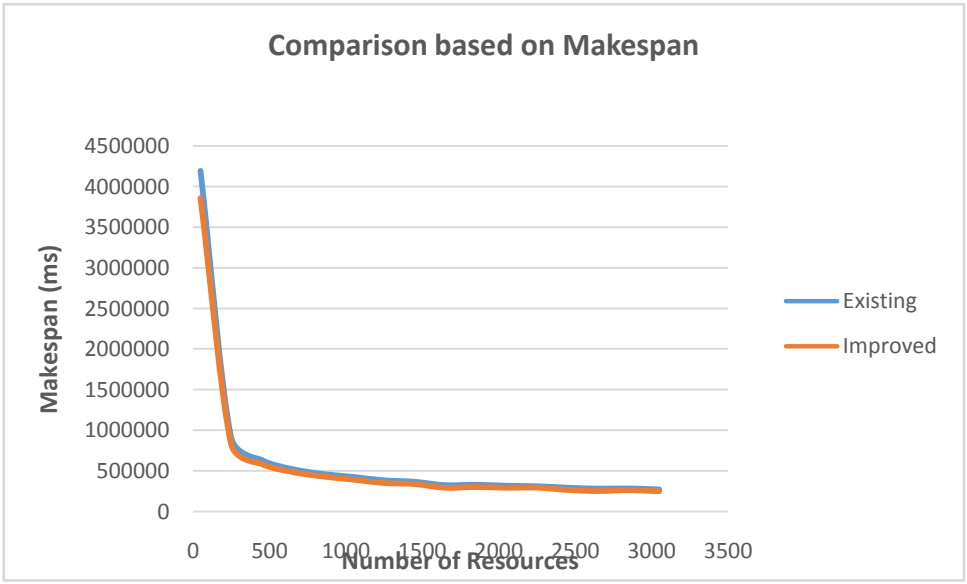


Figure 4.5: Graph of Average Makespan time for varied Number of Resources

Figure 4.5 shows the aggregate comparative performance on makespan between the two systems over varying numbers of resources.

Table 4.6: Average Turnaround time for varied Resources

| Number of Gridlets | Number of Resources | Existing ATAT | Improved ATAT | % Improvement |
|---|---|---|---|---|
| 3000 | 50 | 3026939.18 | 2784784.04 | 8.00% |
| 3000 | 250 | 569764.08 | 535578.23 | 6.00% |
| 3000 | 450 | 324526.24 | 301809.41 | 7.00% |
| 3000 | 650 | 232252.33 | 213672.15 | 8.00% |
| 3000 | 850 | 186910.56 | 170088.62 | 9.00% |
| 3000 | 1050 | 159492.9 | 148328.39 | 7.00% |
| 3000 | 1250 | 141939.98 | 140520.58 | 1.00% |
| 3000 | 1450 | 130883.26 | 120412.6 | 8.00% |
| 3000 | 1650 | 121695.2 | 111959.59 | 8.00% |
| 3000 | 1850 | 116649.38 | 106150.93 | 9.00% |
| 3000 | 2050 | 112504.84 | 103504.45 | 8.00% |
| 3000 | 2250 | 109215.73 | 100478.47 | 8.00% |
| 3000 | 2450 | 107411.06 | 100966.4 | 6.00% |
| 3000 | 2650 | 104947.09 | 99699.74 | 5.00% |
| 3000 | 2850 | 103099.57 | 95882.5 | 7.00% |
| 3000 | 3050 | 101897.4 | 93745.61 | 8.00% |

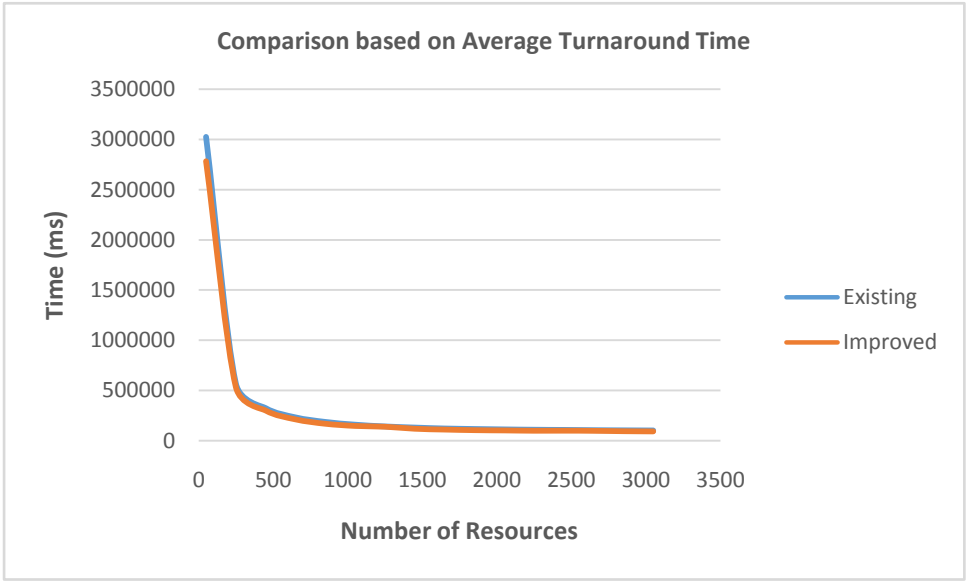


Figure 4.6: Graph of Average Turnaround time for varied Number of resources

Figure 4.6 shows the aggregate comparative performance on average turnaround time (ATAT) between the two systems over varying numbers of resources.

Beside the improvements on standard performance metrics, the research also ensured that the reliability of the resources was taken into cognisance; in the proposed architecture, a replica has been proposed to handle the replication of process state to all grid resources in case the checkpoint repository may also fail.

# CHAPETER FIVE

# SUMMARY, CONCLUSION AND RECOMMENDATIONS

## 5.1 Summary

This dissertation proposed and implemented an enhanced fault tolerance system using checkpointing mechanism to improve performance of the resource scheduling algorithm in heterogeneous grid computing environments. In this research work, tendency failure was employed alongside the response time and failure rate to determine the best number of checkpoints required and the checkpoint size (checkpoint interval) without having or encountering too much overhead; in a situation where there is a failure, there wouldn't be much loss of computation. The results of the simulation using GridSim showed that keeping the number of resources constant and varying the number of gridlets, improvements of 9%, 11%, and 11% on throughput, makespan and turnaround time, respectively, were achieved while varying the number of resources and keeping the number of gridlets constant, improvements of 8%, 11%, and 9% on throughput, makespan and turnaround time respectively were achieved. These results indicate the potential usefulness of our research contribution to practical grid computing environments.

Beside the achievement in improving the standard performance metrics, the research also ensured that reliability of the resources was taken into cognisance rather than considering it probabilistic.

## 5.2 Conclusion

Grid computing is a distributed and interconnected computer systems whereby the machines share and utilize common resources. Grid computing is a utility and heterogeneous system where users don't care about where the resources reside or where

the computer processes a request. Users only send request of information and have the results delivered within shortest possible time. But the machines or resources may fail at the time of execution which, conventionally, restarts the execution from scratch. The fault tolerance technique was used to address this issue (fault) whereby the checkpointing mechanism was adopted in order to avoid restarting the execution of user's application thereby improving performance. However, having many checkpoints leads to overheads which affect the system's performance. This dissertation addressed this issue overheads attributable to frequent checkponting. Simulation results obtained using the popular GridSim simulator showed modest improvements on performance in terms of makespan, throughput, and turnaround time as compared with Idris (2015). The research also ensured that the reliability of the grid has been significantly addressed. The improvements recorded by this research should be beneficial to sectors like the financial services and insurance industries as well as life sciences research where fault tolerant grid systems are employed.

The major contributions made are:

1. The recovery strategies were implemented by combining checkpointiing and replication;
2. The number of checkpoints were determined prior to the job's execution;
3. The checkpoint size was based on the failure rate, response time, and tendency of failure of the grid resources.
4. The checkpoint file was purged after successive checkpoint has been invoked and successfully passed without resources failed.
5. The performance in terms of makespan, throughput, and average turnaround time were improved.

## 5.3 Recommendations

One of the major drawbacks to this research is allocation and deallocation of memory; the checkpoint states are replicated to all other machines in the grid besides the checkpoint repository in order to ensure reliability and availability of the system. This leads to incurring the cost of saving and restoring the checkpoint state. To address this issue in future research, it is recommended that the limited number of checkpoints repository should be extended. When a checkpoint is invoked, there should be a garbage collection housekeeper that helps cleans unneeded checkpoint states. An algorithm needs to be developed that reduces or probably eliminates the reclamation of the checkpoints that are no longer useful.

## **REFERENCES**:


AbdelShafi, H., Speight, E., and Bennett, K. J. (1999). Efficient userlevel thread migration and checkpointing on Windows NT clusters. *in Proceedings of the 3rd USENIX Windows NT Symposium.* Seattle, Washington.

Agarwal, S., Garg , R., Gupta, M., and Moreira, J. (2004). Adaptive Incremental Checkpointing for massively Parallel Systems.

AlKiswany, S., Ripeanu, , M., Vazhkudai, S. S., and Gharaibeh, A. (2008). stdchk: a checkpoint storage system for desktop grid computing. *in Proceedings of the 28th International Conference on Distributed Computing Systems*, (pp. 613-624).

Altameem, T. (2013). Fault Tolerance Techniques in Grid Computing Systems. *International Journal of Computer Science and Information Technologies*, 5.

Alvisi , L. , Elnozahy, E., Rao, , S., Husain, S. A., and Mel, A. D. (1999, June). An analysis of communication induced checkpointing.

Amoon. (2012, 11). A Fault Tolerant Scheduling System Based on Check pointing for Computational Grids. *Vol. 48*.

Amoon, M. (2011). A DEVELOPMENT OF FAULT-TOLERANT AND SCHEDULING SYSTEM FOR GRID COMPUTING. *Computer Science and Telecommunications*, 52.

Ansel, J. , Arya, K., and Cooperman, G. (2009). DMTCP: transparent checkpointing for cluster computations and the desktop. *in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, (pp. 1-12). Rome, Italy.

Antony, L. T., and Dalya, S. (2010). Dynamic Adaptation of Checkpoints and Rescheduling in Grid computing. *International Journal of Computer Application, VOL.3*.

Bakhta, M., and Ghalem, B. (2014). Abstraction Checkpointing Levels Problems and Solution. *International Journal of Computing*, 12.

Bashir, A., and Kumar , A. (2013). Fault Tolerance Issues in Real Time Systems with Energy Minimization. *International Journal of Information and Computation Technology, 3*, 1001-1008. Retrieved 05 20, 2016, from http://www. irphouse.com /ijict.htm

Bronevetsky, G., Marques, D., Pingali, K., and Stodghill, P. (2003). A system for automating applicationlevel checkpointing of MPI programs. *in Proceeding of the 16th International Workshop Languages and Compilers for Parallel Computing (LCPC 2003).* College Station, TX, USA.

DeVale, J. (1999). *Checkpoint-Recovery.* Dependable Embedded Systems .

Dieter, R. W., and Lumpp, E. J. (1999). A userlevel checkpointing library for POSIX threads programs. *in Proceedings of the Twenty-Ninth Annual International Symposium on FaultTolerant Computing (FTCS'99)*, (pp. 224–227). Madison, Wisconsin.

Dieter, R. W., and Lumpp, E. J. (2001). Userlevel checkpointing for Linux: threads programs. *in Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, (pp. 81–92). Boston, Massachusetts, USA.

Fontes, V., Schulze,, B. , Dutra , M., and Porto, F. (2004). Checkpointingbased rollback recovery for parallel applications on the InteGrade Grid Middleware. *in Proceeding of the 2nd Workshop on Middleware for Grid Computing*, (pp. 18-22,). Toronto, Ontario, Canada.

Gioiosa, R., Sancho, C. J., Jiang, S., and Petrini, F. (2005). Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. *in Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing.* Seattle, WA, USA.

Gokuldev, S., and Valarmathi, M. (2013, 04 10). Fault Tolerant System for Computational and Service Grid. *International Journal of Engineering and Innovative Technology (IJEIT), Volume 2*( Issue 10).

Gokuldev, S., & Radhakrishnan, R. (2014). An Adaptive Job Scheduling with efficient Fault Tolerance Strategy in Computational Grid. *International Journal of Engineering and Technology*, 6.

Gokuldev, S., and Valarmathi, M. (2013). Global Load Balancing and Fault Tolerant Scheduling in Computational Grid.

Gupta, P. (2011). Grid computing and checkpoint approach. *IJCSMS International Journal of Computer Science & Management Studies, VOL. 11, Issue 01*, 2231– 5268.

Hargrove, H. P., and Duell, C. J. ( 2006). Berkeley lab checkpoint/ restart (BLCR) for Linux clusters. *in Proceedings of SciDAC.* Denver, CO.

Harshadkumar, B. P., and Vipul, A. S. (2014). Scheduling in Grid Computing Environment. *Advanced Computing & Communication Technologies (ACCT).* India: IEEE. Retrieved 02 2016, from http://ieeexplore.ieee.org/document/6783470/?arnumber=6783470

Idris, H. (2015). *An Improved Job Scheduling Algorithm in Grid Computing Environment Using Fault Tolerance Mechanisms.* Zaria: Ahmadu Bello University, Zaria.

Jiang, H., Chaudhary, V., and Walters, J. (2003). Data conversion for process/thread migration and checkpointing. *in Proceedings of the 32nd International Conference on Parallel Processing (ICPP 2003)* (pp. 473 - 480). Kaohsiung, Taiwan: IEEE.

Koch, D., Haubelt, C., and Teich, J. (2007). Efficient hardware checkpointing, concepts, overhead analysis, and implementation. *in Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2007)*, (pp. 188–196). Monterey, CA.

Koch, D., Haubelt, C., and Teich, J. (2007). Efficient hardware checkpointing, concepts, overhead analysis, and implementation. *in Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2007)*, (pp. 188–196). Monterey, CA.

Krishnan, S., and Gannon, D. (2004). Checkpoint and restart for distributed components in XCAT3. *in Proceedings of the 5th International Workshop on Grid Computing (GRID'2004)*, (pp. 281-288). Pittsburgh, PA, USA.

Latchoumy, P., and Sheik, A. K. (211). Survey on Fault tolerance in Grid Computing. *International Journal of Computer Science & Engineering Survey, Vol.2*, No.4.

Leon, J., Fisher, L. A., and Steenkiste, P. ( 1993). *Failsafe PVM: a portable package for distributed programming with transparent recovery.* Technical report in Carnegie Mellon University.

Li, C., Stewart, E., and Fuchs, W. (1994). Compilerassisted full checkpointing. *Journal Software-Practice and Experience*, 871–886.

Li, X., and Yeung, D. (2008). Exploiting applicationlevel correctness for lowcost fault tolerance. *Journal of Instruction-Level Parallelism*, 1–18.

Lin, Y.B. (2002). Design issues for optimistic distributed discrete event simulation. *Journal of Parallel and Distributed Computing*, 62.

Lyon, B. (1984). *Sun external data representation specification.* Technical report RFC-1832, SUN Microsystems.

Malarvizhi, N., and Uthariaraj, R. V. (2010). FAULT TOLERANT SCHEDULING STRATEGY FOR COMPUTATIONAL GRID ENVIRONMENT. *International Journal of Engineering Science and Technology*, 4361-4372.

Maria, C., Filip, H. A., and Bart, D. (2009). Adaptive Task Check pointing and Replication: Toward Efficient Fault-Tolerant Grids. *IEEE Transaction on Parallel and Distributed Systems, No. 2*, Vol. 20.

Ndiaye, N. M., Sens, P., and Thiare, O. (International Journal of Artificial Intelliegence and Interacrtion Multimedia). Performance Comparison of Hierarchical Checkpoint Protocols Grid Computing. 8.

Osman, S., Subhraveti, D., Su, G., and Nieh, J. (2002). The design and implementation of zap: a system for migrating computing environments. *in Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002).* Boston, Massachusetts, USA.

Perrone, L. F., Wieland, F. P., Liu, J., Lawson, B. G., and Fujimoto, R. M. (2010). Incremental checkpointing with application to distributed discrete event simulation. *Proceedings of the Winter Simulation Conference*, (pp. 1004–1011). Monterey, California, USA.

Petrini, F., Davis, K., and Sancho, C. J. (2004). Systemlevel faulttolerance in largescale parallel machines with buffered coscheduling. *in Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004).* Santa Fe, New Mexico, USA.

Plank, J., Beck, M., and Kingsley, G. (1995). Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments, 7*, 10–14. Retrieved 06 25, 2016, from http://web.eecs.utk.edu/~plank/plank/papers/IEEETCOS95.pdf

Prashar, T., Nancy, and Dinesh, K. (2014). Fault Tolerance ACO using Checkpointing in Grid Computing. *International Journal of Computer Applications*, 6.

Prvulovic, M., Zhang, Z., and Torrellas, J. (2002). ReVive: Costeffective architectural support for rollback recovery in sharedmemory multiprocessors. *in Proceedings of 29th International Symposium on Computer Architecture (ISCA 2002)*, (pp. 111–122). Anchorage, AK, USA.

Ramkumar, B., and Strumpen, V. (1997). Portable checkpointing for heterogeneous architectures. *in Proceedings of the Twenty Seventh Annual International Symposium on Fault-Tolerant Computing, (FTCS-27)*, (pp. 59-67). Seattle, Washington, USA.

Rieker, M., Ansel, J., and Cooperman, G. (2006). Transparent userlevel checkpointing for the native posix thread library for Linux. *in Proceeding of the PDPTA'2006*, (pp. 492–498 ). Las Vegas, Nevada, USA.

Ritu, G., and Awadhesh, K. S. (2011). Fault Tolerance in Grid Computing: State of the Art and Open Issues. *International Journal of Computer Science & Engineering Survey (IJCSES)*, 10.

Rodriguez, G., Martin, M., Gonzalez, P., & Tourio, J. (2010). CPPC: a compilerassisted tool for portable checkpointing of messagepassing applications. *Journal Concurrency and Computation: Practice and Experience*, 749–766.

Shahana, M. (2013). Global Load Balancing and Fault Tolerant Scheduling in Computational Grid.

Samer, A. K., Matei, R., Sudharshan, S. V., and Abdullahi, G. (2013). A Checkpoint Storage System for Desktop Grid Computing. *Oak Ridge National Laboratory*, 12.

Sancho, , J. C., Petrini, , F., Davis, K., Gioiosa, R., and Jiang, S. (2005). Current practice and direction forward in checkpoint/restart implementation for fault tolerance. *in Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005).* Denver, CO, USA.

Sancho, J. C., Petrini, F., Davis, K., Gioiosa, R., and Jiang, S. (2005). Current practice and direction forward in checkpoint/restart implementation for fault tolerance. *in Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005).* Denver.

Silva , M. L., and Silva, G. J. (1998). Systemlevel versus userdefined checkpointing. *in Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, (pp. 68–74). West Lafayette, Indiana.

Siva, S. S., and Syam, K. B. (2010). Survey of fault tolerant techniques for grid. *Computer Science Review*, 101-120.

Sorin, J. D., Martin, K. M., Hill, D. M., and Wood, A. D. (2002). SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *in Proceedings of 29th International Symposium on Computer Architecture (ISCA'2002)*, (pp. 123-134). Anchorage, AK, USA.

The Gridbus Project, GRIDS Lab, The University of Melbourne. (2002, 02 04). *GridSim*. Retrieved 06 13, 2016, from A Grid Simulation Toolkit for Resource Modelling and Application Scheduling for Parallel and Distributed Computing: www.buyya.com/gridsim

Tullmann, P., Lepreau, J., Ford, B., and Hibler, M. ( 1996). Userlevel checkpointing through exportable kernel state. *in Proceeding of the International Workshop on Object Oriented Operating System.* Seattle, Washington.

Vijay, K. N., Samuel, P. M., and Jose, E. M. (2014). A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems. *Technical Paper*, 19.

Walters, P. J., and Chaudhary, V. (2006). Applicationlevel checkpointing techniques for parallel programs. *in Proceedings of the Third International Conference on Distributed Computing and Internet Technology (ICDCIT'06)*, (pp. 221–234). Bhubaneswar, India.

Wang, Z., Shi, X., Jin, H., Wu, S., and Chen, Y. (2014). Iteration based collective I/O strategy for parallel I/O systems. in Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (pp. 287 - 294). IEEE.Zhou, H., and Geist, A. (1995).

Receiver makes right data conversion in PVM. *in Proceedings of the IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications* (pp. 458-464). Scottsdale, AZ: IEEE.